AMS/MAA | ANNELI LAX NEW MATHEMATICAL LIBRARY

Exploring Mathematics with Python

Arthur Engel
Andrew Davison

Preface

The heart of this book is Arthur Engel's 1993 textbook *Exploring Mathematics with your Computer* which utilizes Turbo Pascal for its exploratory work, and is still available for purchase at `https://bookstore.ams.org/NML/35`. The first six chapters (and Appendix A) are essentially that book, but with the programming language changed to Python, some rewording, reformatting in Latex, and a few additions. Those chapters masterfully cover over 60 topics, that are mostly independent of each other.

Sadly, Engel passed away in 2022, and so the revisions were carried out by the second author (Andrew Davison). I'm also responsible for the new chapters, a potpourri of math topics, starting with fractals in chapter 7.

Engel rightly claimed that his book was a mathematics, not a programming, text. However, one of the shifts in programming since the 1990's is the appearance of a vast number of math coding libraries. For example, in Python we have Matplotlib, Numpy, SciPy, Sympy, and many more. In addition, hardware has advanced so much that an average laptop is certain to have multiple CPUs, a GPU, and gigabytes of RAM at its disposal.

Taking advantage of this increased functionality requires a deeper knowledge of computing than was necessary back in the 1990's. As a result, I would classify this version of the book as a mathematics *and* programming text.

The increased complexity means that this book is probably most suitable for readers with a year's exposure to maths and programming at the university level. However, I hope that much of the material is understandable by advanced high school students who have studied additional maths and been introduced to Python. This latter point should be emphasized – I assume that the reader already knows Python; this is not a 'Learn Python' book, although I've included a short refresher appendix.

One of my aims is to try to retain Engel's emphasis on maths by keeping the programming relatively simple, so that even someone with a short introduction to Python should be able to follow (most of) the code. As a consequence, I've deliberately excluded advanced scientific Python modules such as Numpy, SciPy, and Sympy. However, I've relented over charts, and made extensive use of Matplotlib

(which is introduced in its own appendix). The other visualization library that's heavily represented is Python's turtle module, which most students will have met during their introduction to Python, but there's an appendix on it if you need to brush up. Nevertheless, you will encounter some tricky programming topics: recursion, Python list comprehensions, generators, classes, higher-order functions, and references to big-Oh running times. They're not a major element, but they do crop up.

One issue that I've been uncertain about is the use of "I" versus "We". This book would have been impossible without Engel's inspiration, and the first six chapters are his. However, should I use "We" for the later chapters which haven't benefited from input by Engel? As you'll see, I've gone with "We", but any errors or poorly explained material are solely the responsibility of Andrew Davison.

I would like to thank the Mathematical Association of America (MAA) for permission to update Engel's book, and my regards also to Ralf Engel, Arthur's son, for his encouragement.

Unfortunately, after lengthy discussions with the MAA, my hopes of publishing this (rather large) expansion have proved impossible, and so I've decided to put it online, hopefully to be of use to others.

Arthur Engel
Andrew Davison

Contents

Preface v

1 Introductory Problems 1
- 1.1 The Factorial 1
- 1.2 Fibonacci's Sequence: A Two-Term Recursion 9
- 1.3 The Tower of Hanoi 23
- 1.4 Derangements 27
- 1.5 The Bisection Method 32
- 1.6 Gambling Boldly 35
- 1.7 The Josephus Problem 37
- Exercises for Sections 1.1 to 1.7 42

2 Algorithms in Number Theory 47
- 2.1 Greatest Common Divisor 47
- Exercises 57
- 2.2 Representing n in the Form $x^2 + y^2$ 61
- 2.3 Pythagorean Triples 62
- 2.4 Counting the Lattice Points in a Ball 63
- Exercises for Sections 2.2 to 2.4 66
- 2.5 Sieves 67
- Exercises for Section 2.5.4 72
- Exercises for Section 2.5 73
- 2.6 Rotation of a List 77
- 2.7 Partitions 79
- 2.8 The Money Changing Problem 81
- 2.9 An Abstractly Defined Set 82
- 2.10 Recursive Pile Splitting 84
- 2.11 The Stern-Brocot Sequence 85
- 2.12 The Best Rational Approximation 90
- Exercises for Sections 2.6 to 2.12 91
- 2.13 Primes 95
- 2.14 n as a Sum of Four Squares 109

Exercises for Sections 2.13 to 2.14 111
2.15 The Maximum of a Unimodal Function 113
2.16 A Very Difficult Problem 117
Exercises 118

Additional Exercises for Chapters 1 to 2 120

3 Probability 127
3.1 A Pseudo-Random Number Generator 127
3.2 Python's PRNGs 129
3.3 Random Walks 130
3.4 Problems Involving Random 1D Walking 132
3.5 Self Avoiding Random Walks 140
Exercises 143
3.6 Finding Geometric Probabilities by Simulation 144
Exercises 148
3.7 The Bertrand Paradox 151
3.8 The Broken Stick Problem 155
3.9 The Monty Hall Problem 158
3.10 Random Choice of an s-Subset from an n-Set 160
Exercises 163
3.11 Coin Tosses in Software 163
Exercise 168
3.12 Shift Registers: More Coin Tosses 168
Exercises 170
3.13 Random Sequence Generation by Cellular Automata 172
Exercise 173
3.14 The Period Finding Problem 173
3.15 Shrinking Squares 175
Exercises 180

Additional Exercises for Chapters 1 to 3 182

4 Statistics 187
4.1 The Subset Sum Problem (Part 1) 187
4.2 Matched Pairs 190
4.3 P-value by Simulation 194
4.4 P-value by Computation 195
Exercises for Sections 4.1 to 4.4 196
4.5 The Two Sample Problem of Wilcoxon 197
4.6 The General Two Sample Test 204
Exercises for Sections 4.5 to 4.6 205
4.7 Kendall's Rank Correlation 206

Exercises 211
4.8 The Binomial Distribution 212
Exercises 216
4.9 Hypergeometric Distribution 218
Exercises 222
4.10 Fishing Laws 224
Exercises 225
Exercise 227
4.11 Probability of Not Being Lonesome 227
4.12 Runs 228
Exercises 235
4.13 Waiting for a Complete Set 236
Exercises 239
4.14 The Birthday Problem (and Factoring) 240
Exercise 246

Additional Exercises for Chapters 3 to 4 247

Additional Exercises for Chapters 1 to 4 250

5 Combinatorial Algorithms 251
5.1 Sorting 251
Exercises 260
5.2 The kth Smallest Element of a List 264
5.3 Binary Search 265
Exercises 270
5.4 Binary Questions 270
Exercises 270
5.5 Recreational Search 271
Exercises 286
5.6 Permutations 288
Exercises 292
5.7 Coupled Difference Equations 294
Exercises 296
5.8 Games 296
Exercises 302
5.9 The Subset Sum Problem (Part 2) 304
Exercises 314

6 Numerical Algorithms 317
6.1 Powers with Integer and Real Exponents 317
Exercises 319
6.2 Design of a Square Root Procedure 319

	Exercises	323
6.3	The Natural Logarithm	327
6.4	The Inverse Trigonometric Functions	329
	Exercise	331
6.5	The Function exp()	331
6.6	The Cosine	332
6.7	One Million Decimals of e and π	334
	Exercises	353
6.8	Least-Squares Curve Fitting	353
	Exercises	363
6.9	Polygon Circumscribing	363
6.10	Archimedes' Integration of the Parabola	365
6.11	Numerical Integration	367
6.12	Richardson Extrapolation	372
6.13	Romberg Integration	373
	Exercises	376
6.14	Problems of Pursuit	376
	Exercises	385
6.15	A Rumor Spreads Through the Town	387
	Exercises	391
6.16	Eat and be Eaten: the Struggle for Existence	392
	Exercises	395

Additional Exercises for Chapters 1 to 6 396

7 Fractals 403

	Why Study Fractals?	403
7.1	Introduction	404
7.2	The von Koch Curve	404
7.3	Fractal Dimensionality	406
7.4	Measuring Fractals	406
7.5	The Sierpinski Gasket	408
7.6	Other Ways to Define Fractals	408
7.7	Coding Fractals	409
7.8	The 2D Hilbert Curve	412
7.9	The 3D Hilbert Curve	416
	Exercises	418

8 Chaos 429

	Why Study Chaos?	429
8.1	Introduction	430
8.2	The Logistic Map	430

8.3 Stability and Period Doubling 436
8.4 The Lyapunov Exponent 438
8.5 The Feigenbaum Constants 439
8.6 Self-similarity Again 444
8.7 The Lorenz System 445
8.8 A Poincaré Section 448
Exercises 450

9 Computer Graphics Transformations 453
Why Study Computer Graphics Transformations? 453
9.1 Introduction 453
9.2 2D Transformations 454
9.3 Homogeneous Coordinates 455
9.4 Composition of 2D Transformations 459
9.5 Transforming a Knee 460
9.6 Multiple Transformations 465
9.7 Baravelle Spirals 466
9.8 Hilbert Curves Again 468
9.9 Matplotlib's Affine2D Classes 470
9.10 Affine2D Images 472
9.11 Affine2D Paths 473
9.12 Henderson's Functional Geometry 475
9.13 Hierarchy of Transformations 477
9.14 3D Transformations as Matrices 480
9.15 Composition of 3D Transformations 482
9.16 Recursive Cube Drawing 485
Exercises 488

10 Computational Geometry 493
Why Study Computational Geometry? 493
10.1 Introduction 493
10.2 Geometry Data Structures 494
10.3 The Dot and Cross Products 495
10.4 The ccw() function 498
10.5 Segment Intersection 499
10.6 Finding the Convex Hull 502
10.7 Finding the Closest Pair of Points 506
Exercises 510

11 Continued Fractions 513
Why Study Continued Fractions? 513
11.1 Introduction 513

11.2 Continued Fractions and Euclid's Algorithm 520
11.3 Continued Fractions for Square Roots 520
11.4 The Law of Best Approximation 522
Exercises 524

12 Discrete-Time Markov Chains 527
Why Study Discrete-Time Markov Chains (DTMCs)? 527
12.1 Introduction 527
12.2 *n*-Step Transitions 531
12.3 Simulation 533
12.4 Classification of States 536
12.5 Steady-State Behavior 537
12.6 The Two Urns 539
12.7 Absorption 540
12.8 Finding Absorbing Probabilities 542
12.9 Absorption Time and Probabilities 544
12.10 Implementing Absorption 545
12.11 Implementing Steady State Using Matrices 546
Exercises 548

13 Navigating the Earth 553
Why Study Spherical Trigonometry? 553
13.1 Introduction 553
13.2 Properties of the Sphere 554
13.3 Curves on a Sphere 555
13.4 Mapping a Sphere 555
13.5 Spherical Triangles 557
13.6 The Spherical Sine Law 559
13.7 Triangles made from Lunes 560
13.8 Are Great Circle Arcs the Shortest Distance? 562
13.9 Spherical Triangles Summarized 564
13.10 Calculating with Great Circles 564
13.11 The Mercator Projection 572
13.12 Calculating with Rhumb Lines 579
Exercises 585

14 Quaternions and Rotation 589
Why Study Quaternions? 589
14.1 Introduction 589
14.2 3D Rotation 590
14.3 Vectors Again 591
14.4 `Vec`: A vector class in Python 594

14.5 Rodrigues' Rotation Formula (RRF) 596
14.6 Quaternions 598
14.7 Rotations with Quaternions 602
14.8 A Quaternion class in Python 603
14.9 Spherical Linear Interpolation (Slerp) 606
14.10 Visualizing Lerp and Slerp 611
14.11 Navigating Again 613
14.12 Spherical Cubic Interpolation (Squad) 615
Exercises 619
Answers 620

15 Curve Fitting 623
Why Study Curve Fitting, Splines and Bézier curves? 623
15.1 Introduction 623
15.2 Newton Interpolation 624
15.3 Least Squares Curve Fitting 627
15.4 Cubic Spline Interpolation 628
15.5 The Bézier Curve 635
Exercises 649
Answers 650

16 Conic Sections 653
Why Study Conic Sections? 653
16.1 Introduction 653
16.2 A Geometric Definition 654
16.3 A Locus-Distance Definition 656
16.4 Classifying Conic Sections by Eccentricity 664
16.5 Polar Coordinates 666
16.6 Ellipses, Parabolas, and Hyperbolas Unified 666
16.7 Drawing Conic Sections using their Eccentricity 668
16.8 Dandelin Spheres 671
16.9 Quadratic Equations 675
16.10 Parameterized Conic Sections 680
16.11 Drawing Parameterized Equations 683
16.12 Reflections in Conic Sections 685
16.13 Confocal Conic Sections 693
Exercises 700
Answers 703

17 Solving a Cubic Equation 707
Why Study and Solve Cubic Equations? 707
17.1 Introduction 707

17.2 Depressing Cubic Equations 708
17.3 The Tartaglia-Cardano's Method 709
17.4 Viète's Method 713
17.5 Implementing Cardano's and Viète's Methods 715
17.6 Visually Finding Roots in a 2D Plot 718
17.7 Visually Finding Roots with a Modulus Surface 720
17.8 Visually Finding Roots with Enhanced Phase Maps 722

18 The Catenary 725
Why Study the Catenary? 725
18.1 The Catenary Chain 725
18.2 A Parabolic Chain 733
18.3 Visually Comparing the Catenary and Parabola 735
18.4 Determining the Catenary Parameter a 740
18.5 Using a Catenary to Calculate Logarithms 741
18.6 Rolling a Parabola 743
18.7 The Tractrix 746
18.8 Riding a Bike with Square Wheels 750
18.9 The Catenoid 757
Exercises 763
Answers 764

19 The Cycloid 767
Why Study the Cycloid? 767
19.1 The Cycloid 767
19.2 The Tautochrone 773
19.3 A Cycloid's Involute 777
19.4 The Brachistochrone 781
19.5 Epicycloids 786
19.6 Hypocycloids 790
19.7 Linking Hypocycloids and Epicycloids 793
19.8 Squaring the Circle 794
Exercises 795
Answers 796

20 Function Approximation 805
Why Study Function Approximation? 805
20.1 Dirty Functions as Motivation 806
20.2 Dirty Square Root 807
20.3 Dirty Log Base 10 812
20.4 Dirty Arctangent 820
20.5 Dirty Sine 824

20.6 The Maths of Function Approximation 828
20.7 The Taylor Series 829
20.8 The Lagrange Polynomial 842
20.9 Orthogonality 849
20.10 Chebyshev Approximation 854
20.11 Padé Approximation 861
Exercises 870
Answers 873

A A Crash Course in Probability 875
A.1 A Bold Gamble 879
A.2 Random Variables and Expectation 880
A.3 More on Random Walks 881
A.4 Five Practice Problems 882
A.5 More Information 885

B Solutions to Some Exercises 886
B.1 Sections 1.1 to 1.7 886
B.2 Section 2.1 889
B.3 Sections 2.2 to 2.4 893
B.4 Section 2.5 893
B.5 Section 2.5.4 893
B.6 Sections 2.6 to 2.12 894
B.7 Sections 2.13 to 2.14 895
B.8 Section 2.16 896
B.9 Chapters 1 to 2 896
B.10 Section 3.10 898
B.11 Section 3.11 898
B.12 Section 3.12 898
B.13 Section 3.15 899
B.14 Chapters 1 to 3 899
B.15 Sections 4.5 to 4.6 901
B.16 Section 4.13 901
B.17 Section 4.14 902
B.18 Chapters 3 to 4 902
B.19 Chapters 1 to 4 902
B.20 Section 5.4 903
B.21 Section 5.5.5 903
B.22 Section 5.6 904
B.23 Section 5.8 905
B.24 Section 6.2 905
B.25 Section 12 905

B.26 Chapters 1 to 7 909

Appendices 875

C Summary of Notation and Formulas 913

D A Summary of Python 915
D.1 Comments 915
D.2 Variables 915
D.3 Operators 916
D.4 Strings 916
D.5 I/O 917
D.6 Lists 917
D.7 Tuples 919
D.8 Dictionaries 919
D.9 Sets 919
D.10 If Statements 920
D.11 Loops 920
D.12 Functions 921
D.13 Exceptions 923
D.14 File I/O 923
D.15 Modules 923
D.16 Classes 924
D.17 Three Examples 924

E Numbers in Python 929
E.1 The int and float Types 929
E.2 The math Module 930
E.3 Numerical Errors 930
E.4 Representing Reals 931
E.5 Avoiding Float Problems 943
Exercises 947

F Matplotlib 950
F.1 The Big Three 952
F.2 Common Chart Variants 954
F.3 Customizing a Plot 957
F.4 Other Useful Plot Types 961
F.5 3D Plots 968
F.6 Animation 971
F.7 Widgets 973

G Turtle Graphics 975

G.1 Drawing Polygons 976
G.2 Responding to Mouse Clicks 977
G.3 Beautiful Recursion 978
G.4 Multiple Turtles 979
G.5 LOGO Flowers 980

H Graphviz 984
H.1 A Simple Undirected Graph 984
H.2 Using Nodes 986
H.3 More Layouts 990
H.4 More Attributes 991
H.5 Clusters: An Automaton 993
H.6 More Shapes 995

I Matrices 998
I.1 What is a Matrix? 998
I.2 Equality 999
I.3 Addition and Subtraction 999
I.4 Multiplication 1000
I.5 Transpose of a Matrix 1002
I.6 Special Matrices 1003
I.7 Determinant of a Square Matrix 1004
I.8 Inverse of a Square Matrix 1006
I.9 Solution of a Set of Linear Equations 1006
I.10 Gaussian Elimination 1007
I.11 Matrix Inversion Using Gaussian Elimination 1008

J Complex Numbers and `cmath` 1013
J.1 Multiplication in Polar Form 1015
J.2 De Moivre's Formula 1015
J.3 The *n*th Root of a Number 1016
J.4 Euler's Formula 1017
J.5 Multi-valued Exponentiation 1019
J.6 Complex Numbers in Python 1021

Bibliography 1029

Authors Bio 1035

1

Introductory Problems

This chapter is introductory in the sense that it focuses on the parts of Python that confuse beginners, and also on visualization tools (matplotlib and turtle). One of the most confusing areas of Python, which it shares with other programming languages, is recursion, which has motivated our choice of factorial, Fibonacci, the Towers of Hanoi, and the other examples. Another area, more specific to Python, are the use of generators. Other topics discussed here include a comparison of recursion vs. iteration, different ways of measuring running time, techniques for reducing running time, and integers vs. reals (floats and decimals in Python).

1.1 The Factorial

The factorial function is defined on non-negative integers as:

$$1! = 1, \quad n! = n(n-1)!$$

This can be readily translated into the recursive function in Listing 1.1.

```
def factRec(n):
  if n <= 1:
    return 1
  else:
    return n*factRec(n-1)
```

Listing 1.1. Recursive factorial

Note that this is not an exact conversion since the base case (the if-case) has been modified to handle 0 and negative numbers (`factsLib.py`).

factRec() can be called like so from inside Python's IDE:

```
>>> from factsLib import *
>>> factRec(5)
120
```

A series of function instantiations deal with progressively decreasing values, until the base case is encountered at $n = 1$ (see Fig. 1.1). These are stored as *stack frames* by the run time system until later instantiations complete and return values to them.

One problem for a beginner is recursion's dynamic nature – Fig. 1.1 only depicts the moment when the maximum number of function instantiations exist, just prior to the base case returning. The parent function will resume, return a value, and disappear.

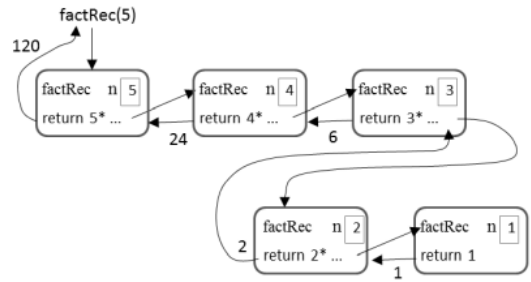


Figure 1.1. Executing fact(5)

Another difficult aspect is that the call to factRec(5) eventually triggers the creation of five copies of the function's n variable, whereas only a single n is present in the factRec() source. Indeed, the number of variables created at run time varies linearly with the size of the initial n.

1.1.1 Iterative Factorials.

It's possible to define factorial as a product:

$$n! = \prod_{i=1}^{n} i$$

This is easily translated into iterative code (factsLib.py), although there are several possibilities:

```
def factIter(n):
  fact = 1
  while n > 0:
    fact *= n
    n -= 1
  return fact

def factIterUp(n):
  fact = 1; count = 0
  while count < n:
    count += 1
    fact *= count
  return fact

def factRange(n):
  fact = 1
  for i in range(1, n+1):
    fact *= i
  return fact
```

Listing 1.2. Three iterative factorials

They all produce the same result as the recursive version:

```
>>> factIter(5)
120
>>> factIterUp(5)
120
>>> factRange(5)
120
```

But this may not be the case when n is large:

```
>>> factRange(1000)
402387260077093773543702433923003985719374864210 71
463254379991042993851239862902059204420848696 9...
700277534720000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
>>>
>>> factRec(1000)
Traceback:
  File "factsLib.py", line 14, in factRec
    if n < 2:
RecursionError: maximum recursion depth exceeded in comparison
```

The space used to represent Python integers automatically grows, so factRange() has no trouble calculating 1000!. This is also the case for factRec(), but it falls foul of Python's default recursion depth limit (100). The depth required by factRec() depends linearly on the size of n, and so factRec(1000), needing at most 1000 stack frames, crashes.

The depth limit can be changed:

```
>>> import sys
>>> sys.setrecursionlimit(10**6)
>>> factRec(1000)
402387260077093773543702433923003985719374864210 71
463254379991042993851239862902059204420848696 9...
700277534720000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
>>>
```

Another problem that may arise in Python version 3.10.7 or later is due to its integer string conversion length limit, which defaults to 4300 characters. It will crop up when a very large factorial is converted to a string before being printed:

```
>>> factRange(10000)
Traceback:
ValueError: Exceeds the limit (4300 digits) for
integer string conversion;
use sys.set_int_max_str_digits() to increase the limit
```

This limit can be removed:

```
>>> import sys
>>> sys.set_int_max_str_digits(0)
>>> factRange(10000)
28462596809170545189...
   :  # many lines not shown
000000000000000
```

1.1.2 Recursion vs. Iteration. Recursive and iterative programs can be compared by looking at their running times and amount of memory used. factCompare.py employs Python's time and tracemalloc modules, running factRec() and factRange() with a range of values.

The times and memory allocations are plotted in the graphs shown in Fig. 1.2. (If you're unfamiliar with Python's Matplotlib module, please read Appendix F.)

The left-hand chart shows that there's essentially no difference between the recursive and iterative running times; both are O(n). The significant difference is highlighted by the right-hand graph – the memory used by the iterative code is constant (it only requires two variables and a range() iterator), whereas the memory used by the recursive function increases linearly since a new stack frame is created for each recursive call. This was illustrated in Fig. 1.1.

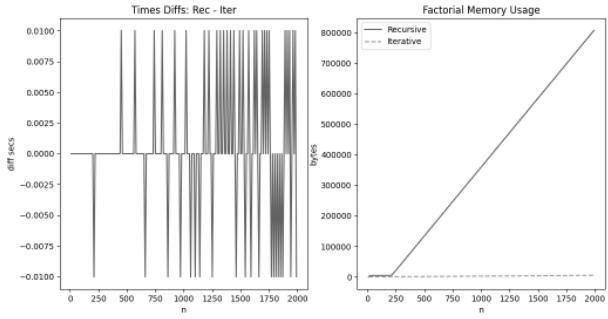


Figure 1.2. Time and memory usage comparisons

1.1.3 Functional Factorials. Python utilizes a number of functional programming ideas, including list comprehensions, anonymous functions, and higher-order operators such as map(), reduce() and apply(). This allows (at least) two other ways to write factorial (factsLib.py):

```
def factProd(n):
  return math.prod(range(1, n+1), start=1)

def factReduce(n):
  return reduce(lambda x, y: x * y, range(1, n+1), 1)
```

Listing 1.3. Two functional factorials

The product definition for factorial is implemented succinctly using reduce() in factReduce(). It applies the supplied function (multiplication) to an iterable, reduces it to a single cumulative value.

factTimings.py calculates the average running times for all of these variants (factRec(), factIter(), factRange(), factProd(), factReduce()) and Python's own built-in, math.factorial(). Unlike the timing code in Listing 1.2 (factCompare.py), Python's timeit module is employed (`https://docs.python.org/3/library/timeit.html`). It provides a convenient way to run functions several times over, in order to calculate an average time.

The resulting chart is in Fig. 1.3 (factTimes.py). The unsurprising result is that math.factorial() is the fastest (it's a small wrapper around a high-speed C library). Also, range() is faster than a while-loop, but the timing difference is marginal. Although the reduce and recursive versions are arguably the closest to the math definitions, they're also the slowest.

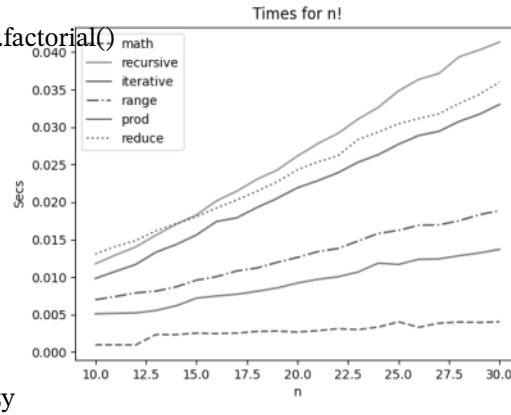


Figure 1.3. Factorial times

1.1.4 Lazy Evaluation. Lazy evaluation debuted in the lambda calculus, and was first utilized in functional programming languages. It lets the evaluation of an expression be suspended until required by another part of the computation.

Lazy evaluation can be employed in Python with *generator* functions. Generator code looks like an ordinary function but utilizes 'yield' rather than 'return'. When the generator is invoked, an iterator is created, which will yield a new result whenever it is called with next().

factGen() (factsLib.py) shown below is an *infinite* generator since it can keep producing factorials forever (if requested).

```
def factGen():
  fact = 1
  i = 0     # fact(0) = 1
  while True:
    yield fact
    i += 1
    fact *= i
```

Listing 1.4. A factorial generator

```
>>> fg = factGen()
>>> next(fg)
1
>>> next(fg)
1
>>> next(fg)
2
>>> next(fg)
6
>>> next(fg)
24
```

The call to factGen() creates an iterator (called fg), which returns its first value when next() is called. The iterator wakes up and executes up to the yield line which returns the fact value *and then suspends*. When next() is called again, the iterator resumes from where it was suspended and progresses until it next encounters a yield.

Python's itertools module offers higher-level operations for working with iterators, many of which will be familiar to functional programmers (`https://docs.python.org/3/library/itertools.html`). Listing 1.5 (factGens.py) contains a few examples:

```
import itertools
from factsLib import *

n = int(input("n=? "))
print("Fact("+str(n)+"):",next(itertools.islice(factGen(),n,None)))
lessFacts = list(itertools.takewhile(lambda i: i < n*n,factGen()))
print("\nFacts < :", *lessFacts)
print("\nnum Facts: ", *facts(n))
print()
gen = factGen()
for i in range(n+1):
  print(f"{next(gen):12d}", end = '')
  if (i+1)%5 == 0:
    print()
```

Listing 1.5. Using the factorial generator

Listing 1.5 uses itertools.islice() and itertools.takewhile(). islice allows a slice of several values to be returned from the iterator, although our example only gets the n-th value. takewhile() keeps extracting data from an iterator while its condition (in this case, $i < n^2$) is true.

The third example calls facts(n), a function from factsLib.py, which constructs a list of the first n numbers of factGen(). The code at the end of Listing 1.5 shows how to embed next() inside a loop.

```
> python factGens.py
n=? 5
Fact(5): 120
Facts < : 1 1 2 6 24
num Facts:  1 1 2 6 24 120
```

1.1.5 Factorial Approximations. Stirling's series (`https://mathworld.wolfram.com/StirlingsSeries.html`) is an asymptotic approximation for n!:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{51840n^3} + O(\frac{1}{n^4})\right)$$

The approximation error is equal to the first omitted term, which for the above is $O(\frac{1}{n^4})$. Another source of errors are the use of floats to represent π, square roots, and real-number division; see Appendix E for more discussion of these issues.

Listing 1.6 (factsLib.py) contains two implementations of Stirling's approximation: stirling() utilizes only the first term of the series, while stirling2() includes the second.

```
def stirling(n):
  return math.sqrt(2*math.pi*n)*(n/math.e)**n

def stirling2(n):
  return  (1 + 1/(12*n)) * math.sqrt(2*math.pi*n)*(n/math.e)**n
```

Listing 1.6. Two Stirling approximations

factApproxs.py runs these two on a variety of inputs, and charts their *relative* errors compared to math.factorial() (see Fig. 1.4).

Although the relative errors are small, the enormous size of the factorials means there's very few correct digits in the results. For instance, consider 20! and 100!:

```
>>> factRange(20)
2432902008176640000
>>> stirling(20)
2.422786846761135e+18
>>> stirling2(20)
2.432881791955973e+18
```

```
>>> factRange(100)
933262154439441526816992388563...
168640000000000000000000000000
>>> stirling(100)
9.32484762526942e+157
>>> stirling2(100)
9.332618331623811e+157
```

stirling() only manages two correct digits, while stirling2() produces four or five. It might be tempting to switch to an implementation of Stirling's series using decimals (see Appendix E for more on Python's decimal), which replaces float's 15 digit precision with 28 digits (factsLib.py). Let's try it:

```
def stirlingD(n):
  root = (2*PI*D(n)).sqrt()
  return root*((D(n)/E)**D(n))

def stirling2D(n):
  term12 = D(1) + D(1)/(D(12) * D(
      n))
  root = (2*PI*D(n)).sqrt()
  return term12*root*((D(n)/E)**D(
      n))
```

Listing 1.7. Stirling approx using decimals

```
>>> stirlingD(50)
Decimal('3.036344593938155820798372692E+64')
>>> stirlingD(100)
Decimal('9.324847625269343247764756226E+157')

>>> stirling2D(50)
Decimal('3.041405168261386080499703315E+64')
>>> stirling2D(100)
Decimal('9.332618331623734367137893520E+157')
```

The number of digits in the answers increases, but the number of correct digits does not. The reason is that the greater part of the error is due to the series approximations not the use of floating point. In the case of stirling() (and stirlingD()), the error is $O(\frac{1}{n})$ which for n = 100 means an error of about 1%, or two-digit precision. For stirling2() (and striling2D()), the error is $O(\frac{1}{n^2})$ which for n = 100 is an error of 0.01% and a precision of four digits.

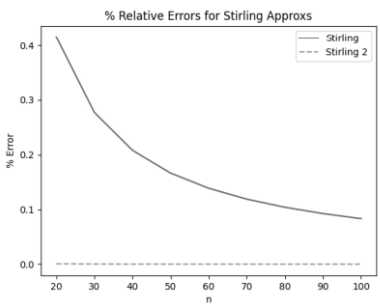


Figure 1.4. Relative errors when using Stirling

1.1.6 Using Log Gamma. In languages that don't have the benefit of Python's unlimited size integers, a commonly suggested alternative for calculating large factorials is to manipulate them

as logarithms. For example, this would replace the division of two massive factorials (e.g. 100!/99!) with the subtraction of two much smaller logarithms (although a wily student would just simplify the expression directly).

Python doesn't have a logarithmic factorial function, but it does offer logarithmic gamma – math.lgamma(). The gamma (Γ) function extends factorial to complex numbers, excluding negative integers, but the factorial of a positive integer n is simply $n! = \Gamma(n+1)$, so that $\log n! = \log \Gamma(n+1)$. 100!/99! can be expressed as:

```
>>> import math
>>> math.exp( math.lgamma(101) - math.lgamma(100))
99.9999999999981
```

Since these functions utilize floats, we can expect them to be accurate to at most 15 digits, as here. This may be satisfactory, but it illustrates that using Python's gamma() or lgamma() is *not* the same as calculating factorial with integers.

In general, it's much better to stick with integer-based calculations if possible. If you must use floats, then be aware of the problems with rounding errors and precision. These issues are explained at some length in Appendix E. Python's decimal type may help, but it depends on the source of the approximation errors (e.g. do they stem from the math, or from the algorithm, or both).

1.2 Fibonacci's Sequence: A Two-Term Recursion

The Fibonacci sequence is given by the recursive relation:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2}, \quad n > 1 \end{aligned}$$

An alternative is to start the base cases at F_1 and define $F_2 = 1$.

Leonardo Pisano (1170-1250), or Leonardo of Pisa, but better known by his nickname, Fibonacci, used the sequence in his *Liber Abaci* ('Book of Calculations') published in 1202, although it was already known to Indian mathematicians as early as 200BC. The series is ubiquitous in nature, and has been eagerly connected to many forms of art and architecture, although some of these seem a little dubious. For a good account, see Vorobev [**Vor11**].

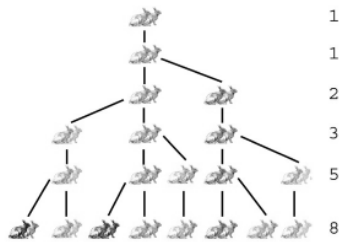


Figure 1.5. Fibonacci's rabbits

The *Liber Abaci* introduced Hindu-Arabic digits to Europe, but also debuted Fibonacci's rabbit problem:

> A pair of rabbits is born and produces a further pair in the second month of its existence, and in each subsequent month. Each succeeding generation reproduces in the same way. How many pairs are there after a year?

The situation is depicted in Fig. 1.5.

A quick calculation produces Table 1.1. After a year (i.e. at the start of the 13th month), there will be 233 pairs of rabbits.

Month	1	2	3	4	5	6	7	8	9	10	11	12	13
No. Pairs	1	1	2	3	5	8	13	21	34	55	89	144	233

Table 1.1. Fibonacci's rabbits

The recurrence relation is readily translated into recursive code (Listing 1.8; fibsLib.py).

```
def fibRec(n):
  if n <= 0:  # base cases
    return 0
  elif n == 1:
    return 1
  else:
    return fibRec(n-1) + fibRec(n-2)
```

Listing 1.8. Recursive Fibonacci numbers

Happily, the function gives the same answer for the number of rabbits:

```
>>> from fibsLib import *
>>> fibRec(13)
233
```

The call graph for fibRec(5) (Fig. 1.6) reveals a worrying duplication of effort in different parts of the execution tree (or *call graph*). Note, we've abbreviated the function name to fib() to make the diagram less cluttered.

fibRec(2) is evaluated three times, and fibRec(3) twice. The amount of repeated work only becomes more serious as the number supplied to the initial fibRec() call is increased.

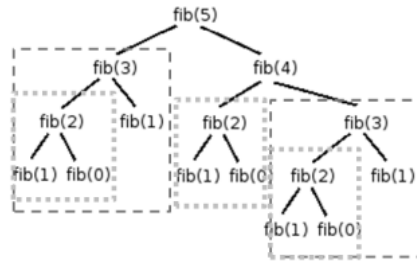


Figure 1.6. Call graph for fibRec(5)

The call graph also gives an indication of the running time for the function. Each call results in *two* further calls, causing a doubling of the computational effort at each level. Of course, once a base case is reached, the calculations stop, and the tree isn't the same height across its breadth. Nevertheless, this doubling leads to an exponential running

time of $O(2^n)$, which will bring any computer to a grinding halt when n is even modestly large.

In one respect the call graph is misleading: it suggests that the entire tree is in memory at once. This is not the case since execution progresses down one branch at a time, reaches a base case, returns, and the call frames are deleted. This means that the memory costs are related only to the length of a single branch, which will be $O(n)$. The exponential problem is confined to the running time.

The running time can be reduced to $O(n)$ by switching to an iterative solution (see Listing 1.9; fibsLib.py). The algorithm relies on the fact that the current Fibonacci number only depends on the two previous values, and so a loop using two variables to store those numbers is sufficient.

```
def fibIter(n):
  a = 0; b = 1  # F0, F1
  if n <= 0:
    return a
  elif n == 1:
    return b
  else:
    for i in range(2, n+1):
      a, b = b, a + b
    return b
```

Listing 1.9. Iterative Fibonacci

1.2.1 Memoization. Memoization is the (rather obvious) idea of storing answers to sub-calculations so that if the same calculations reappear later, then the stored values can be used instead of repeating the computations. fibRec() suffers from a great deal of recalculation, and so will benefit from memoing; the outcome is Listing 1.10 (fibsLib.py).

```
fibsStore = [0, 1]
numCalls = 0

def fibMemo(n):
  global numCalls
  numCalls += 1

  if n < len(fibsStore):
    return fibsStore[n]
  else:
    res = fibMemo(n-1) + fibMemo(n-2)
    fibsStore.append(res)
    return res
```

Listing 1.10. Memoing Fibonacci numbers

fibMemo() is still recursive, with the same structure as fibRec(), but also accesses fibsStore, a global variable that's retained in memory even after fibMemo() returns. The numCalls variable isn't needed for the memoing, and is included to total up the number of function calls made in fibMemo().

Listing 1.11 (fibMemo.py) calls fibMemo() three times: fib(50) is calculated twice, and then fib(60).

```
def useFibMemo(n):
  fibsLib.numCalls = 0
  print("\nfib of", n, "=",
      fibMemo(n))
  print("No of calls:", fibsLib.
      numCalls)
  print("fibsStore size:", len(
      fibsStore))


useFibMemo(50)
useFibMemo(50)
useFibMemo(60)
```

Listing 1.11. Using memoing

```
> python fibMemo.py

fib of 50 = 12586269025
No of calls: 99
fibsStore size: 51

fib of 50 = 12586269025
No of calls: 1
fibsStore size: 51

fib of 60 = 1548008755920
No of calls: 21
fibsStore size: 61
```

Memoization has an immediate benefit in the first call to fib(50). Instead of generating around 2^{50} calls, there's only 99. The savings continue with the second fib(50): since fibsStore is still in memory, there's no need for any calculations.

fib(60) does require new work, but only for the Fibonacci numbers between 51 and 60, which takes linear rather than exponential time.

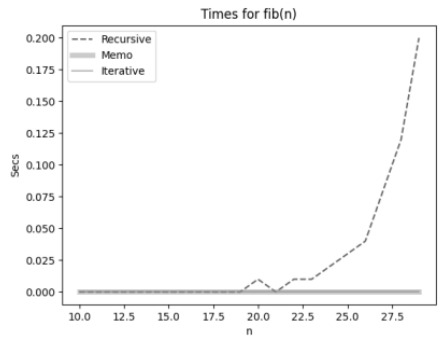


Figure 1.7. Timings for Fib variants

fibTimes.py calls fibRec(), fibIter(), and fibMemo() for a variety of values, and plots the running times. The results are shown in Fig. 1.7.

The plot for fibMemo() is linear (the same as fibIter()) even though the function is recursive. However, the curve for fibRec() has a characteristic exponential rise.

1.2.1.1 Memoization using Lazy Evaluation. Python's lazy evaluation can be utilized as a form of memoization. fibsGen() in Listing 1.12 (fibsLib.py) is a version of fibIter() using yield rather than return.

```
def fibGen():
  # generator that never stops
  current = 0 # first fibonacci numbers
```

```
  b = 1
  while True:
    yield current
    current, b = b, current+b
```

Listing 1.12. Generating Fibonacci numbers

Once the iterator has been created, it stays in memory, waking up when the next Fibonacci number is requested:

```
>>> fg = fibGen()
>>> next(fg)
0
>>> next(fg)
1
>>> next(fg)
1
>>> next(fg)
2
>>> next(fg)
3
>>> next(fg)
5
>>> next(fg)
8
```

However, iterators are *single-use* which means that `fg` can only generate a single sequence of Fibonacci numbers. For instance, we can't request fib(50) twice, or jump from fib(50) to fib(60) without generating all the intermediate numbers with calls to next().

1.2.2 Fibonacci Approximations. The Fibonacci sequence can be defined as a closed-form expression called Binet's formula (`https://en.wikipedia.org/wiki/Fibonacci_sequence#Binet's_formula`), which uses powers of the golden ratio ϕ:

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$$

ϕ is equal to $\frac{1+\sqrt{5}}{2}$.

One way to derive this formula is by considering the quadratic $x^2 - x - 1 = 0$. Its roots are $\frac{1\pm\sqrt{5}}{2}$, so one of them is ϕ. The other, $\frac{1-\sqrt{5}}{2}$, is sometimes called τ.

The quadratic can be rearranged as $x^2 = x + 1$ and then used to produce increasing powers of x^n in terms of multiples of x:

$$\begin{aligned} x^3 &= x \cdot x^2 = x \cdot (x+1) \\ &= x^2 + x = (x+1) + x \\ &= 2x + 1 \end{aligned}$$

$$\begin{aligned} x^4 &= x \cdot x^3 = x \cdot (2x+1) \\ &= 2x^2 + x = 2(x+1) + x \\ &= 3x + 2 \end{aligned}$$

$$\begin{aligned} x^5 &= 5x + 3 \\ x^6 &= 8x + 5 \end{aligned}$$

The prevalence of the Fibonacci numbers is quite noticeable, which lets us make the leap of faith to:

$$x^n = F_n x + F_{n-1}$$

Since ϕ and τ are roots of the quadratic, they must both satisfy x^n, permitting us to say:

$$\phi^n = F_n\phi + F_{n-1} \quad \text{and} \quad \tau^n = F_n\tau + F_{n-1}$$

Subtracting the second equation from the first equation yields:

$$\phi^n - \tau^n = F_n(\phi - \tau) + F_{n-1} - F_{n-1}$$

and so:

$$F_n = \frac{\phi^n - \tau^n}{\phi - \tau} = \frac{\phi^n - \tau^n}{\sqrt{5}}$$

Returning to the quadratic, $x^2 - x - 1 = 0$, it can also be expressed as:

$$\begin{aligned} x - 1 - x^{-1} &= 0 \\ \frac{1}{x} &= x - 1 = \phi - 1 \\ &= -\frac{1-\sqrt{5}}{2} = -\tau \\ \tau &= -x^{-1} = -\phi^{-1} \end{aligned}$$

This value for τ in terms of ϕ (they are in fact complex conjugates), can be substituted into the F_n expression to give:

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$$

Binet's formula is coded using floats and decimals in `fibsLib.py`, and as with the factorial approximations, we need to consider precision.

```
import math
from decTrig import *

PHI_F = (1 + math.sqrt(5)) / 2

def fibPhi(n): # Binet's formula
  return (PHI_F**n - \
      (1-PHI_F)**n)/math.sqrt(5)

def fibPhiD(n):
  return (PHI**D(n) - \
     (D(1)-PHI)**D(n))/D(5).sqrt()
```

```
>>> fibIter(13)
233
>>> fibPhi(13)
233.00000000000006
>>> fibPhiD(13)
Decimal('233.000000000000000000000000')
>>>
>>> fibIter(100)
354224848179261915075
>>> fibPhi(100)
3.542248481792631e+20
>>> fibPhiD(100)
Decimal('354224848179261915075.0000000')
```

Binet's formula doesn't involve series approximations like Stirling's formula, and so the only errors are those associated with floating point and decimal accuracy (15 digits and 28 digits respectively). These can be seen in the results produced by fibPhi() and fibPhiD().

One of the consequences of Binet's formula is that the ratio of two consecutive Fibonacci numbers tends to ϕ as n approaches infinity:

$$\lim_{x \to \infty} \frac{F_{n+1}}{F_n} = \phi$$

The $(-\phi)^{-n}$ in Binet progressively approaches 0 so the remaining ratio approaches $\frac{\phi^{n+1}}{\phi^n}$, or ϕ. This limit has nothing to do with the choice of base cases (i.e $F_0 = 0$ and $F_1 = 1$) but is due to the recurrence relation. Listing 1.13 (golden.py) emphasizes this by generating a Fibonacci series beginning with two random integers between 1 and 10000, and calculating the ratio of successive terms:

```
n1 = random.randint(1, 10000)
n2 = random.randint(1, 10000)
print(n1, n2)
sums = [n1, n2]

ratios = []
for i in range(20):
  sumCurr = sums[-1] + sums[-2]
  sums.append(sumCurr)
  ratio = sums[-1]/sums[-2]
  print(f"{i:2d} {sumCurr:10d}  {(ratio-PHI)/PHI: 15.6%}")
  ratios.append(ratio)
```

Listing 1.13. Generating Fibonacci ratios

No matter what values are chosen for n1 and n2, after 20 or so iterations, the ratio is equal to ϕ to at least six decimal places.

1.2.3 The Golden Spiral. A golden spiral is a logarithmic spiral whose growth factor is set to ϕ. It can be approximated using a Fibonacci spiral, which we'll construct in two different ways – using turtle graphics and through more conventional curve plotting. (If you're unfamiliar with Python's turtle module, please read Appendix G.)

A logarithmic spiral is distinguished from other types, such as the Archimedean spiral, by the way that the distances between its turnings increase in a geometric progression (an Archimedean spiral's turning distances remain constant). A natural way to implement this is by using a turtle to do the turning.

A golden spiral's ϕ growth factor is approximated by using the Fibonacci series (F_1, F_2, F_3, ..., F_n) since the ratio F_n/F_{n-1} approaches ϕ as n increases.

The crucial part of `fibSpiral.py` is:

```
fibs = fibs(n+1)[1:]   # drop F0 value
  :
for i in range(len(fibs)):
  dist = math.pi/180 * fibs[i] * scale
  for j in range(90):  # quarter circle
    t.forward(dist)
    t.left(1)
```

The `dist` value is used to draw a quarter circle. In effect we are approximating the growth steps of the spiral by a series of quarter turns centered inside squares with Fibonacci side lengths.

This is much more apparent in Fig. 1.8 – a box is drawn around each quarter turn and boxes' side lengths are labeled.

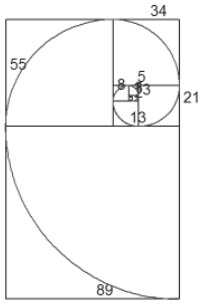


Figure 1.8. Fib squares and spiral

Although Fibonacci is a commonly utilized approximation, it isn't all that close to the actual golden spiral, as we'll see shortly.

The polar coordinate (r, θ) for a golden spiral is $r = a\phi^{2\theta/\pi}$. where θ is in radians, and a some constant. This can be compared to the polar equation for a general logarithmic spiral $r = ce^{b\theta}$, where b is the growth factor, and c some constant. They can be seen to be equivalent by changing the base of the power:

$$\begin{aligned} r &= a\phi^{2\theta/\pi} \\ \log r &= \log a + \frac{2\theta}{\pi} \cdot \log \phi \\ &= \log a + \theta b \\ r &= ae^{b\theta} \end{aligned}$$

Listing 1.14 (goldSpiral.py) doesn't use the Fibonacci series. Instead, it loads `PHI_F` from `fibsLib.py` to calculate the polar coordinates directly. These are converted to Cartesian form and plotted by Matplotlib.

```
NUM_PTS = 1000

def plotGolden(n):
  angles = linspace(0, n*math.pi, NUM_PTS)
  # Convert to (r,theta) to (x,y)
  xs = [PHI_F**(2*ang/math.pi) * math.cos(ang) for ang in angles]
  ys = [PHI_F**(2*ang/math.pi) * math.sin(ang) for ang in angles]

  plt.figure(figsize=(8, 8))
  plt.plot(xs, ys, color="blue", linewidth=2)
  plt.axis("equal")
```

```
  plt.axhline(0)
  plt.axvline(0)
  plt.title("Golden Spiral")
  plt.show()

n = float(input("n? "))
plotGolden(n)
```

Listing 1.14. Plotting the golden spiral

The result is in Fig. 1.9. The easiest-to-see differences between Figs. 1.8 and 1.9 are at the four compass points: the Fibonacci spiral is always a tangent to these directions whereas the golden spiral is pointing slightly outwards.

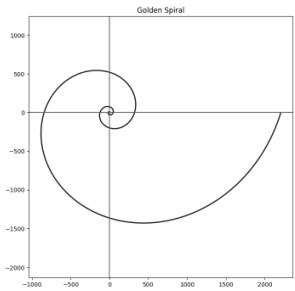


Figure 1.9. Golden spiral

1.2.4 Golden Spirals of 2D Points. Another popular way of drawing the golden spiral is by rendering multiple spirals as points filling the plane. We generate n polar coordinates, where $0 \leq i < n$:

$$r_i = (i/n)^{\frac{1}{2}} \quad \text{and} \quad \theta_i = 2\pi i\phi$$

The square-root of the radius has nothing to do with ϕ but is required to distribute the points uniformly over a circular area. If the square-root wasn't applied then the points would tend to congregate near the center.

Listing 1.15 (goldScatter2D.py) also draws a pale circle behind the points to make it easier to see their relative positions.

```
NUM_PTS = 1000

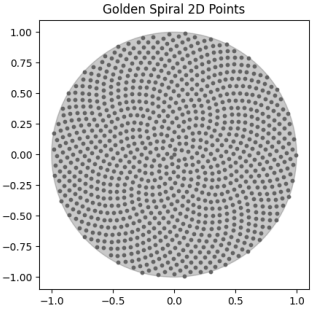
circle = plt.Circle((0, 0), 1, color="red", alpha=0.3)
plt.gca().add_patch(circle)

xs = [0]*NUM_PTS
ys = [0]*NUM_PTS
for i in range(NUM_PTS):
  r = (i/NUM_PTS)**0.5
  theta = 2*math.pi*(i*PHI_F)
  xs[i] = r*math.cos(theta)
  ys[i] = r*math.sin(theta)

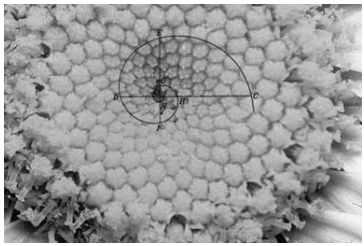
plt.scatter(xs, ys, s=10)
plt.axis('scaled')
plt.title("Golden Spiral 2D Points")
plt.show()
```

Listing 1.15. Plotting the golden spiral as 2D points

The result is in Fig. 1.10a.



(a) The golden spiral as 2D points



(b) Spirals in flower seeds

Figure 1.10. The golden spiral

Inevitably, images like Fig. 1.10a are compared to the position of seeds on flower heads (e.g. Fig. 1.10b), but it's quite unclear whether these spirals are specifically golden (i.e. based on ϕ) or just various logarithmic spirals.

It's also worth mentioning that even if the golden spiral is most commonly used in nature, that doesn't imply that there's a Fibonacci sequence behind it. Any series that satisfies the linear recurrence

$$G_{n+2} = G_{n+1} + G_n$$

such as Lucas numbers (`https://en.wikipedia.org/wiki/Lucas_number`), would work just as well.

The strongest argument in favor of nature's use of the golden spiral is that the points in Fig. 1.10a possess the densest packing for same-sized circles within a circular region.

1.2.5 Spirals of 3D Points. We don't have to stop at two dimensions; its entirely possible to map the golden spiral to higher dimensions. As a lead-in, let's start with randomly placed points on the surface of a sphere.

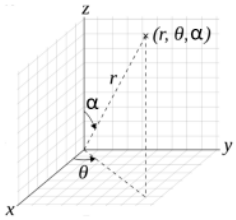


Figure 1.11. Spherical coordinates

1.2.5.1 Randomly Distributing Points on a Sphere. In spherical coordinates, r is the radius, $\theta \in [0, 2\pi]$ is the azimuthal angle, and $\alpha \in [0, \pi]$ is the polar angle, as in Fig. 1.11.

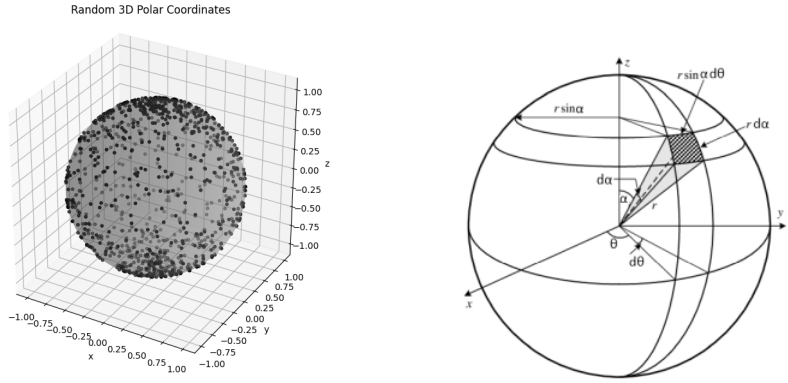
This is called the 'mathematics' convention (to distinguish it from the 'physics' convention). Also,

we've replaced the more usual polar angle variable, ϕ, with α to avoid confusion with the golden ratio.

This means that:

$$\begin{aligned} x &= r\sin(\alpha)\cos(\theta) \\ y &= r\sin(\alpha)\sin(\theta) \\ z &= r\cos(\alpha) \end{aligned}$$

A tempting way to obtain the points is to generate a uniform distribution of θ and α, then convert them to (x, y, z) points, as in randScatter3D.py



(a) Poorly randomized points on the surface of a sphere

(b) Defining dA on a sphere

Figure 1.12. Spherical Points

Fig. 1.12a shows that this approach does not lead to a good random distribution of the points. Instead, they tend to cluster around the poles ($\alpha = 0$ and $\alpha = \pi$) and are sparse around the equator ($\alpha = \pi$).

The reason for this is that the area dA of a small surface element is a square of dimension $rd\alpha$ by $r\sin(\alpha)\, d\theta$, as illustrated in Fig. 1.12b. This implies that close to the poles (when $\alpha = 0$ and $\alpha = \pi$), the surface area element determined by $d\theta$ and $d\alpha$ will get smaller since $\sin(\alpha) \to 0$. Thus, we should put less points near $\alpha = 0$ and $\alpha = \pi$ and more points near $\alpha = \pi/2$ to achieve better randomness.

Let v be a point on the unit sphere S. We want the probability density function (PDF) $f(v)$ to be constant for a uniform distribution.

The surface area of a unit sphere is 4π, which can be expressed as $\int\int_S dA = 4\pi$. Also, $f(v)\, dA$ is the probability of finding a point dA in an area about v on the sphere. Since $\int\int_S f(v)\, dA = 1$ then $f(v) = \frac{1}{4\pi}$.

However, we really want to represent points v using a parameterization of θ and α and so want the PDF in the form $f(\theta, \alpha)$. We can obtain it by equating:

$$f(v)\, dA = \frac{1}{4\pi}\, dA = f(\theta, \alpha)\, d\theta\, d\alpha,$$

We know that $dA = \sin(\alpha)\, d\alpha\, d\theta$ (assuming $r = 1$), so

$$\begin{aligned} f(\theta, \alpha)\, d\theta\, d\alpha &= \tfrac{1}{4\pi}\, dA \\ &= \tfrac{1}{4\pi}\, \sin(\alpha)\, d\alpha\, d\theta \\ f(\theta, \alpha) &= \tfrac{1}{4\pi}\, \sin(\alpha) \end{aligned}$$

Now separate this joint distribution to get the PDFs of θ and α:

$$f(\theta) = \int_0^{\pi} f(\theta, \alpha) d\alpha = \frac{1}{2\pi}$$
$$f(\alpha) = \int_0^{2\pi} f(\theta, \alpha) d\theta = \sin(\alpha)/2.$$

We see that θ is already uniform (so needs no changes), but $f(\alpha)$ scales with $\sin(\alpha)$. This means that we want more points around the equator, $\alpha = \pi/2$, which is where $\sin(\alpha)$ takes its maximum. In that case, how can we sample numbers α that follow the distribution $f(\alpha)$?

Inverse Transform Sampling allows us to sample a general PDF using a uniform random number. For this, we need the cumulative distribution function (CDF) of the random variable (α in our case).

The CDF of a random variable X is the function

$$F_X(x) = P(X \leq x)$$

The CDF can also be expressed as the integral of the PDF, $f(x)$:

$$F(x) = \int_{-\infty}^{x} f(x)\, dx.$$

This allows us to write the CDF of α as:

$$F(\alpha) = \int_0^{\alpha} f(\alpha)\, d\alpha = \int_0^{\alpha} \sin(\alpha)/2\, d\alpha = \frac{1}{2}(1 - \cos(\alpha))$$

Let U be the uniform random number in $[0, 1]$. Note that

$$Pr(U \leq F(\alpha)) = F(\alpha).$$

As F is invertible and monotone, we can preserve this inequality by writing:

$$Pr(F^{-1}(U) \leq \alpha) = F(\alpha).$$

This shows that $F(\alpha)$ is the CDF for the random variable $F^{-1}(U)$. Thus, $F^{-1}(U)$ follows the same distribution as α.

The algorithm for sampling the distribution using inverse transform sampling becomes:

- Generate a uniform random number U from the distribution $U[0,1]$. (We use Python's random.random() for this.)
- Compute $\alpha = F^{-1}(U)$ such that $F(\alpha) = U$.
- Use this α as a random number drawn from the distribution $f(\alpha)$.

In our case, $F(\alpha) = U = \frac{1}{2}(1 - \cos(\alpha))$, and so $\alpha = F^{-1}(U) = \arccos(1 - 2U)$.

This requires the change of one line in `randScatter3D.py` – the commenting out of the old assignment to alpha, replacing it with

```
alpha = math.acos(1-2*random.random())
```

The resulting globe is much more smoothly random (see Fig. 1.13).

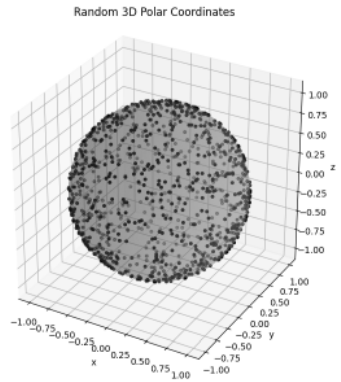


Figure 1.13. Uniformly randomized points on the surface of a sphere

1.2.5.2 Evenly Distributing Points on a Sphere. A much harder task is to *evenly* distribute points over a sphere's surface. Unfortunately, you can only do this if the points are the vertices of a regular solid. So you can exactly evenly space 4, 6, 8, 12, or 20 points, but if you want to position more than 20, then some will necessarily be closer to their neighbors than others. Of course, there are ways to approximately position the points, and the golden spiral offers both a simple, and very accurate, solution.

The implementation begins by evenly distributing n points inside a rectangle, where $0 \leq i < n$:

$$x_i = i\phi \quad \text{and} \quad y_i = i/n$$

Each (x_i, y_i) pair is mapped to the angles we used in randScatter3D.py:

$$\theta_i = \pi x_i \quad \text{and} \quad \alpha_i = \arccos(1 - 2y_i)$$

Listing 1.16 shows the part of goldScatter3D.py where the coordinates are generated. The resulting sphere is in Fig. 1.14.

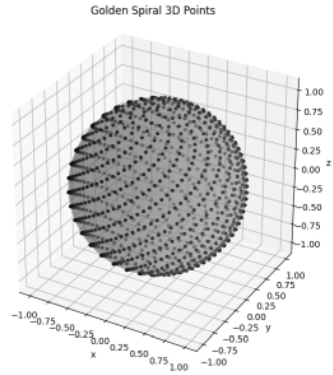


Figure 1.14. Golden spiral points on the surface of a sphere

```
xs = [0]*NUM_PTS
ys = [0]*NUM_PTS
zs = [0]*NUM_PTS
radius = 1
for i in range(NUM_PTS):
  theta = math.pi*(i*PHI_F)
  alpha = math.acos(1-2*(i/NUM_PTS))   # for uniformity
  xs[i] = radius*math.cos(theta)*math.sin(alpha)
  ys[i] = radius*math.sin(theta)*math.sin(alpha)
  zs[i] = radius*math.cos(alpha)
```

Listing 1.16. Coordinates for the golden spirals on the surface of a sphere

One way to measure the distance between points is to construct a line from the center to each one. We can measure the angle between two lines by taking their dot product.

E. B. Saff and A. B. J. Kuijlaars, 'Distributing many points on a sphere', *The Mathematical Intelligencer*, Vol. 19, No. 1, pp. 5-11, 1997 is a great paper on this topic. Another is 'Measurement of areas on a sphere using Fibonacci and latitude-longitude lattices' by Álvaro González, available at `https://arxiv.org/pdf/0912.4540.pdf`. A very nice online resource is Martin Roberts's blog post at `https://extremelearning.com.au/how-to-evenly-distribute-points-on-a-sphere-more-effectively-than-the-canonical-fibonacci-lattice/`.

1.2.6 Generalizing the Fibonacci Series. There are numerous ways that the Fibonacci series can be modified (`https://en.wikipedia.org/wiki/Generalizations_of_Fibonacci_numbers`). The Lucas sequence is an important one, and appears in one of the exercises at the end of this part. Another generalization is to higher order sequences, where each element is the sum of the previous n elements. The *tribonacci* numbers starts with *three* predefined terms and each subsequent term is the sum of the preceding *three*. *Tetranacci* numbers start with four terms, and each following term is the sum of the preceding four. Of course, Fibonacci numbers follow this pattern, with order 2.

These sequences start with $(n-1)$ 0's and a single 1. So the first few tribonacci numbers are:

```
0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, ...
```

The tetranacci numbers begin as:

```
0, 0, 0, 1, 1, 2, 4, 8, 15, 29, 56, 108, 208, 401 ...
```

Code for generating these sequences is in higherFibs.py. It employs the user's supplied n value to create a list filled with $(n-1)$ 0's and a 1. A new value is calculated by summing the list, then the list's data is shifted one element to the left, causing the oldest value to be dropped. The space opened up at the end is used to store the new value, and the program repeats.

1.3 The Tower of Hanoi

In 1883 the French mathematician, Édouard Lucas, unveiled 'The Tower of Hanoi' (La Tour D'Hanoï) puzzle. There are eight wooden disks with holes in their centers, stacked in order of decreasing size on one pole in a row of three (see Fig. 1.15).

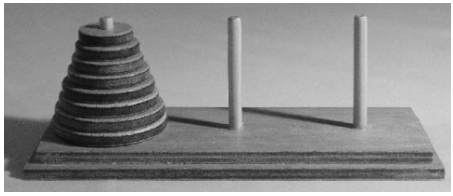


Figure 1.15. The Tower of Hanoi

The objective is to move the entire stack to one of the other poles, while following three rules:

(1) Only one disk may be moved at a time.

(2) Each move consists of taking the top disk from one of the stacks and placing it on top of another stack or on an empty pole.

(3) No disk may be placed on top of a smaller disk.

Fig. 1.16 shows the seven moves required to move three disks to another pole.

Lucas' instructions claim that the puzzle was based on an Indian legend:

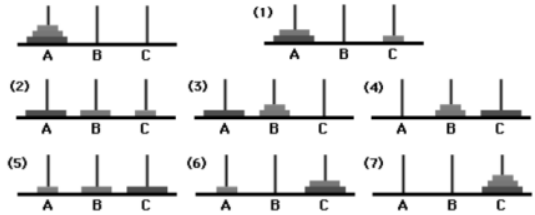


Figure 1.16. Moving a stack of three disks

> On the steps of the altar in the temple of Benares, for many, many years Brahmins have been moving a tower of sixty-four golden disks from one pole to another; one by one, never placing a larger on top of a smaller. When all the disks have been transferred the Tower and the Brahmins will fall, and it will be the end of the world.

Lucas offered a prize of ten thousand francs (about $200,000 today) to anyone who could move a tower of 64 disks by hand while following the game rules. What he didn't reveal was that the minimal number of required moves is $2^n - 1$, where n is the number of disks. So, if the eager contestant was able to move disks at a rate of one per second, it would take them $2^{64} - 1$ seconds or roughly 585 billion years to finish, which is some time after the end of the universe. Indeed, the Earth will have become uninhabitable due to the Sun's decay after a mere billion years. This constrains the problem to around a maximum of 55 disks.

A program for solving the puzzle arises naturally by expressing the solution recursively. The poles are labeled A, B, and C, with A the home of the stack of disks at the start, and the aim being to move them to pole C.

The base case, when there's just one disk, involves moving that disk directly from pole A to C.

The recursive case, involving the transfer of n disks from A to C, is divided into three stages:

(1) Move the top $n - 1$ disks from A to B. Note that this is a recursive version of the original goal with a smaller stack of disks, and using pole B as the destination. Pole C will be utilized as an auxiliary.

(2) Move the bottom disk from A to C.

(3) Move the $n - 1$ disks from B to C (using pole A as the auxiliary this time). This is another recursive task.

A representation of these three stages for $n = 4$ disks is shown in Fig. 1.17.

A key insight is that the recursive stages 1 and 3 involving $n - 1$ disks are possible because the largest disk is at the bottom of a pole, and so cannot limit the choice of moves for the other disks.

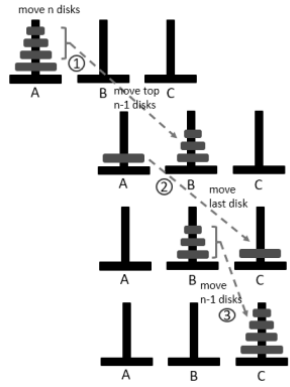


Figure 1.17. Moving n disks in three stages

The resulting code is in Listing 1.17 (hanoi.py).

```
def moves(n, src, dest, aux):
  if n == 1:
    print("Disk 1 :", src, "-->", dest)
  else:
    moves(n-1, src, aux, dest)
    print("Disk", n, ":", src, "-->", dest)
    moves(n-1, aux, dest, src)


n = int(input("n? "))
print("Moving", n, "disks from A to C")
moves(n, "A", "C", "B")
```

Listing 1.17. Recursive Tower of Hanoi

The following output replicates the steps shown in Fig. 1.16.

```
> python hanoi.py
n=? 3
Moving 3 disks from A to C
Disk 1 : A --> C
Disk 2 : A --> B
Disk 1 : C --> B
Disk 3 : A --> C
Disk 1 : B --> A
Disk 2 : B --> C
Disk 1 : A --> C
```

For larger stacks it's useful to print a textual representation of the poles state, which is done by hanoiShow.py. It stores the poles data in a Python dictionary, with the pole name as the key attached to a list of disks currently on that pole.

The output for the 3-disk puzzle is now:

```
> python hanoiShow.py
n? 3
Moving 3 disks from A to C
                    A         B         C
                    [ 3 2 1   [         [
Disk 1: A --> C   : [ 3 2     [         [ 1
Disk 2: A --> B   : [ 3       [ 2       [ 1
Disk 1: C --> B   : [ 3       [ 2 1     [
Disk 3: A --> C   : [         [ 2 1     [ 3
Disk 1: B --> A   : [ 1       [ 2       [ 3
Disk 2: B --> C   : [ 1       [         [ 3 2
Disk 1: A --> C   : [         [         [ 3 2 1
```

1.3.1 Calculating the Running Time. A recurrence relation can be used to calculate the big-Oh running time. T_n is the time to move n disks from the starting pole to the destination, and is defined as:

$$\begin{aligned} T_n &= b + 2T_{n-1}, \quad n > 1 \\ &= a, \quad n = 1 \end{aligned}$$

where a and b are constant times for a single move. The $b + 2T_{n-1}$ sum is the cost for the three stages. The recursive stages 1 and 3 each take time T_{n-1} since they are moving $n - 1$ disks. This relation can be solved by substitution:

$$\begin{aligned} T_n &= b + 2T_{n-1} \\ &= b + 2(b + 2T_{n-2}) \\ &= b + 2b + 2^2 T_{n-2} \\ &= b + 2b + 2^2(b + 2T_{n-3}) \\ &= (1 + 2 + 2^2)b + 2^3 T_{n-3} \\ &\vdots \\ &= (1 + 2 + 2^2 + \cdots 2^{n-2})b + 2^{n-1} T_1 \\ &= (1 + 2 + 2^2 + \cdots 2^{n-2})b + 2^{n-1} a \\ &= (2^{n-1} - 1)b + 2^{n-1} a \\ &= 2^{n-1}(a + b) - b \end{aligned}$$

Since a and b are constants, T_n is $O(2^{n-1}) + O(1)$, which can be simplified to T_n is $O(2^n)$, or exponential running time.

1.3.2 An Iterative Version. In "The Tower of Hanoi for Humans" [**Sto18**], Paul K. Stockmeyer describes three iterative algorithms, which he classifies as 'easy for humans' due to the rules for disk moves only utilizing a bounded amount of memory, augmented by configuration information such as disk size or color.

The third method is by Scorer, Grundy, and Smith from their paper 'Some Binary Games', *Math. Gaz.*, 28 (1944), no. 280 (July) pp.96-103. For $X, Y \in$

$\{A, B, C\}$, $(X \leftrightarrow Y)$ denotes a disk move involving poles X and Y in either direction. There is one rule:

- Move disks cyclically in the order:

$$(A \leftrightarrow B), \quad (A \leftrightarrow C), \quad (B \leftrightarrow C), ...$$

(The relative sizes of the disks on the poles determine the direction of the move.)

Note: if the number of disks is odd, then the cyclic order changes to

$$(A \leftrightarrow C), \quad (A \leftrightarrow B), \quad (B \leftrightarrow C), ...$$

Essentially, this technique first makes the move that avoids pole C, then the move that avoids pole B, then the move that avoids pole A, and so on, cycling the excluded pole in the direction $C \to B \to A \to \cdots$.

The Python implementation of this method is in `hanoiSGS.py`, and reuses the poles dictionary from `hanoiShow.py`.

The output when four disks are moved:

```
> python hanoiSGS.py
n? 4
                   A          B          C
                   [ 4 3 2 1  [          [
Disk 1: A --> B  : [ 4 3 2    [ 1        [
Disk 2: A --> C  : [ 4 3      [ 1        [ 2
Disk 1: B --> C  : [ 4 3      [          [ 2 1
Disk 3: A --> B  : [ 4        [ 3        [ 2 1
Disk 1: C --> A  : [ 4 1      [ 3        [ 2
Disk 2: C --> B  : [ 4 1      [ 3 2      [
Disk 1: A --> B  : [ 4        [ 3 2 1    [
Disk 4: A --> C  : [          [ 3 2 1    [ 4
Disk 1: B --> C  : [          [ 3 2      [ 4 1
Disk 2: B --> A  : [ 2        [ 3        [ 4 1
Disk 1: C --> A  : [ 2 1      [ 3        [ 4
Disk 3: B --> C  : [ 2 1      [          [ 4 3
Disk 1: A --> B  : [ 2        [ 1        [ 4 3
Disk 2: A --> C  : [          [ 1        [ 4 3 2
Disk 1: B --> C  : [          [          [ 4 3 2 1
```

1.4 Derangements

A *derangement* is a permutation of objects that leaves no object in its original position. For example, 21453 is a derangement, but 21543 is not because 4 has not moved.

The problem of counting derangements was first considered by Pierre Raymond de Montmort in his *Essay d'analyse sur les jeux de hazard* in 1708.

The number of derangements of n objects (OEIS sequence A000166 (`https://oeis.org/A000166`)) for small n is given in Table 1.2.

n	0	1	2	3	4	5	6	7	8	9	10
D_n	1	0	1	2	9	44	265	1,854	14,833	133,496	1,334,961

Table 1.2. The number of derangements of n objects

For instance, $D_3 = 2$, because the only derangements of a sequence of length 3, such as 123, are 231 and 312.

Another version of the problem arises when we ask for the number of ways n letters, each addressed to a different person, can be placed in n pre-addressed envelopes so that no letter is assigned the correctly addressed envelope.

Another popular variant is the *hat-check problem*, in which we count the number of ways in which n hats (let's call them h_1 through h_n) can be returned to n people (P_1 through P_n) such that no hat makes it back to its owner. In other words, we want to calculate D_n. Each person may receive any one of the $n-1$ hats that are not their own. Call the hat that the person P_1 receives h_i and then consider h_i's owner. P_i may receive either P_1's hat, h_1, or some other, so the problem splits into two cases:

(1) P_i does *not* get hat h_1, which means this case is equivalent to solving the problem with $n-1$ people and $n-1$ hats (P_1 received h_i, so the number of hats and people is reduced by 1). We can write this as D_{n-1}.

(2) P_i does get h_1. In this case the problem reduces to $n-2$ people and $n-2$ hats. (P_1 and P_i received h_i and h_1, so the number of hats and people reduces by 2). We can write this as D_{n-2}.

To summarize, the number of actions possible for P_i is the sum of the counts for the two cases, which is $D_{n-1} + D_{n-2}$. However, this must be multiplied by $(n-1)$ because any one of the $(n-1)$ people could have been chosen to be P_i.

Thus, the number of derangements D_n is

$$D_n = (n-1)(D_{n-1} + D_{n-2}) \text{ for } n \geq 2, \tag{1.1}$$

where $D_0 = 1$ and $D_1 = 0$.

There's a direct translation to Python shown in Listing 1.18 (derange.py).

```
def countDer(n):
  if n == 0:
    return 1
  elif n == 1:
    return 0
  else:
    return (n-1) * \
       (countDer(n-1) + countDer(n-2))
```

Listing 1.18. A recursive derangement counter

This code works correctly, but becomes prohibitively slow for larger values of n. More efficient encodings are needed.

1.4.1 Derivation of a Closed Form. Subtracting nD_{n-1} from both sides of Equ. (1.1) yields

$$D_n - nD_{n-1} = -(D_{n-1} - (n-1)D_{n-2})$$

Let $A_n = D_n - nD_{n-1}$, and the recurrence relation becomes

$$A_n = -A_{n-1}.$$

with $A_0 = 1$.

Since $A_0 = 1$, we get $A_1 = -A_0 = -1, A_2 = -A_1 = 1, A_3 = -A_2 = -1$, and so on. In general, $A_n = (-1)^n$.

Now we can write

$$D_n - nD_{n-1} = A_n, = (-1)^n. \tag{1.2}$$

This can be expressed as the recurrence

$$D_n = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot D_{n-1} + (-1)^n & \text{if } n > 0. \end{cases}$$

Although this is still a recursive definition, the fact that it only uses a single recursive call rather than the two calls in the original definition, suggests that it can be implemented more efficiently. Also, in a system supporting tail-recursive optimization (which is not part of Python), this code would be compiled to use iteration. In any case, this form makes it easier to carry out a manual translation (see the next subsection). The single-recursive function is shown in Listing 1.19 (derange.py).

```
def countDer2(n):
  if n == 0:
    return 1
  else:
    return n*countDer2(n-1) + (-1)**n
```

Listing 1.19. A single-recursive derangement counter

This version is not much slower than the iterative code (as we'll see below), but it may still crash if it exceeds Python's recursion depth limit, and also uses a lot of memory for stack frames.

Back to the math. Divide both sides of Equ. (1.2) by $n!$ to obtain

$$\frac{D_n}{n!} - \frac{D_{n-1}}{(n-1)!} = \frac{(-1)^n}{n!}$$

Let $B_n = \frac{D_n}{n!}$, so we have

$$B_n = B_{n-1} + \frac{(-1)^n}{n!}$$

with $B_0 = 1$. Solving this recurrence, we get

$$\frac{D_n}{n!} = B_n = \sum_{k=0}^{n} \frac{(-1)^k}{k!}$$

Multiplying both sides by $n!$ yields the derangement formula:

$$D_n = n! \sum_{k=0}^{n} \frac{(-1)^k}{k!} \tag{1.3}$$

The sum is complex, but we can utilize the identity:

$$\lim_{x\to\infty} \sum_{k=0}^{n} \frac{x^k}{k!} = e^x$$

So when $x = -1$, we have

$$D_n \approx n! \lim_{x\to\infty} e^x = n!\, e^{-1}$$

More precisely:

$$D_n = \left[\frac{n!}{e}\right] = \left\lfloor \frac{n!}{e} + \frac{1}{2} \right\rfloor \text{ for } n \geq 1$$

where [x] is the nearest integer function and $\lfloor x \rfloor$ is the floor function.

We'll implement this with Python's factorial() and math.e, resulting in the one-liner in Listing 1.20 (derange.py):

```
def countDerApprox(n):
  return int(math.factorial(n)/math.e + 0.5)
```

Listing 1.20. An approximate derangement counter

1.4.1.1 Using Iteration. The original recursive definition is quite similar to our Fibonacci code from Listing 1.2, and so can utilize the same sort of iterative conversion. The idea is to employ two additional variables to hold the current and previous derangement values, which are used to calculate the next count. The code is in Listing 1.21 (derange.py).

```
def countDerIter(n):
  if n == 0:
    return 1
  elif n == 1:
    return 0
  prev = 0
  dr = 1
  for i in range(3, n+1):
    curr = (i-1)*(prev+dr)
    prev = dr
    dr = curr
  return dr
```

Listing 1.21. An iterative derangement counter

The translation of the single-recursive version is slightly complicated by the need to add $(-1)^n$, For efficiency's sake, this should *not* be implemented using powers; instead an additional variable, `mult`, maintains the ongoing power, which is multiplied by -1 on each iteration (see Listing 1.22; derange.py).

```
def countDerIter2(n):
  if n == 0:
    return 1
  mult = -1
  dr = 1
  for i in range(1, n+1):
    dr = i*dr + mult
    mult *= -1
  return dr
```

Listing 1.22. A faster iterative derangement counter

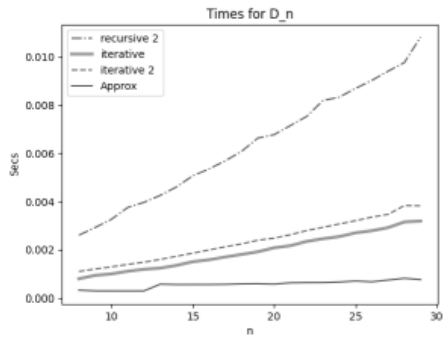


Figure 1.18. Running times for derangement

We can measure and plot the times for the five derangement functions using the timeit module that we saw back in Listing 1.1. Actually, the original recursive function (1.18) is so slow that we commented out its call, resulting in the timing plots shown in Fig. 1.18.

The timings are based on the average of 1000 calls for calculating D_8 to D_{30}.

The single-recursive function (Fig. 1.19) is the slowest but with very similar running times to the others in absolute terms. There's not much to choose between the two iterative versions, but the approximation function (Listing 1.20) is the clear winner. Incidentally, the approximation is fairly accurate, as demonstrated below:

```
>>> from derange import *
>>> countDerApprox(3)
2
>>> countDerIter(3)
2
>>> countDerApprox(25)
57062552826334675195658 24
>>> countDerIter(25)
57062552826334667623572 24
```

1.5 The Bisection Method

The following sequence was first described by Édouard Lucas (who later devised 'The Tower of Hanoi' puzzle in Sec. 1.3) and his ideas were developed further by Raoul Perrin in 1899:

$$v(0) = 3, \quad v(1) = 0, \quad v(2) = 2$$
$$v(n) = v(n-2) + v(n-3) \text{ for } n \geq 3.$$

Two recursive functions (Listings 1.23 (linRec.py) and 1.24 (linRec1.py)) print the ratio $q(n) = v(n)/v(n-1)$. The first stops as soon as $v(n-1) > 10^8$ holds, the second prints 50 terms of the sequence q(n), starting with n = 3. We'll return to Perrin numbers (`https://en.wikipedia.org/wiki/Perrin_number`) again and again in future sections.

```
def v(n0, n1, n2):
  print(f"{n1/n0:.10f}")
  if n0 <= 1E+08:
    v(n1, n2, n0+n1)  # shift left

v(2, 3, 2)
```

Listing 1.23. Perrin numbers to 1e8

```
def v(n0, n1, n2, col):
  if col > 0:
    print(f"{n1/n0:14.10f}", end ='')
    if (col-1)%5 == 0:
      print()
    v(n1, n2, n0+n1, col-1)  # shift left

v(2, 3, 2, 50)
```

Listing 1.24. 50 Perrin numbers

Output from Listing 1.24:

```
 1.5000000000  0.6666666667  2.5000000000  1.0000000000
 1.4000000000  1.4285714286  1.2000000000  1.4166666667
 1.2941176471  1.3181818182  1.3448275862  1.3076923077
```

```
1.3333333333  1.3235294118  1.3222222222  1.3277310924
1.3227848101  1.3253588517  1.3249097473  1.3242506812
1.3251028807  1.3245341615  1.3247362251  1.3247787611
1.3246492986  1.3247604639  1.3247049867  1.3247126437
1.3247288503  1.3247093499  1.3247218789  1.3247177382
1.3247164894  1.3247195193  1.3247170266  1.3247182160
1.3247180989  1.3247177043  1.3247181492  1.3247178740
1.3247179579  1.3247179924  1.3247179218  1.3247179776
1.3247179522  1.3247179536  1.3247179631  1.3247179530
1.3247179590  1.3247179573
```

If $q(n)$ converges, the limit is a root of the equation:

$$x^3 - x - 1 = 0 \tag{1.4}$$

This can be obtained by rearranging the equation,

$$v(n) = v(n-2) + v(n-3) \Rightarrow \frac{v(n)}{v(n-1)} = \frac{v(n-2)}{v(n-1)} + \frac{v(n-3)}{v(n-2)} * \frac{v(n-2)}{v(n-1)}$$

Let x be the limit of $q(n)$. Then:

$$x = \frac{1}{x} + \frac{1}{x^2}, \quad \text{or} \quad x^3 = x + 1$$

So $q(n)$ converges to a root of Equ. (1.4). For $f(x) = x^3 - x - 1$ we have $f(1) = -1$, $f(2) = 5$, so there must be a root between 1 and 2. We can visually check this in Fig. 1.19, which is generated by `plotEq.py` after supplying it with the equation of the curve, and start and end x values:

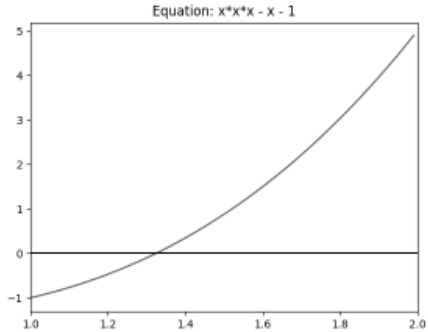


Figure 1.19. Plot of $x^3 - x - 1$ between $1 \leq x \leq 2$

```
> python plotEq.py
x equation? x*x*x - x - 1
xStart xEnd=? 1 2
```

The equation string is converted to a function by Python's eval().

We'll pass the interval [1, 2] to the recursive bisection method in Listing 1.25 (bis.py).

```
def perrin(x):  # root between 1 and 2
  return x*x*x - x - 1

def bis(fn, a, b):
  if sign(fn(a)) == sign(fn(b)):
    print(f"{a} and {b} do not span a root")
    return None, 0
  else:
    return biRec(fn, a, b, 0)

def biRec(fn, a, b, iters):
  mid = (a + b)/2
  if iters > MAX_ITERS:
    print(f"Iterations exceeded", MAX_ITERS)
    return mid
  if abs(fn(mid)) < EPS:
    return mid
  elif sign(fn(a)) == sign(fn(mid)):
    # mid is an improvement on a
    return biRec(fn, mid, b, iters+1)
  elif sign(fn(b)) == sign(fn(mid)):
    # mid is an improvement on b
    return biRec(fn, a, mid, iters+1)

if __name__ == "__main__":
  a,b = map(float, input("a b? ").split())
  root = bis(perrin, a, b)
  # root = bis(ef, a, b)
  print(f"Root = {root:.10f}")
```

Listing 1.25. Recursive bisection

```
> python bis.py
a b? 1 2
Root = 1.3247179571
```

The bisection method works because a continuous function with values of opposite signs in an interval must have a root within that interval. biRec() narrows in on the root by calculating the midpoint $mid = \frac{a+b}{2}$. This gives us two smaller intervals: (a, mid) and (mid, b). The sign of fn(mid) compared to fn(a) and fn(b) determines which of these contains the root, and biRec() is recursively called on that sub-interval.

Each call reduces the interval by half, making bisection a *linear convergence* technique that's guaranteed to find a root (if it was supplied with a starting interval that brackets the root).

Slow convergence is the main drawback compared to other root-finding approaches, so an iteration counter is used to check if the function is taking too long. The other convergence problem is if there are multiple roots within the interval, which may cause the method to move away from the root we intended to find, and converge on another one.

Since we're dealing with floats, we stop the recursion when fn(mid) is less than EPS rather than 0 at the root, which is unlikely to occur.

It's fairly simple to rewrite bis() to be iterative since the recursive calls in biRec() are at the end of the function. This *tail recursion* is translated into a loop which 'jumps' back to the top of the function rather than recursing. Exercise 2 makes use of this approach.

Bisection is one of the big three root-finding techniques, the other two being Newton-Raphson and the secant methods. Newton-Raphson is great for speed (it can converge quadratically), but requires the function's derivative, and does not guarantee that the method will converge. The secant method gets around Newton-Raphson's derivative issue by estimating it using a secant line (see Fig. 1.20), but has the same convergence problem.

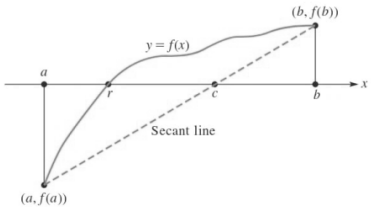


Figure 1.20. Secant method

1.6 Gambling Boldly

My current fortune is $x < 1$ (perhaps 1 stands for \$1 million), and I decide to start gambling *boldly* with the aim of increasing x to 1.

If $0 < x \leq 0.5$ then I stake x, all I have. If I win, my fortune will jump to $2x$, although a loss will wipe me out. If $0.5 < x < 1$, then I bet $1 - x$. If I win, then my fortune increases to $x + (1 - x) = 1$, and I retire happy. Also, if I lose, I still have $x - (1 - x) = 2x - 1$ left.

Suppose in one round my chance of winning is p, my chance of losing q, with $p + q = 1$. Let $f(x)$ be the probability of eventual success under these bold gambles starting with x. Then we have:

$$f(0) = 0, f(1) = 1, \quad f(x) = \begin{cases} p \cdot f(2x) & 0 < x \leq \frac{1}{2}; \\ p + q \cdot f(2x - 1) & \frac{1}{2} < x < 1. \end{cases} \tag{1.5}$$

The equation is nonlinear, which can be expressed separately as a function $d()$ defined on $[0,1]$:

$$d(x) = 2x - \lfloor x \rfloor = \begin{cases} 2x & : \ x \in \left[0, \frac{1}{2}\right) \\ 2x-1 & : \ x \in \left[\frac{1}{2}, 1\right) \end{cases}$$

d() is sometimes called the *doubling function*.

Another way to think about $f(x)$ is to treat x as a binary number. The function left-shifts the number by one bit then drops the integer part if the result is > 1.

Our program based on Equ. (1.5) is in Listing 1.26 (bold.py). The function is plotted for several values of p (see Fig. 1.21), with x ranging between 0 and 1.

```
''
def f(x,p):
  if x <= 0:
    return 0
  elif x >= 1:
    return 1
  elif x <= 0.5:
    return p*f(2*x,p)
  else:
    return p + (1-p)*f(2*x-1,p)
```

Listing 1.26. Gambling boldly by doubling

It can be shown that gambling boldly is optimal for $p < 0.5$ (i.e. in an unfair game). For $p = 0.5$ the gambling strategy has no influence on $f(x)$. For $p > 0.5$ bold gambles are bad; you should play as timidly as the rules of the game permit.

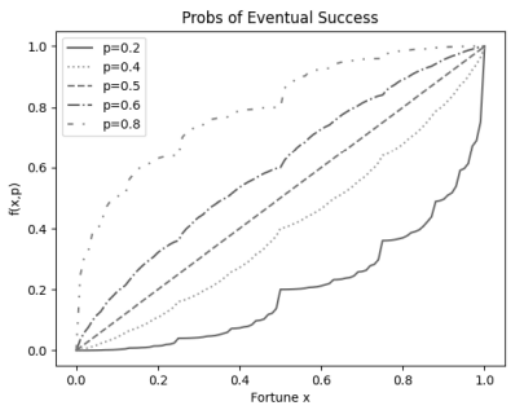


Figure 1.21. Probabilities of success for bold gambles for various p's

The mathematical analysis is explained in detail in the classic text by Lester E. Dubins and Leonard J. Savage [**DS76**]. Less technical sources are P. Billingsley: "The Singular Function of Bold Play", *Am. Scientist*, 71(1983), 392-397 and Rényi [**RÉ87**]. An online explanation begins at `https://stats.libretexts.org/Bookshelves/Probability_Theory/Probabil`

ity_Mathematical_Statistics_and_Stochastic_Processes_(Siegrist)/13%3A_Games_of_Chance/13.08%3A_The_Red_and_Black_Game. It is often called a *red and black* game, after the color bets in roulette.

1.7 The Josephus Problem

During the Jewish rebellion against Rome (A.D. 70), forty men were trapped in a cave, and to avoid a life of slavery they agreed upon mutual destruction. They would stand in a circle numbering themselves from 1 to 40. Every seventh person was then to be killed until only one was left, who would then commit suicide. We know the story from the only survivor, the historian Flavius Josephus, who obviously didn't carry out that last step.

This story is the source for the **Josephus problem**: n people are arranged in a circle, numbered 1 to n. Every k-th person is removed, with the circle closing up after each removal. What is the number $f(n)$ of the last survivor? What's the number of the s-th removed person?

The problem is fairly simple for $k = 2$ since we can express $f(2n)$ and $f(2n+1)$ in terms of $f(n)$. In Fig. 1.22a, with $2n$ people around the circle, we eliminate numbers $2, 4, ..., 2n$, and are left with $1, 3, ..., 2n-1$, which are then renumbered as 1 to n.

In Fig. 1.22b, with 2n+1 people, we eliminate numbers $2, 4, ..., 2n, 1$, and are left with $3, 5, ..., 2n+1$ who are renumbered to $1, 2, ..., n$.

Since $f(n)$ denotes the last survivor in the inner circle, we see that his original number (on the outer circle) is:

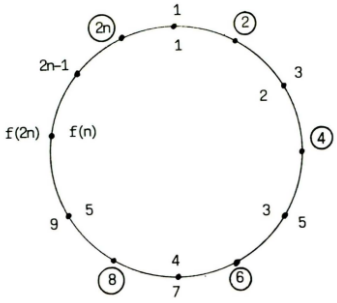
$$f(2n) = 2f(n) - 1 \quad \text{or} \quad f(2n+1) = 2f(n) + 1.$$

In addition we have $f(1) = 1$.

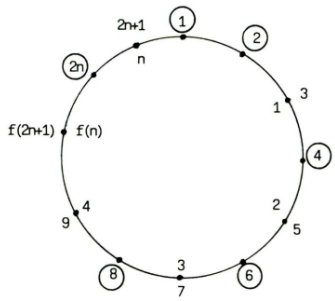
These three $f()$ equations translate into the recursive function in Listing 1.27 (jos.py). The use of // ensures that any decimal part of the division is omitted.

```
def f(n):
  if n < 3:
    return 1
  elif (n % 2 == 1): # odd
    return 2*f(n//2)+1
  else:
    return 2*f(n//2)-1

if __name__ == "__main__":
  print("Josephus Problem for n=2..40 and k = 2")
  print(" n: position of survivor at end")
  print("         ", end ='')
  for i in range(2, 41):
    print(f"{i:2d}: {f(i):2d}; ", end = ' ')
    if i%5 == 0:
```



(a) Removals using $f(2n) = 2f(n) - 1$



(b) Removals using $f(2n+1) = 2f(n) + 1$

Figure 1.22. Josephus circles

```
    print()
print()
```

Listing 1.27. Recursive Josephus

Fig. 1.23 suggests that the recursion tree depth is $O(\log_2 n)$ since each f() call roughly halves the number of people left until a single person remains.

The wikipedia page for the problem (`https://en.wikipedia.org/wiki/Josephus_problem`) includes a useful table for various group sizes, n, and step sizes, k. Looking at the values in the $k = 2$ column confirms that the following output is correct:

```
> python jos.py
Josephus Problem for n=2..40 and k = 2
 n: position of survivor at end
          2:  1;   3:  3;   4:  1;   5:  3;
 6:  5;   7:  7;   8:  1;   9:  3;  10:  5;
11:  7;  12:  9;  13: 11;  14: 13;  15: 15;
16:  1;  17:  3;  18:  5;  19:  7;  20:  9;
21: 11;  22: 13;  23: 15;  24: 17;  25: 19;
26: 21;  27: 23;  28: 25;  29: 27;  30: 29;
31: 31;  32:  1;  33:  3;  34:  5;  35:  7;
36:  9;  37: 11;  38: 13;  39: 15;  40: 17;
```

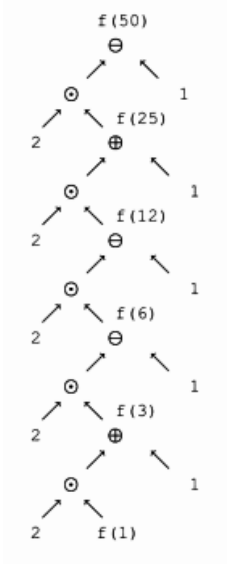


Figure 1.23. Execution of f(50)

Note that the survivor is the first person in the circle (i.e. $f(n) = 1$) whenever $n = 2^m$.

For $k = 2$, there's a simple expression for the number $f(n)$ of the survivor. For arbitrary n, let m be the largest

integer such that $2^m \leq n$. We can then write $n = 2^m + (n-2^m)$. Now remove the people numbered $2, 4, 6, ..., 2(n-2^m)$, leaving 2^m people in the circle. By the above result, the first one of these 2^m people will survive. The place number of this person is $2(n-2^m)+1$. Hence, for $k = 2$, the number of the last survivor is:

$$f(n) = 2(n-2^m)+1,$$

where 2^m is the largest power of $2 \leq n$. For example, for $n = 1988 = 1024 + 964$, then $f(n) = 2 * 964 + 1 = 1929$, which we can confirm with Python's IDE:

```
>>> from jos import *
>>> f(1988)
1929
```

If n is encoded in binary, then $f(n)$ can be computed by transferring the first digit of n to the end:

$$n = 1b_1b_2...b_m = 2^m + (n-2^m) \Rightarrow b_1b_2...b_m1 = 2(n-2^m)+1 = f(n).$$

An excellent, detailed discussion of the Josephus problem can be found in section 1.3 of Graham [**GKP94**], while the most extensive account is by Ahrens [**Ahr01**]. The surprising depth of the Josephus permutation is treated in Herstein and Kaplansky [**HK78**].

1.7.0.1 Finding the s-th Eliminated Person. For arbitrary k, the number x of the s-th eliminated person can be determined by means of Listing 1.28 (joseph1.py), which comes from problem 1.3.3-33 in Knuth, Vol. 1 [**Knu97**].

```
n,k,s = map(int, input("n k s=? ").split())
x = k*s
while x > n:
  x = int((k*(x-n)-1)/(k-1))
print(f"The {s}th eliminated person is labeled #{x}")
```

Listing 1.28. Calculating elimination position

For $k = 2$, $k/(k-1) = 2$, and the statement in the while-loop becomes $x = 2 * (x-n) - 1$. An example (which is simple enough to check manually):

```
> python joseph1.py
n k s=? 50 2 5
The 5th eliminated person is labeled #10
```

Consider a person P whose initial number in the Josephus circle is x_1, and whose number after the i-th round of eliminations is x_i. Assume the original circle is of size n, and the elimination step is k. We know x_i and want to determine x_1.

If P wasn't eliminated during the i-th cycle, then:

$$x_{i+1} = x_i + n - \left\lfloor \frac{x_i}{k} \right\rfloor. \tag{1.6}$$

The number of people eliminated during the x_i round was $\lfloor x_i/k \rfloor$, and there remains $n - \lfloor x_i/k \rfloor$ lucky survivors. After counting off these people we get back to P.

Now we form the sequence $x_1, x_2, ...$, until we reach an x_j which is divisible by k; that person will become the (x_j/k)-th eliminated. Thus the last number of the s-th eliminated person is ks. We must solve Equ. (1.6) for x_i, so we can work backwards to find his original number x_1. Now:

$$\frac{x_i}{k} = \left\lfloor \frac{x_i}{k} \right\rfloor + e, \quad \text{with } \frac{1}{k} \le e \le \frac{k-1}{k}; \tag{1.7}$$

$e \neq 0$, else x_i would be a multiple of k, and the person would have already been eliminated in the i-th round. From Equs. (1.6) and (1.7) we get:

$$x_i = \frac{(x_{i+1} - n)k}{k-1} - \frac{k}{k-1}e = \frac{(x_{i+1} - n)k - 1}{k-1} - \frac{k}{k-1}(e - \frac{1}{k})$$

By Equ. (1.7), the term subtracted on the right is between 0 and $\frac{k-2}{k-1}$. Since x_i is an integer, it is the larger integer part of the first term:

$$x_i = \left\lfloor \frac{(x_{i+1} - n)k - 1}{k-1} \right\rfloor = x_{i+1} - n + \left\lfloor \frac{x_{i+1} - n - 1}{k-1} \right\rfloor. \tag{1.8}$$

To find the original number of the s-th eliminated person, set $x_{i+1} = ks$ and apply Equ. (1.8) repeatedly until we get a number $\le n$; this is the number we are looking for.

The reader should try out this procedure by forming the sequence $x_1, x_2, ...$ and obtain the inverse sequence by means of Equ. (1.8). Suggestion: use $n = 18, k = 3, x_1 = 7$. Listing 1.28 reports:

```
> python joseph1.py
n k s=? 18 3 7
The 7th eliminated person is labeled #4
```

1.7.0.2 Josephus Iteratively. The cyclic nature of the Josephus problem means that iteration is probably a better match for coding it than recursion. Listing 1.29 (josIter.py) is more general than Listing 1.27 since it can process different step sizes, and it also prints a list of all the eliminated numbers. The 'last man standing' is the final number in the list.

```
def josIter(n, k):
  people = list(range(1, n+1))
  elims = []      # ID numbers of
      eliminated
  pos = 0
  while people:
    pos = (pos + k - 1) % len(
        people)
    elims.append(people.pop(pos))
  return elims
```

Listing 1.29. Iterative Josephus

```
> python josIter.py
n k? 18 3
[3, 6, 9, 12, 15, 18, 4, 8, 13,
17, 5, 11, 1, 10, 2, 16, 7, 14]

> python josIter.py
n k? 16 7
[7, 14, 5, 13, 6, 16, 10, 4, 2,
1, 3, 9, 15, 8, 11, 12]
```

1.7.0.3 Josephus Visualized. An understanding of the Josephus problem is greatly enhanced by having access to a visualization tool, which you can find in josVis.py. As an example, consider the problem of removing every 3rd person in a group of 18. What's the number of the seventh person to be eliminated? Listing 1.28 reports:

```
> python joseph1.py
n k s=? 18 3 7
The 7th eliminated person is labeled #4
```

This can be pictured by running `josVis.py` with $n = 18$ and $k = 3$. The animation progresses a cycle at a time, waiting for the user to press 'n' to continue. The seventh removal occurs in the second round, to person #4:

```
> python josVis.py
n k=? 18 3
Starting at ID 1
Removed: 3 6 9 12 15
At ID 16
Removed: 18 4 8 13 17
At ID 1
```

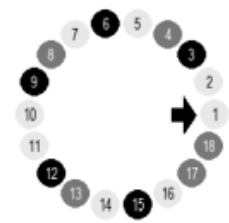


Figure 1.24. Josephus visualization

This is depicted in Fig. 1.24. Light gray circles represent people who are still alive, black circles are for people eliminated in earlier cycles, and dark gray circles are those removed during the cycle that has just finished.

We won't explain josVis.py here since the animation techniques using Pygame (`https://www.pygame.org/docs/`) aren't relevant to this problem.

Exercises for Sections 1.1 to 1.7

(1) The sequence $f(n)$ is defined by $f(1) = f(2) = f(3) = f(4) = 1$ and $f(n) = 4f(n-1) + 3f(n-2) + 2f(n-3) + f(n-4)$ for $n > 4$.

a) Write a program which prints f(20)/f(19). Hint: use the mapping $(a, b, c, d) = (b, c, d, a + 2b + 3c + 4d)$.

b) Using Listing 1.25 find the maximal root of $x^4 - 4x^3 - 3x^2 - 2x - 1 = 0$, and compare with f(20)/ f(19).

(2) Rewrite Listing 1.25 so that it works for functions increasing *or* decreasing from a to b. Also, try to remove the recursion.

(3) Find by means of Listing 1.25 the smallest positive solution of a) $x - \cos x = 0$; b) $x \ln x = 1$; c) $xe^x = 1$; d) $\tan x = x$.

(4) The equation $x^3 - 3x + 1 = 0$ has three real roots. Find them using Listing 1.25.

(5) For this and the following problems, fib(n) = f(n) and define the Lucas sequence as $g(0) = 2, g(1) = 1, g(n) = g(n-1) + g(n-2), n \geq 2$.

a) Write a program which for input n finds $g(n) - 5f^2(n)$.

b) Conjecture and prove, for instance by induction.

(6) Write a program which for input n tests if the following formulas are true:

a) $g(n) = f(n-1) + f(n+1)$;

b) $f^2(n+1) - f(n)f(n+2) = (-1)^n$.

(7) Let $f(x)$ be the probability of eventual success in the bold gamble, starting with fortune x. Find exactly

a) $f(1/3)$; b) $f(2/5)$; c) $f(1/1984)$ for $p = q = 1/2$.

(8) Here's another solution of the Josephus problem for any k by Domoryad [**Dom63**].

The number t of the s-th eliminated person can be found by means of the following algorithm: set $x = 1 + k(n-s)$ and $q = k/(k-1)$. Then generate the "integer" geometric sequence $x(1) = \lceil x \rceil, x(2) = \lceil qx(1) \rceil, x(3) = \lceil qx(2) \rceil, ...$, where $\lceil y \rceil$ denotes the ceiling of y, i.e. the smallest integer $\geq y$. If a is the largest term $\leq nk$ then $t = 1 + kn - a$.

Translate this algorithm into a program.

(9) The number $f(n)$ in the Josephus problem for $k = 2$ can also be found as follows: write n in binary, replace each 0 by -1 and you get $f(n)$. For instance, $n = 50 = 110010$ implies $f(n) = 11-1-11-1 = -1+2-4-8+16+32 = 37$. Show this.

(10) Show that for $k = 2$ in the Josephus problem the last remaining person has the number $f(n) = 1 + 2n - 2^k$ with $k = \lfloor \log_2 n \rfloor + 1$.

(11) *Duplication Formula for the Fibonacci Sequence.* Let $u(1) = 0, u(2) = 1, u(n) = u(n-1) + u(n-2)$ for $n \geq 3$. That is, u(n) = fib(n-1). Prove by induction:

$$\begin{array}{lrcll} (i) & u(2n) & = & u^2(n) + u^2(n+1), & n > 1 \\ (ii) & u(2n+1) & = & 2u(n) \cdot u(n+1) + u^2(n+1), & n \geq 1. \end{array}$$

Construct a fast recursive function based on (i) and (ii). It requires only $O(\log n)$ steps instead of $O(n)$ in fibIter() and $O(2^n)$ in fibRec(). Time the speeds of fibIter() and this function.

Remark: By means of matrices one can show that similar duplication formulas exist for all linear difference equations.

(12) Write a program which prints the sequence $J(n, k)$ of eliminations in the Josephus problem. For instance, $J(8, 3) = (3, 6, 1, 5, 2, 8, 4, 7)$. This sequence is called the *Josephus permutation*.

(13) Let $H_n = 1 + 1/2 + 1/3 + ... + 1/n$ be the harmonic numbers.

a) Find H_{10000} by adding terms from left to right.

b) Find the same sum by adding terms from right to left.

c) Which sum is more accurate and why?

(14) Find by experimentation the sum

```
round(n/2) + round(n/4) + round(n/8) + ...
```

for positive integers n. Try to prove your observations.

(15) a) Find all pairs (x, y) of positive integers from 1...100, which satisfy

$$|x^2 - xy - y^2| = 1.$$

b) How might you generate all the solutions.

(16) How many of the 6-digit blocks 000000 to 999999 have the property that the sum of the first three digits equals the sum of the last three digits. (The idea here is to reduce computation as much as possible.)

(17) The *stutter* function. Translate the functional equation

$$f(0) = 0,\ f(1) = 1,\ f(x) = \begin{cases} f(2x)/4 & \text{for } 0 < x < \frac{1}{2} \\ \frac{3}{4} + f(2x-1)/4 & \text{for } \frac{1}{2} \leq x < 1. \end{cases}$$

into a recursive program. (Theoretically, the program should stop only if x is a finite binary fraction, although the equation determines $f(x)$ uniquely for

all rational x. It can not determine $f(x)$ for irrational values unless we specify the additional condition that f be monotonic. After studying Sec. 2.1 you will be able to translate x and $f(x)$ into binary, and you'll see the reason for the function's name.

(18) Investigate how the recurrence relation:

$$x_n = x_{n-1} - \frac{2}{9}x_{n-2}$$

behaves with various starting numbers for x_0 and x_1. Verify that the ratio x_n/x_{n-1} gradually approaches 2/3 unless the initial values are in the ratio 1/3, in which case it stays at that value. Try starting with values very nearly in the ratio 1/3 to see what happens.

(19) *Factorial Sum List*. Given a number, calculate the sum of the factorials of its digits. Repeat with the new number. Stop when a repetition occurs. For example: $25 \to 122 \to 5 \to 120 \to 4 \to 24 \to 26 \to 722 \to 5044 \to 169 \to 3636001 \to 1454 \to 169$. For more information on *factorions*, see `https://en.wikipedia.org/wiki/Factorion`.

(20) Write a function that returns a count of trailing zeros in $n!$. For example, 5! ends with 1 zero, 20! has 4 zeros, and 100! finishes with 24.

(21) The Ackermann function takes two arguments, each of which can be assigned any nonnegative integer. Its definition:

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)) \end{aligned}$$

Find A(1,2) and A(1,3).

The function exhibits a phenomenal rate of growth. While the early values aren't that impressive, you'll be hard-pressed to obtain A(4, 1). Our direct translation to recursive code crashed due to a lack of recursion depth, and an iterative version took 20 minutes to finish.

Nevertheless, the Ackermann function is *well defined* in the sense that on a machine with enough memory it will always return a result. That is not the case in the next exercise.

(22) The following recursive definition, despite appearances, is not a well defined function:

$$G(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + G(\frac{n}{2}) & \text{if } n\%2 = 0 \\ G(3n-1) & \text{if } n\%2 = 1 \text{ and } n > 1 \end{cases}$$

Translate the definition into a function, and try running it on a range of positive integers. Resilient code will need to make use of Python exception handling.

Execute G(5) by hand and keep simplifying the expression. You'll end up with $G(5) = 3 + G(5)$, which is suggesting that $0 = 3$. Since the assumption that G is a function has led to a false statement, it follows that G is not a function.

(23) *The Catalan numbers* occur in various counting problems, often involving recursively defined objects. They are named after the French-Belgian mathematician Eugène Charles Catalan (1814-1894) even though they had appeared in the work of Leonhard Euler 70 years before, and were known to Chinese mathematicians before that.

The first Catalan numbers for $n = 0, 1, 2, 3, 4, , 5, ...$ are $1, 1, 2, 5, 14, 42, ...$

The recurrence relation:

$$C_0 = 1 \quad \text{and} \quad C_n = \sum_{i=1}^{n} C_{i-1} C_{n-1} \quad \text{for } n > 0$$

A non-recursive expression:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

An asymptotic approximation:

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

Implement theses definitions and compare their results.

Catalan numbers turn up in a surprisingly variety of problems, including counting the number of binary bracketings of n parentheses, the number of states possible in an n-flexagon, the number of different diagonals possible in a frieze pattern with n+1 rows, polygon triangulation, the number of rooted binary trees with n internal nodes, the number of mountains which can be drawn with n upstrokes and n downstrokes, and the number of non-crossing handshakes possible across a round table between n pairs of people. *Enumerative Combinatorics*, Vol. 1 by Richard P. Stanley [**Sta12**] describes 66 different interpretations, and a few are explained at `https://en.wikipedia.org/wiki/Catalan_number`.

(24) According to Zeckendorf's theorem (`https://en.wikipedia.org/wiki/Zeckendorf%27s_theorem`), any positive integer can be expressed as a sum of distinct non-consecutive Fibonacci numbers. Write a function that tests this theorem, as shown in the examples in Table 1.3.

n	Sum
53	[34, 13, 5, 1]
25	[21, 3, 1]
31	[21, 8, 2]
3009	[2584, 377, 34, 13, 1]

Table 1.3. Testing Zeckendorf

(25) *The Tower of Hanoi with an adjacency requirement.* Suppose that the overworked Brahmins are restricted to moving a disk only to an *adjacent* pole. Assume poles A and C are at the two ends of the row and pole B is in the middle.

Work out the number of moves required for 1, 2, 3, and 4 disks, and use this information to help you write a recurrence relation. Translate this relation into a function.

2

Algorithms in Number Theory

Number theory is a sprawling topic with a very long history, but benefits from the availability of many excellent textbooks, such as Ireland [**IR90**], Hardy [**HWHBS08**], and Niven [**NZM91**]. Silverman's *A Friendly Introduction to Number Theory* [**Sil12**] is particularly good for beginners, and several chapters are online at `https://www.math.brown.edu/johsilve/frint.html`. The *LibreTexts Mathematics* 'Combinatorics and Discrete Mathematics' section (`https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics`) includes several textbooks on number theory. Youtube has many videos on different topics, and an entire introductory course from Berkeley at `https://www.youtube.com/playlist?list=PL8yHsr3EFj53L8sMbzIhhXSAOpuZ1Fov8`.

2.1 Greatest Common Divisor

2.1.1 Euclid's Algorithm. Let a and b be integers. We denote their greatest common divisor by $gcd(a, b)$. If we define $gcd(0, 0) = 0$, then for all integers a, b

$$\gcd(a, 1) = 1, \quad \gcd(a, a) = |a|, \quad \gcd(a, 0) = |a|, \quad \gcd(a, b) = \gcd(b, a)$$

If $gcd(a, b) = 1$, we say that a and b are relatively prime, co-prime, or that one is prime to the other.

Let's assume $a \geq b > 0$, then a can be uniquely represented by b in the form

$$a = bq + r, \quad 0 \leq r < b. \tag{2.1}$$

The integers q and r are the quotient and remainder on division of a by b. Thus $q = \lfloor a/b \rfloor, r = a - \lfloor a/b \rfloor$, as depicted on the number line in Fig. 2.1.

In Python these operations are implemented as

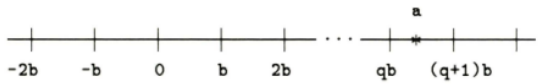


Figure 2.1. $a = bq + r$ on the number line

```
q = a // b;  r = a % b
```

Any divisor of a and b is also a divisor of $ax + by$ for any integers x, y. Thus in Equ. (2.1)

$$d|a,\ d|b \to d|r \quad \text{and} \quad d|b,\ d|r \to d|a.$$

where $a|b$ means that a *divides* b; i.e. b is a *multiple* of a for integers a, b. Hence

$$\gcd(a, b) = \gcd(b, r) = \gcd(b, a\%b). \tag{2.2}$$

By means of Equ. (2.2) we can replace the pair (a, b) by the smaller pair $(b, a\%b)$. Repeating this step, we obtain ever smaller pairs, until finally a pair $(g, 0)$ is reached. Then

$$\gcd(a, b) = \gcd(g, 0) = g.$$

This can be translated into the recursive function in Listing 2.1 (gcdrec.py), and its iterative version in Listing 2.2 (gcditer.py) is obtained by replacing the tail-recursion by a loop.

```
def gcd(a,b):
  if b == 0:
    return a
  else:
    return gcd(b, a % b)
```

Listing 2.1. Recursive GCD

```
def gcd(a,b):
  while b > 0:
    a, b = b, a%b
  return a
```

Listing 2.2. Iterative GCD

2.1.2 Euler's Totient Function.

$\phi(n)$ is the number of elements in $1..n$ which are co-prime with n.

Listing 2.3 (totient.py) computes $\phi(n)$ by using GCD to check for primality. $\phi(3000000) = 800000$ takes 46 seconds with the recursive version of GCD in Listing 2.1 but only 26 seconds with the iterative version from Listing 2.2.

```
import time
# from gcdrec import gcd
from gcditer import gcd

n = int(input("n=? "))
start_time = time.time()
```

```
phi = 0
for i in range(1,n+1):
  if gcd(n,i) == 1:
    phi += 1
print("phi(", n, ")==", phi)
print(f"Elapsed time: {(time.time() - start_time):.2f} secs")
```

Listing 2.3. Timing Euler's totient function

We could compute ϕ(n) faster by employing the formula

$$\phi(n) = n\frac{p-1}{p} \cdot \frac{q-1}{q} ...$$

where $p, q, ...$ are the *distinct* prime factors of n. For example $\phi(150) = 40$ according to the above code, and also note that $150 = 2 \cdot 3 \cdot 5^2$. This means we can also calculate the result using:

$$\begin{aligned} \phi(150) &= 150 \cdot \tfrac{2-1}{2} \cdot \tfrac{3-1}{3} \cdot \tfrac{5-1}{5} \\ &= 5 \cdot 1 \cdot 2 \cdot 4 = 40 \end{aligned}$$

This formula is derived from the Chinese Remainder Theorem (sec. 2.1.4), and is rather involved. A fairly simple explanation can be found at `https://math.stackexchange.com/questions/4346635/chinese-remainder-theorem-in-euler-phi-function`.

2.1.3 Extended Euclidean Algorithm. The GCD of a and b can be written as a linear combination of a and b with integer coefficients x, y. That is, there exist integers x, y such that

$$\gcd(a, b) = ax + by.$$

x and y can be obtained by keeping track of what happens as we carry out Euclid's algorithm. We set up two columns to record the contributions of a and b to the remainders we compute. We start with the two lines

$$\begin{aligned} a \quad 1 \quad 0 \quad &\text{(i.e. } a = 1 * a + 0 * b), \\ b \quad 0 \quad 1 \quad &\text{(i.e. } b = 0 * a + 1 * b), \end{aligned}$$

Let $q = a \mathbin{//} b$. Subtracting q-times row (2) from row (1) we get

$$r \quad 1 \quad -q \quad \text{(i.e. } r = 1 * a + (-q) * b).$$

Now we cross out row (1) and repeat the step with the remaining rows. The last line before r becomes zero will be

$$\gcd(a, b) = ax + by.$$

Listing 2.4 (egcd.py) returns $gcd(a, b)$, x, and y by using a recursive subfunction, egRec() to produce the rewrites.

```
def egcdRec(a, b):
  return egRec(a, 1, 0, b, 0, 1)

def egRec(g, x, y, g1, x1, y1):
  if g1 == 0:
    return g, x, y
  else:
    q = g // g1
    g2 = g - q*g1
    return egRec(g1, x1, y1, g2, x
        -q*x1, y-q*y1)
```

Listing 2.4. Recursive extended GCD

```
def egcd(a, b):
  x, x1 = 1, 0
  y, y1 = 0, 1
  g, g1 = a, b
  while g1 > 0:
    q = g // g1
    x, x1 = x1, x - q*x1
    y, y1 = y1, y - q*y1
    g, g1 = g1, g - q*g1
  return g, x, y
```

Listing 2.5. Iterative extended GCD

Probably the iterative version of the algorithm (Listing 2.5; egcd.py) is easier to understand since the changes to $g, x, y, g1, x1$, and $y1$ are more explicit. Examples:

```
> python egcd.py
a b? 13 7
gcd == a*x + b*y
gcd == 1 ; x == -1 ; y == 2

> python egcd.py
a b? 7 5
gcd == a*x + b*y
gcd == 1 ; x == -2 ; y == 3
```

Of course, both Listings 2.4 and 2.5 give the same answers.

2.1.4 The Chinese Remainder Theorem. The Chinese Remainder Theorem (CRT) involves $m_1, m_2, ..., m_k$ natural numbers, each greater than 1, and every pair co-prime.

Let $M = m_1 \cdot m_2 \cdot ... \cdot m_k$, and $b_1, b_2, ..., b_k$ be integers. There are a system of congruences involving x:

$$\begin{aligned} x &\equiv b_1 \pmod{m_1} \\ x &\equiv b_2 \pmod{m_2} \\ &\vdots \\ x &\equiv b_k \pmod{m_k} \end{aligned}$$

The CRT states that there is a unique solution for x somewhere in the set {0,1, 2,..., M-1}. In other words, we are given k numbers which are pairwise co-prime, and the remainders when an unknown number x is divided by them. We need to find the minimum possible value of x that produces those remainders. The CRT tells us that such an x exists $0 \leq x < M$.

An example:

$$\begin{aligned} x &\equiv 2 \pmod 3 \\ x &\equiv 3 \pmod 4 \\ x &\equiv 1 \pmod 5 \end{aligned}$$

The naive approach for finding x (see Listing 2.6; crt.py) is to start with 0 and check if dividing it by the given modulo values produces the corresponding remainders. If not, then keep incrementing x until an answer is found or M is reached. One optimization would be to first use gcd() to check that the pairs of modulo values were indeed all co-prime.

```
def crt(mods, rems):
  M = math.prod(mods)
  n = len(mods)
  x = 1
  for x in range(M): # Check x against all mods and rems
    j = 0
    while j < n:
      if x % mods[j] != rems[j]:
        break
      j += 1
    # if all remainders  matched, we found x
    if j == n:
      return x

rems = [2, 3, 1]
mods = [3, 4, 5]
print("Rems=", rems, " Mods=", mods)
print("x =", crt(mods, rems));
```

Listing 2.6. Naive CRT implementation

The program outputs:

```
> python crt.py
Rems= [2, 3, 1]  Mods= [3, 4, 5]
x = 11
```

A more efficient approach requires the calculation of the *modulo multiplicative inverse*. The inverse of integer a modulo m is x such that:

$$ax \equiv 1 \pmod m$$

A value for x can be obtained by using the egcd() function. Recall that $egcd(a, b) = g, x, y$ where $ax + by = g$. So:

$$\begin{aligned} & \text{egcd}(a, m) &&= 1, x, y \\ \equiv \quad & ax + my &&= 1 \\ & ax &&\equiv 1 \pmod m \end{aligned}$$

This can be coded as:

```
def modinv(a, m):
  g, x, y = egcd(a, m)
  if g != 1:
    raise Exception('modular inverse does not exist')
  else:
    return x % m
```

The modulo inverse gives a value for x that leaves a remainder of 1, which is a good starting point for finding values of x that give larger remainders.

A nice video introduction to CRT can be found at `https://www.youtube.com/watch?v=ru7mWZJlRQg`.

2.1.5 Visible Points in the Plane Lattice. Let L_2 denote the plane lattice, i.e. the set of points (x, y) with integer coordinates. A point (x, y) in L_2 will be *visible* from the origin if $gcd(x, y) = 1$. Otherwise, the point is *invisible*. Fig. 2.2 shows that $(9, 4)$ is visible which is confirmed by $gcd(9, 4) = 1$. In other words, since 4 and 9 are co-prime, then the diagonal from $(0, 0)$ to $(9, 4)$ does not intersect any other lattice points

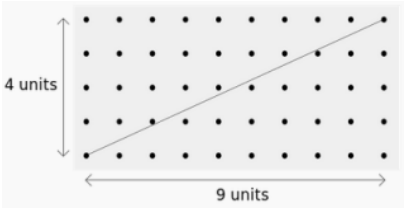


Figure 2.2. (9,4) is visible from (0,0)

Let $s(n)$ be the number of visible lattice points in the square $1 \leq x, y \leq n$. Then the proportion of visible lattice points in the square is $p(n) = s(n)/n^2$. What happens to $p(n)$ for $n \to \infty$? Each visible point (x, y) obscures infinitely many points $(kx, ky), k = 2, 3, 4, ...$ Hence we might guess that $p(n) \to 0$. Listing 2.7 (visible.py) computes $p(n)$ for input n to see if this hypothesis might be true.

```
from gcdrec import gcd

n = int(input("n=? "))
s = 0
for i in range(1, n+1):
  for j in range(1, n+1):
    if gcd(i,j) == 1:
      s += 1
p = s/n/n
print(f"s == {s} p == {p:.8f}")
```

Listing 2.7. How many visible lattice points?

n	s(n)	p(n)
100	6087	0.60870000
500	152231	0.60892400
1000	608383	0.60838300
5000	15200915	0.60803660

Table 2.1. Output from visible.py.

The stability of $p(n)$ for various values of n in Table 2.1 suggests that on the contrary $\lim_{n\to\infty} p(n) \approx 0.6080$. But $n = 5000$ isn't particularly large, so let's consider $n = 30$ million. Now we have to test $n^2 = 900$ million points, a large task. So we settle for a random sample of $m = 1$ million points and find the proportion of visible points in the sample. Listing 2.8 (randCoprime.py) estimates p(n) to be 0.60842700.

```
import random
from egcd import gcd

M = 1000000
N = 3000000
count = 0
for i in range(M):
  a = random.randint(1,N)
  b = random.randint(1,N)
  if gcd(a, b) == 1:
    count += 1

print(f"{count/M:.8f}")
```

Listing 2.8. Random testing for co-primality

We now sketch two ways of obtaining the value of the probability $p(n)$ for large values of n. To make the proof rigorous we must deal with the fact that for any finite n the equations we use are only approximately true. We should also prove that p() approaches the same limit for shapes other than squares, e.g. circles.

a) Let A_d represent $gcd(x, y) = d$. A_d is equivalent to the occurrence of three independent events: x is a multiple of d, y is a multiple of d, and $gcd(\lfloor x/d \rfloor, \lfloor y/d \rfloor) = 1$. Thus the probability of the event A_d is

$$P(A_d) = \frac{1}{d} \cdot \frac{1}{d} \cdot p \; = \; \frac{p}{d^2}.$$

Note: if your grasp of probability is a bit rusty, please read Appendix A.

Since any one of the disjoint events $A_1, A_2, A_3, ...$ must occur, we have

$$p/1 + p/4 + p/9 + ... = 1,$$

$$p(\sum_{i=1}^{\infty} 1/i^2) = 1$$

and using Euler's result $\sum_{i=1}^{\infty} 1/i^2 = \pi^2/6$, we find that $p = 6/\pi^2 \approx 0.6079271019$.

b) Let q and r be different primes. In the sequence of natural numbers we consider the subsequences $a_n = qn, b_n = rn, c_n = qrn$ of all multiples of q, r, qr. A randomly chosen natural number falls into the subsequences a_n, b_n, c_n with probabilities $1/q$, $1/r$, $1/qr$, respectively.

Since $1/(qr) = (1/q)(1/r)$, we can say that divisibility by q and r are independent events. If we choose the natural numbers x, y at random, then both will fall into a_n with probability $1/q^2$, and q is not a common factor of x and y with probability $1 - 1/q^2$. The events that neither x nor y is divisible by q are independent for different primes q. So the probability for $gcd(x, y) = 1$ is

$$\begin{aligned} p &= \prod(1 - \frac{1}{q^2}) \quad \text{(over all primes } q\text{)}, \\ \frac{1}{p} &= \frac{1}{1-1/2^2} * \frac{1}{1-1/3^2} * \frac{1}{1-1/5^2} * ..., \\ &= \sum_{n\geq 0} \frac{1}{2^{2n}} * \sum_{n\geq 0} \frac{1}{3^{2n}} * \sum_{n\geq 0} \frac{1}{5^{2n}} * ... \\ &= \sum_{n\geq 1} \frac{1}{n^2} = \frac{\pi^2}{6} \end{aligned}$$

By multiplying out the brackets we got the reciprocal of each square exactly once. This follows from the unique factorization theorem. Hence

$$p = \frac{6}{\pi^2} \approx 0.6079271019$$

How can we find the sum $s = \sum_{n\geq 1} \frac{1}{n^2}$ to the maximum accuracy of real arithmetic in Python, 15 digits, without using Euler's result? Let $s = 1 + 1/4 + ...+1/n^2+E(n)$, and we want to estimate the error $E(n) = 1/(n+1)^2+1/(n+2)^2+...$

For large values of k, k^2 is relatively close to $(k - \frac{1}{2})(k + \frac{1}{2})$ and hence

$$\frac{1}{k^2} \approx \frac{1}{(k-\frac{1}{2})(k+\frac{1}{2})} = \frac{1}{k-\frac{1}{2}} - \frac{1}{k+\frac{1}{2}}.$$

Consequently

$$\begin{aligned} E(n) &= \frac{1}{(n+1)^2} + \frac{1}{(n+2)^2} + ... \\ &\approx \left(\frac{1}{n+\frac{1}{2}} - \frac{1}{n+\frac{3}{2}}\right) + \left(\frac{1}{n+\frac{3}{2}} - \frac{1}{n+\frac{5}{2}}\right) + ... \\ &= \frac{1}{n+\frac{1}{2}}. \end{aligned}$$

The right side is $> E(n)$, but by a factor $\leq (n+1)^2/((n+1)^2 - 1/4)$. Having found a good estimate of E(n), we compute $s(n) = 1 + ... + 1/n^2$ starting at the end (to minimize rounding errors) and add the correcting term $1/(n+0.5)$ (Listing 2.9; sumInvSquares.py). We test $n =$ 50000, 100000, ... until there is no further change, producing Table 2.2.

```
n = int(input("n=? "))
sum = 0.0
for i in range(n, 0 ,-1):
  sum += 1.0/(i*i)
sum += 1.0/(n+0.5)
print(sum)
```

Listing 2.9. Sum inverse squares

n	sum (n)
50000	1.6449340668482269
100000	1.6449340668482266
150000	1.6449340668482264
200000	1.6449340668482264

Table 2.2. sum Inverse Squares Results

```
def gcdBinary(u,v):
  if u == v:
    return u
  if (u%2==0) and (v%2==0): #even
    return 2*gcdBinary(u//2, v//2)
  if (u+v)%2 == 1:
    if v%2 == 1:
      return gcdBinary(u//2, v)
    else:
      return gcdBinary(u, v//2)
  if (u%2 == 1) and (v%2 == 1):
    if u > v:
      return gcdBinary(u-v, v)
    elif v > u:
      return gcdBinary(u, v-u)
```

Listing 2.10. Recursive binary GCD

```
def gcdBinaryIt(u,v):
  x = u; y = v
  k = 1
  while (x%2==0) and (y%2==0):
    x = x//2;  y = y//2
    k *= 2
  while x != y:
    while x%2 == 0:
      x = x//2
    while y%2 == 0:
      y = y//2
    if x > y:
      x -= y
    if y > x:
      y -= x
  return x*k
```

Listing 2.11. Iterative binary GCD

2.1.6 The Binary GCD Algorithm. The binary GCD algorithm is based on the relations

$$\begin{aligned} u \text{ even}, v \text{ even} &\rightarrow \gcd(u,v) = 2 * \gcd(u /\!\!/ 2,\ v /\!\!/ 2) \\ u \text{ even}, v \text{ odd} &\rightarrow \gcd(u,v) = \gcd(u /\!\!/ 2,\ v) \\ u > v &\rightarrow \gcd(u,v) = \gcd(u - v, v) \\ u \text{ odd}, v \text{ odd} &\rightarrow u - v \text{ even}, |u - v| < \max(u,v). \end{aligned}$$

This algorithm can be made a little faster by employing Python's bitwise right shift operator (»), although the performance difference from using integer division (//) will be small. Indeed, many compilers automatically convert power-of-two multiples and divides into left and right bit shifts when the shift amount is known at compile time (as here).

Listing 2.10 (gcdBinary.py) is recursive, and closely matches the definition above, while Listing 2.11 (gcdBinaryIt.py) is an iterative version.

2.1.7 GCD and LCM. The following algorithm due to Stanley Gill [**Dij76**] computes $gcd(a, b)$ and $lcm(a, b)$ at the same time, employing the fact that $lcm(a, b) = ab/gcd(a, b)$.

We start with $x = a, y = b, u = a, v = b$ and progress as follows:

```
if x < y:
  y -= x; v += u
if y < x:
  x -= y; u += v
```

The algorithm ends with $x = y = gcd(a, b)$ and $(u + v)/2 = lcm(a, b)$. The invariants of these transformations are

$$\begin{array}{lcl} P & : & \gcd(x, y) = \gcd(x - y, y) = \gcd(x, y - x), \\ Q & : & xv + yu = 2ab, \\ R & : & x > 0, y > 0. \end{array}$$

P and R are obviously invariant, but we'll show the invariance of Q. At the beginning we have $ab + ab = 2ab$, which is clearly correct. At the next step, the left side of Q becomes either

$$x(v + u) + (y - x)u = xv + yu \quad \text{or} \quad (x - y)v + y(u + v) = xv + yu.$$

Thus, the left side of Q does not change. At the end we have

$$x = y = \gcd(a, b) \quad \text{and} \quad \frac{u + v}{2} = \frac{ab}{\gcd(a, b)} = \operatorname{lcm}(a, b).$$

Listing 2.12 (gcdlcm.py) implements this algorithm.

```
def gcdlcm(u, v):
  x = u; y = v
  while x != y:
    if x < y:
      y -= x
      v = u+v
    else:
      x -= y
      u = u+v
  return x, (u+v)//2   # gcd, lcm
```

Listing 2.12. Calculating GCD and LCM

Exercises

(1) Let $a = 8991, b = 3293$.

a) Find $gcd(a, b)$.

b) Find a solution (x, y) of $gcd(a, b) = ax + by$ using Listing 2.4.

c) Find $gcd(a, b)$ using Listing 2.10.

(2) Find $gcd(a, b)$ and $lcm(a, b)$ for $a = 2431$ and $b = 1309$ using Listing 2.12.

(3) Let a, b be integers made up of at most 11 digits. Find $gcd(a, b)$ for

a) $a = 987654321, b = 123456789$

b) $a = 9753197531, b = 2468008642$

c) $a = 12345678901, b = 10234567891$.

(4) Find the last three (or four) digits of a) 7^{9999} b) 7^{99999} c) 7^{999999}

(5) Make Listing 2.7 twice as efficient by using the fact that if $t(n)$ is the number of visible points in the triangle $0 < y < x \leq n$, then $s(n) = 2t(n) + 1$.

(6) Write an iterative and a recursive function to find $1 + 2 + ... + n$.

(7) Write a recursive and an iterative function to find the binary representation of n.

(8) Write a function which computes the digital sum $d(n)$ of n. For instance, $d(1234) = 10$.

(9) Let $d(n)$ be the digital sum of n. Write a program which finds $d(d(n)), d(d(d(n))), ...$ until a single digit number $f(n)$ is reached. Show that $f(mn) = f(f(m)f(n))$.

(10) Write a function which reverses the digits of an integer. For instance, $2024 \mapsto 4202$.

(11) Write a slow recursive function for $C(n, s)$, the number of s-subsets of an n-set, based on the recursion $C(n, s) = C(n-1, s-1) + C(n-1, s), C(n, 0) = C(n, n) = 1$.

(12) Write a fast recursive function for $C(n, s)$ based on the recursion $C(n, s) = C(n-1, s-1) * n/s, C(n, 0) = 1$.

(13) Write a program which prints fib(n+1)/fib(n) for $n = 1$ to 30. What do you see, and can you prove it?

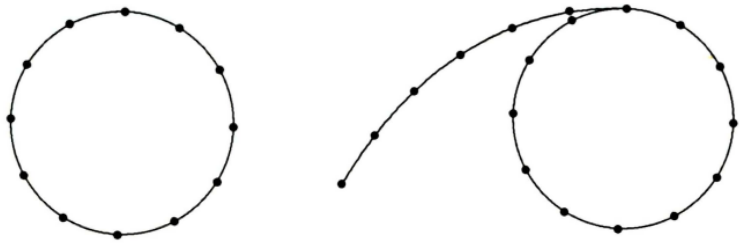
(14) Print those terms of the Fibonacci sequence which are divisible by some fixed positive integer d. Experiment with different values of d, make observations and conjectures. Prove your conjectures.

(15) The Fibonacci sequence $\pmod m$ will eventually become periodic.

a) Why?

b) A periodic sequence can be either purely periodic or periodic with a tail, as in Figs. 2.3a and 2.3b. Show that the Fibonacci sequence $\pmod m$ is always purely periodic.

c) Write a program which finds the period $L(m)$ for the modulus m.



(a) An immediate cycle. (b) An eventual cycle.

Figure 2.3. Periodic sequences

(16) Experiment with the program in Ex. 15c) and make conjectures about

a) $L(p^n)$ for any prime p.

b) $L(p_1^{a_1}, p_1^{a_2}, ..., p_1^{a_r},)$ for primes p_1 to p_r.

By means of b), conjecture a formula for $L(10^n)$.

(17) Formulate theorems about the length L of the period of the Fibonacci sequence $\pmod m$ if the prime p has the form

a) $p = 5k \pm 1$ b) $p = 5k \pm 2$ c) $p = 5$

(18) The Tribonacci sequence is defined by $t(1) = t(2) = t(3) = 1, t(n) = t(n-l) + t(n-2) + t(n-3), n > 3$.

a) Why is this sequence purely periodic $\pmod m$?

b) Find the period $L(m)$ for the modulus m and discover some theorems about $L(m)$.

(19) Find Zeta3 $= 1 + 1/2^3 + 1/3^3 + 1/4^3 + ...$, in a way similar to Listing 2.9.

(20) Define a lattice point (x, y, z) to be visible if $gcd(x, y, z) = 1$. Rewrite Listings 2.7 and 2.8 for 3D space. Guess a formula for the probability that $gcd(x, y, z) = 1$. Can you give a heuristic argument for this formula?

(21) Guess a formula for the probability that k random integers have a $GCD = d$. Check this formula empirically. Give a heuristic argument for this formula.

(22) *The Morse-Thue Sequence.* This infinite sequence of binary digits can be constructed recursively: start with 0; to each initial segment append its complement: 0, 01, 0110, 0110 1001, 01101001 10010110, ... Thus, having constructed the first $1 + 1 + 2 + 4 + 8 + 16 + ... + 2^n = 1 + 2^{n+1}$ digits, the next block of $1 + 2^{n+1}$ is determined.

a) Show that the sequence is aperiodic.

b) The sequence has a much more interesting property: it is self-similar. Striking out every second digit reproduces the sequence. Prove this.

c) Let the digits of the sequence be $x(0), x(1), x(2), ...$ Show that $x(2n) = x(n), x(2n+1) = 1 - x(2n)$. Write a recursive algorithm to find $x(n)$ based on these recurrences.

d) Show that $x(n) = 1 - x(n - 2^k)$, where 2^k is the largest power of 2 contained in n, i.e. the largest power of 2 which is $\leq n$. Write an algorithm to find $x(n)$ based on this property.

e) Write the non-negative integers in binary notation: 0, 1, 10, 11, 100, 101, 110, 111, ... Now replace each number by the sum of its digits $\pmod 2$. You get the sequence 01101001..., which is the Morse-Thue sequence once again. Prove this and write an algorithm for finding the n-th digit $x(n)$ based on this idea.

This exercise and the next were inspired by Jacobs [**Jac69**].

(23) The binary *Keane Sequence* $x(0)\, x(1)\, x(2)... = 00100111000100111011011000 1...$ is formed as follows (with commas separating the successively longer segments):

$$0, 001, 001001110, ..., A, AAC(A), ...$$

where $C(A)$ is the digit-wise complement of the binary block A.

a) Construct an algorithm which finds the n-th digit $x(n)$. *Hint*: translate 0,1,2,3, ... into the ternary system. Do you see a pattern?

b) Show that $x(3n) = x(n), x(3n+1) = x(n), x(3n+2) = 1 - x(n)$. Write a program based on these recurrences.

(24) Rewrite Listing 2.11 to use bit shifting as explained in the text. Test its speed versus the GCD function currently used in Listing 2.8).

(25) a) Print one row of Pascal's triangle, i.e. the sequence $C(n, 0)...C(n, n)$. Use a list generated by addition, as in Table 2.3.

b) Modify a), so that rows 0 to n of Pascal's triangle are printed.

c) Write a function which prints Pascal's triangle $\pmod 2$ and study the recursive pattern.

1	5	10	10	5	1	
	1	5	10	10	5	1
1	6	15	20	15	6	1

Table 2.3. Pascal Triangle row

(26) a) A woman is walking to the market with n eggs in her basket, and a man on a bicycle knocks her down, causing all the eggs to break. The cyclist offers to pay for all n eggs, but she can't remember n exactly. She only knows that by grouping them 2,3,4,5,6 at a time, one egg was always left over, but when they were grouped into 7's, no eggs were surplus. Find all numbers ≤ 3000 satisfying these properties.

b) Which is the most likely number if one egg weighs about 56 grams?

c) Generalize.

(27) Consider Perrin's sequence $v_0 = 3, v_1 = 0, v_2 = 2, v_n = v_{n-2} + v_{n-3}, n > 2$. Write a program which prints a table of those numbers satisfying $3 \leq n \leq 4001$ and $n|v_n$ Compare this table with the table printed in Listing 2.19. Do you think there is enough evidence for the theorem $n|v_n \Leftrightarrow n$ is prime? We will return to this problem later.

(28) *Analysis of an Algorithm.* Start with two piles of a and b chips. You may double the number of chips in the smaller pile by taking them from the larger one, but must stop as soon as the first pile is reduced to 0. This can be summarized as:

```
while a > 0:
  if a < b:
    (a, b) = (2a, b-a)
  else:
    (a, b) = (a-b, 2b)
```

a) For what initial positions does the algorithm stop, and after how many steps?

b) When do you get a pure cycle (e.g. like Fig. 2.3a), and how long is it?

c) When do you get a cycle with a tail (e.g. like Fig. 2.3b), and how long is the tail?

d) Suppose a and b are positive rationals. Answer a) to c) in that case.

e) Let a and b be positive reals. Now answer a) to c) in that case. Hint: you can simplify the algorithm substantially by noting that $a + b$ is an invariant of the transformation.

(29) Find a recursive equation for the number $f(n)$ of ones in the binary representation of n. Hint: express $f(n)$ for $n > 0$ in terms of values of $f()$ for smaller arguments.

(30) By a systematic search find a number n with the property $\phi(n) = \phi(n+1) = \phi(n+2)$. See Listing 2.1.2 for more details on $\phi()$.

(31) $f()$ is defined by $f(1) = 1, f(2n) = f(n)+g(n-1), f(2n+1) = f(n+1)+g(n/2)$ for all positive integers n. Here, $g(x)$ is the smallest power of $2 > x$, where x is a real. Write a recursive program which tests that $f(f(n)) = n$ for any input n. A function with this property is called an *involution*.

2.2 Representing n in the Form $x^2 + y^2$

We want to generate all non-negative integer solutions of $x^2 + y^2 = n$ with $\sqrt{n} \geq x \geq y \geq 0$. It pays to subdivide the task into three parts:

```
1. Initialization.
2. Go from A to B.
3. Go from B to C.
```

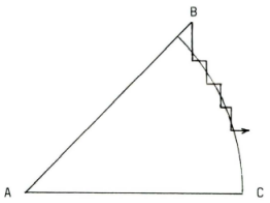


Figure 2.4. B to C

Stage 3 is illustrated in Fig. 2.4. The resulting code is in Listing 2.13 (quadsum1.py).

```
n = int(input("n=? "))
x = 0; y = 0
while x*x + y*y < n:
  # go from A to B
  x += 1
  y += 1
while x*x <= n:
  # go from B to C
  while x*x + y*y > n:
    y -= 1
  if x*x + y*y == n:
    print("x ==", x, "; y ==", y)
  x += 1
```

Listing 2.13. Printing integer squares

Stage 2 (going from A to B) can be optimized to a single step, resulting in Listing 2.14 (quadsum2.py).

```
n = int(input("n=? "))
x = math.trunc(math.sqrt(n/2))
y = x   # jump from A to B
while x*x <= n:
  # go from B to C
  while x*x + y*y > n:
    y -= 1
  if x*x + y*y == n:
```

```
    print("x ==", x, "; y ==", y)
  x += 1
  y -= 1
```

Listing 2.14. Printing integer squares (optimized)

We've utilized the fact that the part of the circle we are concerned with (i.e. B to C) descends more steeply than 45°. Consequently, if $(x, y+1)$ is outside the circle then so is $(x+1, y)$. That allows us to bypass the latter point and go directly to $(x+1, y-1)$.

2.3 Pythagorean Triples

We now extend Listing 2.14 so it prints a table of primitive Pythagorean triples, i.e. all the positive integer solutions (x, y, z) such that

$$x^2 + y^2 = z^2,\ \gcd(x, y) = 1,\ z \leq \text{ max}.$$

z must be odd because if it was even then x and y would have the same parity. Also, if x and y are both even then the triple can't be primitive. If they're both odd then $x^2 + y^2 = z * 2$ cannot be true modulo 4.

These conditions mean that we only need to run through the odd values of z, and print an (x, y, z) triple only if $gcd(x, y) = 1$. The result is Listing 2.15 (pyttrip1.py).

```
max = int(input("max=? "))
z = 1
while z <= max:
  z += 2
  n = z*z
  x = math.trunc(math.sqrt(n/2))
  y = x
  while x*x <= n:
    while x*x + y*y > n:
      y -= 1
    if x*x + y*y < n:
      x += 1
    else:
      if gcd(x,y) == 1:
        print(f"{y:4} {x:4} {z:5}")
      x += 1
      y -= 1
```

Listing 2.15. Prints all triples with z ≤ max

The values in Table 2.4 are output when max is set to 100.

Primitive Pythagorean triples (x, y, z) can also be represented as

$$x = a^2 - b^2, \qquad y = 2ab, \qquad z = a^2 + b^2,$$
$$a > b, \qquad \gcd(a, b) = 1, \qquad a + b \text{ odd};$$

y	3	5	8	7	20	12	9	28	11	33	16	48	36	13	39	65	20
x	4	12	15	24	21	35	40	45	60	56	63	55	77	84	80	72	99
z	5	13	17	25	29	37	41	53	61	65	65	73	85	85	89	97	101

Table 2.4. Pythagorean triples

Listing 2.16 (pyttrip2.py) is based on these formulas. Note that max is now linked to a rather than z, and so a slightly larger number of triplets is printed since z can increase above 101. One useful change to both of these programs would be to store the triplets in a list, which could then be sorted.
For an excellent introduction to this way of solving the Pythagorean equation (and the rest of number theory), see Ore [**Ore17**].

```
max = int(input("max=? "))
a = 2; b = 1
while a <= max:
  while b > 0:
    if gcd(a,b) == 1:
      print(f"{a*a - b*b:4} {2*
          a*b:4} {a*a + b*b:5}"
          )
    b -= 2
  b = a
  a += 1
```

Listing 2.16. Pythagorean triplets

2.4 Counting the Lattice Points in a Ball

Let $lat(s, n)$ be the number of lattice points in the closed n-ball

$$x_1^2 + x_2^2 + ... + x_n^2 \leq s.$$

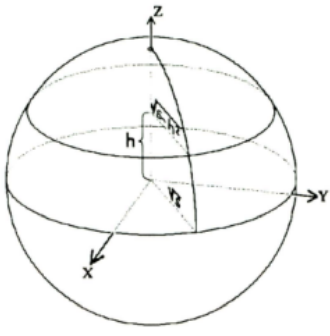
In Fig. 2.5a we use $n = 3$ for illustration. The equatorial section is an $(n-1)$-ball with radius $\sqrt{s}$.

By definition there are $lat(s, n-1)$ lattice points on the ball. To those we must add the latitudinal sections at heights $h = 1, 2, ..., \lfloor\sqrt{s}\rfloor$ above the equatorial section.

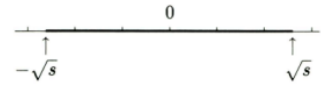
In Fig. 2.5a, the latitudinal section with height h is an $(n-1)$-ball with radius $\sqrt{s-h^2}$ which contributes $lat(s-h^2, n-1)$ lattice points.

For symmetry reasons we ignore the sections in the "southern hemisphere" and instead count the sections above the equator twice. Thus, we get the program fragment:

```
count = lat(s, n-1)
for h in range(int(math.sqrt(s)):
  count += 2 * lat(s-h*h, n-1)
```



(a) A 3D ball



(b) A 1D ball for $x^2 \leq s$

Figure 2.5. Lattice balls

The cleanest termination condition would be:

```
if n == 0:
  lat = 1
```

That is, the 0-ball contains just the origin. Or we could set $lat(s, 1) = 1+2\lfloor\sqrt{2}\rfloor$ (Fig. 2.5b). Then the termination condition would become:

```
if n == 1:
  lat = 1 + 2*int(math.sqrt(s))
```

Listing 2.17 (lattice.py) is based on this approach.

```
def lat(s, n):
  if n == 1:
    return 1 + 2*int(math.sqrt(
        s))
  else:
    count = lat(s,n-1)
    h = 1
    while h*h <= s:
      count += 2*lat(s-h*h, n
          -1)
      h += 1
    return count
```

Listing 2.17. Number of lattice points

Table 2.5 was computed using this function, with s varying as in the table and n set to 2 (i.e. we generated lattice points inside a 2D ball, better known as a circle).

The code is arguably the simplest way to compute π (3.141592653589793) – by counting. The main drawback is that errors will start to accumulate due to all the calls to math.sqrt().

In 1800, Gauss developed a simple estimate for the error bound for the circle's lattice points (the Gauss circle problem; `https://mathworld.wolfram.com/GausssCircleProblem.html`). To each point in or on the circle he assigned the unit square "north-east" of the point. The sum of the areas of these squares is $f(s)$:

$$f(s) = \text{lat}(s, 2) = \pi s + e(s),$$

where $e(s)$ is the error.

s	lat(s, 2)
10	37
100	317
1000	3149
10000	31417
100000	314197
1000000	3141549
10000000	31416025
100000000	314159053
1000000000	3141592409
10000000000	31415925457

Table 2.5. Lattice points in a circle

This is not quite equal to the actual area of the circle since some squares protrude beyond it and there are some unfilled areas, as can be seen in Fig. 2.6.

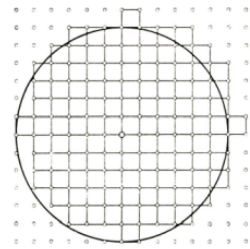


Figure 2.6. A 2D lattice ball

However, the circle about the origin with radius $\sqrt{s}+\sqrt{2}$ covers these squares completely and the circle about the origin with radius $\sqrt{s}-\sqrt{2}$ is completely covered by these squares. Thus, we have the bounds:

$$\pi(\sqrt{s}-\sqrt{2})^2 < f(s) < \pi(\sqrt{s}+\sqrt{2})^2$$

which implies

$$|f(s)-\pi s| < 2\pi(\sqrt{2s}+1)).$$

We can write this less precisely, for large s, as

$$|f(s)-\pi s| < C\sqrt{s}, \text{ where C is some constant}$$

It took over 100 years to find a better estimate, when Sierpinski proved the surprising result in 1906:

$$|f(s)-\pi s| < Cs^{1/3}, \text{ where C is some constant}$$

Hardy and Landau followed this by showing that there is **no** constant C such that

$$|f(s)-\pi s| < Cs^{1/4}.$$

It's currently believed that, for each t > 1/4, there is a number C_t such that

$$|f(s)-\pi s| < C_t s^t,$$

but as of 2003 this has only been proved for $t > 131/416$.

Exercises for Sections 2.2 to 2.4

(1) Find the number of representations as a sum of two squares of the following integers:
$5, 13, 17, 29, 5*13 = 65, 5*13*17 = 1105, 5*13*17*29 = 32045, 5^2*13 = 325,$
$5^3 * 13 = 1625, 5^2 * 13^2 = 4225, 5^3 * 13^3$. Devise theorems about the number of representations.

(2) Write a program which finds all lattice points on the circle $x^2 + y^2 = n$ by starting at C and going to B (as depicted in Fig. 2.4).

(3) If $n \bmod 4 = 3$ then $x^2 + y^2 = n$ has no solutions. Show this. Use this fact to write a brute force algorithm for finding all Pythagorean triples. The variable a runs from 2 to $c - 1$, and print a, b, c if $b = \sqrt{c^2 - a^2}$ is an integer. How do you test if b is an integer? Only one of a, b, c and b, a, c should be printed.

(4) Make Listing 2.14 more efficient by using the fact that

$$(x + 1)^2 = x^2 + (2x + 1), \quad (y - 1)^2 = y^2 - (2y - 1).$$

(5) There is another formula for the number $f(s)$ of lattice points in the circle $x^2 + y^2 \le s$:

$$f(s) = 1 + 4(\lfloor s \rfloor - \lfloor s/3 \rfloor + \lfloor s/5 \rfloor - \lfloor s/7 \rfloor + \lfloor s/9 \rfloor - ...). \tag{2.3}$$

Find by means of this formula a) $f(10000)$ b) $f(1000000)$. Hint for b): with $s = 500000$ we have $\lfloor 2s/(s+1) \rfloor - \lfloor 2s/(s+3) \rfloor + ... + \lfloor 2s/(2s-3) \rfloor - \lfloor 2s/(2s-1) \rfloor = +1 - 1 + 1 - 1 + ... + 1 - 1 = 0$. Using similar relations reduce the number of terms from 500000 to about 50000.

Equ. (2.3) is proved in Hilbert and Cohn-Vossen [**HCV99**] and Shanks [**Sha78**].

(6) Evaluate lat(10000, 3) and give an estimate of π based on this number. Compare it with the estimate obtained with lat(10000, 2).

(7) The volume of the 4-ball is $v_4(r) = c_4 r^4$, where c_4 is the volume of the unit ball in 4-space. Estimate the constant c_4 by counting lattice points using lat(100, 4) and lat(1000, 4), which give estimates for c_4.

a) Can you guess the exact value of c_4?

b) Divide the estimate by π. Can you now guess c_4?

c) Multiply the estimate by 2 and divide by π. Can you now guess c_4?

(8) The circle occupies $\pi/4$ (roughly 3/4) of the circumscribed square. The 3-ball occupies $\pi/6$ (roughly 1/2) of the circumscribed cube. Roughly what proportion of the circumscribed cube do the 4-, 5-, and 6-balls occupy?

(9) The following formula was known to Gauss:

$$\text{lat}(s,2) = 1 + 4\left\lfloor\sqrt{s}\right\rfloor + 4\left\lfloor\sqrt{s/2}\right\rfloor^2 + 8\sum_{i=a}^{b}\left\lfloor\sqrt{s-i^2}\right\rfloor$$
$$\text{with } a = \left\lfloor\sqrt{s/2}\right\rfloor + 1 \text{ and } b = \left\lfloor\sqrt{s}\right\rfloor$$

Prove this formula. Write a program based on it, and compare its speed with that of Listing 2.17.

(10) From Equ. 2.3 in Ex. 5 we can derive Leibniz's formula:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$$

However, this formula was known at least a century earlier in India with three successively better error estimates:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \ldots + \frac{1}{(n-3)} - \frac{1}{(n-1)} + f(n);$$
$$f(n) \approx \frac{1}{2n}, \quad f(n) \approx \frac{n}{2n^2+2}, \quad f(n) \approx \frac{n^2+4}{2n^3+10n}.$$

Find π using the above sum and third remainder formula with $n = 20$ and $n = 50$.

Sources for this formula can be found in C.T. Rajagopal and M.S. Rangachari, "On an Untapped Source of Medieval Keralese Mathematics", *Archive for History of Exact Sciences* 18 (1978), 81-101 and "On Medieval Kerala Mathematics", 35 (1986), 91-99.

2.5 Sieves

Sieving is when we strike out all elements from a set which don't possess a certain property. Sieves play a fundamental role in computer science and mathematics.

2.5.1 Square-free Integers. An integer is called *square-free* if it's not divisible by the square of an integer > 1. Let $A(n)$ be the number of square-free integers in the set $1..n$, and $q(n) = A(n)/n$ is the fraction of square-free integers in that range. If

$$q = \lim_{n\to\infty} q(n)$$

exists, we call it the *density* of square-free integers. We'll compute $q(500)$ to $q(15000)$ in steps of 500 in Listing 2.18 (`squareFree.py`) to get some idea of q.

```
n = int(input("n=? "))
sqs = [False]*(n+1); i = 2
while (i * i <= n):
  for k in range(0, n+1, i*i):
    sqs[k] = True
  i += 1
s = 0
print("  n       q(n)")
for i in range(1, n+1):
  if not sqs[i]:
    s += 1
  if i % 500 == 0:
    print(f"{i:5} {s/i:10.8f}")
```

Listing 2.18. Counting square-free integers

We set each element of the list sqs to False to indicate that all of its values are initially not divisible by a square. Then we sieve with square factors (e.g. test i^2, $2i^2$, $3i^2$, ...) for $k = 0 \cdots n$, and if k is divisible by one of them then it is 'marked' by setting sqs[k] = True.

```
> python squareFree.py
n=? 15000
  n        q(n)
  500 0.61200000
 1000 0.60800000
 1500 0.61000000
  :  # more lines, not shown
14000 0.60778571
14500 0.60772414
15000 0.60800000
```

The output suggests that $q \approx 0.608$, and we conjecture that $q \approx 6/\pi^2$ (0.607927). The exercises explore this in more detail.
The printed sequence gets closest to $6/\pi^2$ at $n = 11000$, and then moves away. Is this due to accumulated errors or a property of the sieving?

2.5.2 The Sieve of Eratosthenes. The following algorithm is attributed to Eratosthenes of Cyrene, a 3rd Century BCE Greek mathematician:

(1) Set $p = 2$.

(2) Strike out all multiples of $p \geq p * p$.

(3) Set $p =$ the first integer beyond p not yet struck out, and jump to 2.

```
n = int(input("n=? "))
primes = [True]*(n+1); p = 2
while (p * p <= n):
  if primes[p]:
    for i in range(p*p, n+1, p):
      primes[i] = False
  p += 1
count = 1
for p in range(2, n+1):
  if primes[p]:
    print(f"{p:6}", end='')
    count += 1
  if count % 10 == 0:
    print(); count = 1
```

Listing 2.19. The sieve of Eratosthenes

The primes $\leq$ 200 produced by eratsiev.py:

```
> python eratsiev.py
n=? 200
     2     3     5     7    11    13
    17    19    23    29    31    37
    41    43    47    53    59    61
    67    71    73    79    83    89
    97   101   103   107   109   113
   127   131   137   139   149   151
   157   163   167   173   179   181
   191   193   197   199
```

2.5.3 A Closed Formula for an Irregular Sequence.

The sequence a_n:

```
1, 3, 4, 6, 8, 9, 11, 12, 14, 16, 17, 19, 21, 22, 24, 25, 27...
```

was produced by the following sieve:

- Take $a_1 = 1$ and delete $a_1 + 1 = 2$.
- Keep the next integer $a_2 = 3$, but delete $a_2 + 2 = 5$.
- Keep the next integer $a_3 = 4$, but delete $a_3 + 3 = 7$, etc.

The program makes use of two lists, x and a, as depicted in Fig. 2.7.

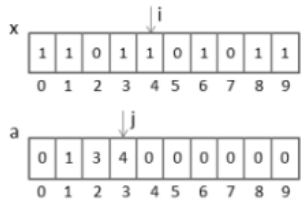


Figure 2.7. Irregular sequence data structures

The x list is used as a source of available numbers, such that if $x[i] = 1$ then the number i is available. If $x[i] = 0$ then it has been 'deleted' by the sieve. The a list holds the growing irregular sequence.

Listing 2.20 (irrSieve.py) finishes by calculating $a[j]/j$ and plots the values in the a list.

```
n = int(input("n=? "))
x = [1]*(2*n+1)  # available numbers
a = [0]*(2*n+1)  # irregular sequence
i = 1; j = 1     # indicies for the two lists
a[j] = 1
while i < n:
  x[i+j] = 0    # delete number i+j
  while True:   # look for an available number
    i += 1
    if x[i] == 1:   #  number i is available
      break
  j += 1
  a[j] = i   # store i in irregular sequence
print(f"{a[j]/j:.8f}")
plt.scatter(range(j+1), a[:j+1])
plt.xlabel('j')
plt.ylabel('a[j]')
plt.title('Irregular Sieve (n='+str(n)+')')
plt.show()
```

Listing 2.20. Irregular sequence sieve

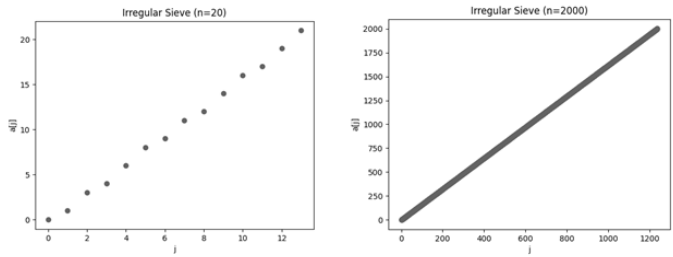


Figure 2.8. Two plots of the irregular sequence for $n = 20$ and $n = 2000$

Our aim is to find a closed formula for a_n. A look at Fig. 2.8 shows that a_n grows almost linearly. It seems likely that $a_n \approx t_n$, where t is some irrational number which accounts for the small but erratic fluctuation about a straight line. To find a good approximation for t we must evaluate a_n/n for large n.

The values for $a[j]/j$ printed by the program are 1.61538462 for $n = 20$ and 1.61762328 when $n = 2000$. This is close to the golden ratio $\frac{1}{2}(1 + \sqrt{5}) = 1.61803....$

We can convince ourselves that a_n/n approaches the golden ratio by the following argument. The number of integers in the complementary sequence up to $a_n + n$ is n. (Two sequences of natural numbers are *complementary* if together they contain each natural number exactly once.)

The number of numbers in the a sequence up to that point is a_n. So $a_{a_n} \approx a_n + n$. Assuming that $a_n = t_n$, we get $t^2 n \approx tn + n$ and hence $t^2 = t + 1$. This equation is satisfied by the golden ratio.

Also, since a_n/n is always smaller than t, then we can conjecture that $a_n = \lfloor t_n \rfloor$. This can be confirmed by adding the following to the program:

```
isSmallT = True
t = (1+math.sqrt(5))/2
for i in range(j):
  if a[i] != math.trunc(t*i):
    isSmallT = False
print("Smaller than t:", isSmallT)
```

2.5.4 An Olympiad Problem. Do there exist 1983 distinct positive integers ≤ 100000 with no three in arithmetic progression? (Problem 5 of the International Mathematical Olympiad XXIV, Paris 1983; available at `https://www.imo-official.org/problems.aspx`.)

To gather some numerical evidence we construct a sequence a_j with no three elements in arithmetic progression by using the following "greedy" algorithm: $a_0 = 0, a_1 = 1, a_j =$ the smallest positive integer, which does not form an arithmetic progression with any two previous terms.

We start with two lists, x and a, and set $a[0] = 0, a[1] = 1, x[i] = 1$ for $i = 2$ to 8000. The terms $a < b < c$ are in arithmetic progression if $2b = a + c$, or $c = 2b - a$. Of the numbers $i = 2$ to 8000 we sieve out those which are in arithmetic progression with any two of the preceding elements $a[0], ..., a[j-1]$. The first i that remains is $a[j]$.

Listing 2.21 (olympiad.py) is the resulting program, and generates the following sequence.

```
N = 8000; M = 132
x = [1]*N
a = [0]*N
a[1] = 1
i = 1; j = 1
while (j < M):
  for k in range(j):
    x[2*i - a[k]] = 0
  i += 1
  while x[i] != 1:
    i += 1
  j += 1
  a[j] = i
for i in range(M):
  print(f'{a[i]:6}',end='')
  if (i+1) % 12 == 0:
    print()
```

Listing 2.21. Olympiad sieving

```
     0      1      3      4      9     10     12     13     27     28     30     31
    36     37     39     40     81     82     84     85     90     91     93     94
   108    109    111    112    117    118    120    121    243    244    246    247
   252    253    255    256    270    271    273    274    279    280    282    283
   324    325    327    328    333    334    336    337    351    352    354    355
   360    361    363    364    729    730    732    733    738    739    741    742
```

```
 756   757   759   760   765   766   768   769   810   811   813   814
 819   820   822   823   837   838   840   841   846   847   849   850
 972   973   975   976   981   982   984   985   999  1000  1002  1003
1008  1009  1011  1012  1053  1054  1056  1057  1062  1063  1065  1066
1080  1081  1083  1084  1089  1090  1092  1093  2187  2188  2190  2191
```

We observe ever longer gaps – between 1 and 3, 4 and 9, 13 and 27, 40 and 81, etc. – and the upper endpoints of these gaps are successive powers of 3: 3, 9, 27, 81, 243, 729, 2187.

Also, the upper endpoint, U, of a gap is one greater than twice its lower endpoint, L, i.e. $U = 2L + 1$. This leads to an alternative algorithm for generating sequences. Beginning with any finite sequence having the required property:

$$a_1, a_2, ..., a_k \tag{2.4}$$

we can add not just the next term, but a block of k terms obtained by translating each term in Equ. (2.4) by $2a_k + 1$ units. The result is a sequence of $2k$ terms

$$a_1, a_2, ..., a_k, a_{k+1} = 2a_k + 1 + a_1, ..., a_{2k} = 2a_k + 1 + a_k \tag{2.5}$$

Exercises for Section 2.5.4

(1) a) Prove that the sequence of $2k$ terms in Equ. 2.5 inherits the desired property from the sequence of k terms in Equ. 2.4.

b) Beginning with the sequence 0, 1, write a program for this algorithm, and check whether it generates the same sequence as our greedy algorithm.

c) Prove that $a_{2^j+1} = 3^j$.

(2) a) Rewrite Listing 2.21 so that it prints the output in base 3, and note that all the numbers in the sequence only use the ternary digits 0 or 1, and that they all appear in increasing order, i.e. 0, 1, 10, 11, 100, 101, 110, ... up to $3^j = 10...0$ (j zeros).

b) Prove that the finite sequence generated by the greedy algorithm consists exactly of the numbers whose ternary digits are 0's and 1 's. (Hint: use induction.)

c) Prove that the sequence generated by the alternative algorithm starting with 0, 1 consists exactly of the numbers whose ternary digits are 0's and 1 's. (Note that b) and c) together prove that your observation in Ex. 1 b) persists for arbitrarily long sequences.)

To answer the question posed in the Olympiad problem, we need only verify that the first $2^{11} = 2048 > 1983$ terms of our sequence are all $\leq 10^5$. The largest of them is $a_{2^{11}}$, the lower endpoint of the gap with upper endpoint 3^{11}. Its value, because of the relation $U = 2L + 1$, or $3^{11} = 2a_{2^{11}} + 1$, is $a_{2^{11}} = (3^{11} - 1)/2 = 88573 < 10^5$. We conclude that our sequence of 2048 integers from 0 to 88573 contains no three in arithmetic progression.

2.5.5 Ulam's Sequence. At the 1963 number theory conference in Boulder, Colorado, Stanislaw Ulam proposed the following sequence $U : u_1 = 1, u_2 = 2, u_n =$ the smallest integer that can be uniquely represented as a sum of two different preceding terms.

This is now known as the standard Ulam sequence, or the (1, 2)-Ulam sequence, since the idea can be generalized to use different starting values (u, v).
There's an obvious, but slow, program for this sequence (see Ex. 2 below). There's also a much faster version which uses a "double sieve" to compute the elements in $U \leq n$. It uses two boolean lists, `lst` and `mst`. For $i > 2$:

- lst[i] = True iff i is representable as a sum in at least one way.
- mst[i] = True iff i is representable as a sum in at most one way.

Listing 2.22 (ulam107.py) implements the algorithm. Several exercises are based on it.

```
n = int(input("n=? "))
lst = [False]*(n+1)
mst = [True]*(n+1)
lst[1] = True; lst[2] = True
k = 1
while k < n:
  k += 1
  while not(lst[k] and mst[k])
      and (k < n):
    k += 1
  m = min(k-1,n-k)
  for i in range(1, m+1):
    mst[k+i] = mst[k+i] and \
      not(lst[k+i] and lst[i]
          and mst[i])
    lst[k+i] = lst[k+i] or (lst
        [i] and mst[i])
for i in range(1,n+1):
  if mst[i]:
    print(f"{i:4}", end='')
```

Listing 2.22. Ulam's sequence

The first few terms are:

```
> python ulam107.py
n=? 100
   1   2   3   4   6   8  11  13  16  18  23  25
  26  28  33  35  36  38  43  45  47  48  53  57
  62  67  69  72  77  82  87  92  94  96  97  99
```

which can be checked at `https://en.wikipedia.org/wiki/Ulam_number`. More information is available in Guy [**Guy04**], section C4. This algorithm is due to M.C. Wunderlich, "The Use of Bit and Byte Manipulation in Computing Summation Sequences", *BIT Numerical Mathematics*, 11 (1971), 217-224, `https://link.springer.com/article/10.1007/BF01934371`, and is discussed at length in `https://math.stackexchange.com/questions/2165222/generating-ulam-sequences-using-bit-manipulation`.

Exercises for Section 2.5

(1) In the central prison of Sikinia there are n cells numbered 1 to n, each occupied by a single prisoner. The state of each cell can be changed from closed

to open and vice versa by a half-turn of the key. To celebrate the Centennial Anniversary of the Republic it was decided to grant a partial amnesty. The president sent an officer to the prison with the instruction

```
for i in range(1,n):
  turn the keys of cells i, 2i, 3i, ....
```

A prisoner was freed if at the end his door was open. Which prisoners are set free? Remark: don't think, just sieve!

(2) a) Write the "obvious" program for the Ulam sequence and compare its speed with Listing 2.22.

The following should be answered using Listing 2.22.

b) In the interval 1..32000 find all solutions of $u_i + u_{i+1} = u_k$.

c) In 1..32000 find all pairs of consecutive numbers which are U-numbers.

d) Among the distances $g_i = u_{i+i} - u_i$, between successive U-numbers in the interval 1..32000, find the largest.

e) What proportion of U-numbers are twins with distance 2?

f) Find the frequency distribution of the distances g_i.

g) Does the asymptotic distribution of U-numbers parallel that of the distribution of primes?

h) What sequence do you get if the two preceding terms need not be different?

(3) Construct a sequence $0 < a_1 < a_2 < a_3 < ...$ of integers with the property that each non-negative integer n can be uniquely represented in the form $n = a_i + 2a_j$, where i and j need not be distinct. Collect numerical data, conjecture, prove. This sequence is due to L. Moser, "An Application of Generating Series", *Mathematics Magazine* 35 (1962), 37-38, `https://www.jstor.org/stable/i326617`.

(4) Consider the sequence of the first digit in

a) fib(n) for $n = 1$ to 10000,

b) 2^n for $n = 1$ to 10000,

c) 3^n for $n = 1$ to 10000.

Find the frequencies $x[1], ..., x[9]$ of the digits 1 to 9 in each of the sequences in a), b), and c). The result is surprising, but it can be shown that the frequency of the digit n is $\log_{10}(1 + 1/n)$ for $n = 1, ..., 9$.

(5) Prove that the density of square-free integers is $p = 6/\pi^2$. Use the code from Listing 2.1.5.

(6) **The Frobenius Problem**. Let $a_1, ..., a_k$ be natural numbers with $gcd(a_1, ..., a_k) = 1$. The natural number N can be represented by $a_1, ..., a_k$ if there are non-negative integers $x_1, ..., x_k$, such that $N = a_1x_1 + ... + a_kx_k$.

It's not hard to show that all sufficiently large numbers can be so represented. The Frobenius Problem is to find the largest number $g(a_1, ..., a_k)$ without such a representation – a problem that has a surprising large number of interpretations. For example, given coins with denominations $a_1, ..., a_k$, find the largest amount which cannot be paid with these coins. The wikipedia article on this topic at `https://en.wikipedia.org/wiki/Coin_problem` is a good source of references.

a) For $k = 2$ there is a simple solution.

b) For $k > 2$ only a few partial results are known. Write a program for $k = 3$, which collects experimental data.

c) Write an efficient program which for any input $a_1, ..., a_k$ finds $g(a_1, ..., a_k)$ (This is a difficult problem, requiring dynamic programming techniques for an efficient solution.)

(7) **The 3n + 1 Problem**. The following algorithm for generating a sequence is quite old, but we still don't know if it always stops:

(a) Start with a natural number n.
(b) If $n = 1$ then stop.
(c) If n is odd then set $n = 3n + 1$.
(d) If n is even then set $n = n /\!/ 2$ and jump to (b)

a) Write a program, which for input n prints the sequence $n, f(n), f(f(n)), ..., 1$, where

$$f(n) = \begin{cases} 3n+1 & \text{if } n \text{ is odd} \\ n /\!/ 2 & \text{if } n \text{ is even} \end{cases}$$

b) Write a program which for input n counts the number $t(n)$ of terms of the sequence.

c) Print a table $(i, t(i))$ for $i = a, a + 1, ..., b$.

d) For input n print the triple $(n, t(n), max(n))$, where $max(n)$ is the largest value in the sequence.

e) The algorithm can stop only if it encounters a power of 2. Write a function, which for input n prints the exponent of this power of 2.

f) Write a function which for i between a and b prints the largest $t(i)$ and the values of i for which it occurs.

This algorithm, better known as the Collatz conjecture, is one of the most famous unsolved problems in mathematics (`https://en.wikipedia.org/w`

iki/Collatz_conjecture), and is named after Lothar Collatz who posed it in 1937. Paul Erdös later remarked: "Mathematics may not be ready for such problems."

(8) Suppose you want to test, for all numbers from 2 to b, whether or not the $3n+1$ algorithm stops.

a) Show that you only need to test odd numbers.

b) Show that if you start with n you may stop as soon as you encounter a number smaller than n.

c) Show that you need to test only numbers of the form $4n-1$.

d) Show that you need not test numbers of the form $16k+3$ or $128k+7$.

(9) Let's again consider the $3n+1$ algorithm, this time focusing on the record-breaking numbers, i.e. those numbers that require more steps to reach 1 than any preceding number. Write a program which prints these numbers, the number of steps to reach 1 and the maximum value reached. A few are listed in Table 2.6.

record breaking number	steps	largest term
2	1	2
3	7	16
6	8	16
7	16	52
9	19	52
18	20	52
25	23	88
27	111	9232
54	112	9232
73	115	9232

Table 2.6. 3n+1 records

(10) **Fibonacci squares**. Are there squares among the Fibonacci numbers? Obviously $fib(1) = fib(2) = 1$ and $fib(12) = 144$ are squares, and also $fib(0) = 0$, but not everybody starts with 0. Are there more squares in the sequence? Our aim is to sieve out the non-squares among the Fibonacci numbers from 1 to 10000. We first set$x[i] = 1$ for i in range(1,10001). These are our potential squares.

The idea is to consider the sequence modulo a prime p. Now any square is also a square $\pmod p$. Equivalently, a non-square $\pmod p$ is not a square. So we sieve out the non-squares mod 3, 5, 7, 11, 13, ...

The Fibonacci sequence mod 3 is 1, 1, 2, 0, 2, 2, 1, 0, 1, 1, which has a period of 8. The only non-square mod 3 is 2. So we can sieve out all terms with indices $8n+3$, $8n+5$, $8n+6$ by setting the corresponding $x[i]=0$.

Now consider the sequence mod 5: 1, 1, 2, 3, 0, 3, 3, 1, 4, 0, 4, 4, 3, 2, 0, 2, 2, 4, 1, 0, 1, 1, ..., which has a period of 20. The non-squares mod 5 are 2 and 3. So we set $x[20n+r]=0$ for $r=3, 4, 6, 7, 13, 14, 16, 17$.

Continue in this way until all the elements except $x[1], x[2], x[12]$ are zero. A proof of this result can be found online at `https://www.ma.imperial.ac.uk/~buzzard/2017nt2.pdf`, based on J.H.E. Cohn, "Square Fibonacci Numbers", *Fibonacci Quart.* 2 (1964), 109-113.

(11) Explore the sequence

$$f(n)=\begin{cases} 3n-1 & \text{if } n \text{ is odd} \\ n /\!/ 2 & \text{if } n \text{ is even.} \end{cases}$$

The sequence always seems to enter one of four cycles. Find them and prove this for $n<10000$. Also check matters with some large values for n.

(12) **Conway's permutation sequences**. A sequence is defined by

$$f(n)=\begin{cases} \lfloor (3n+1)/4 \rfloor & \text{if } n \text{ is odd} \\ 3n /\!/ 2 & \text{if } n \text{ is even} \end{cases}$$

or, more lucidly, $2m \mapsto 3m$, $4m-1 \mapsto 3m-1$, and $4m+1 \mapsto 3m+1$, from which it is clear that the operation is invertible. So, if we connect n with $f(n)$, $n=1, 2, ...$, the resulting graph consists only of disjoint cycles and doubly infinite chains.

Explore some sequences. It is conjectured that there are only four cycles, but there's no conjecture about the number of infinite chains. What is the status of the sequence containing the number 8?

This exercise appears in Guy [**Guy04**], section E17.

(13) Proceeding as in Ex. 10, find by successive sieving the *triangular* Fibonacci numbers below 1000, i.e. Fibonacci numbers of the form $m(m+1)/2$.

2.6 Rotation of a List

How do you rotate a list v of n elements left by m positions? For instance, with $n=10$ and $m=3$ the list 0123456789 should be transformed into 3456789012. Can you do this with little extra storage, which makes sense if m and n are both large?

Of course, you could copy the first m elements into a second list $y[1..m]$, moving the remaining elements left m places and then copy the elements from

the temporary list back to v. But what if m is so large that there's not enough space for this approach?

We first consider an easy to understand and quite efficient solution. Rotation means switching two blocks $AB \mapsto BA$. Suppose we have a function reverse(), which reverses a block: $A \mapsto A^R$. Then we form $(A^R B^R)^R$ which becomes BA. For our example with $n = 10$, $m = 3$ we get

$$0123456789 \mapsto 2109876543 \mapsto 3456789012.$$

So we call reverse(a, b) (reverse the part of the list from the a-th to the $(b-1)$-th elements) three times:

```
reverse(0, m): 210 3456789
reverse(m, n): 210 9876543
reverse(0, n): 3456789 012.
```

Listing 2.23 (rotate.py) implements this idea.

```
def reverse(l,r):
  i = l; j = r-1
  while j-i > 0:
    v[i], v[j] = v[j], v[i]
    i += 1
    j -= 1

m,n = map(int, input("m n=? ").
    split())
v = [i for i in range(n)]
print("Original:", v)
reverse(0, m)
reverse(m, n)
reverse(0, n)
print(" Rotated:", v)
```

Listing 2.23. Rotation by reversals

Switching adjacent blocks by means of reversals is part of computing folklore; for example, see Gries [**Gri81**] and Bentley [**Ben00**].

There's a different algorithm which may be somewhat more efficient if $gcd(m, n) = 1$ or some small number. The **dolphin** algorithm consists of shifting list elements into their new places one by one, with the next move being to fill the space that was just vacated.

First we look at the case $m = 3$, $n = 10$, with $gcd(m, n) = 1$. We save a copy of $v[0]$ in x. Now we shift $v[3]$ to its final position $v[0]$ leaving $v[3]$ free to be occupied by $v[6]$, etc.:

$$x \leftarrow v[0] \leftarrow v[3] \leftarrow v[6] \leftarrow v[9] \leftarrow v[2] \leftarrow v[5] \leftarrow v[8] \leftarrow v[1] \leftarrow v[4] \leftarrow v[7] \leftarrow x.$$

Each move uses $v[i] = v[(i + 3)\%10]$ or $v[i] = v[(i + m)\%n]$ and since $gcd(m, n) = 1$ all the shifts can be achieved in a single pass which ends with the original x being copied into $v[7]$.

Consider another example. This time $m = 6$, $n = 15$ with $gcd(m, n) = 3$.

Each move uses $v[i] = v[(i+6)\%15]$ or $v[i] = v[(i+m)\%n]$ and since $gcd(m,n) = 3$ this will require three passes over the list:

$$x \leftarrow v[0] \leftarrow v[6] \leftarrow v[12] \leftarrow v[3] \leftarrow v[9] \leftarrow x$$
$$x \leftarrow v[1] \leftarrow v[7] \leftarrow v[13] \leftarrow v[4] \leftarrow v[10] \leftarrow x$$
$$x \leftarrow v[2] \leftarrow v[8] \leftarrow v[14] \leftarrow v[5] \leftarrow v[11] \leftarrow x.$$

The corresponding Python program is shown in Listing 2.24 (dolphin.py).

```
m,n = map(int, input("m n
    =? ").split())
v = [i for i in range(n)]
print("Original:", v)
d = gcd(m, n)
print("No. of passes:", d)
for i in range(d):
  j = i; x = v[i]
  while True:
    k = j; j = (j+m)%n
    v[k] = v[j]
    if j == i: # at start
      break
  v[k] = x
print(" Rotated:", v)
```

Listing 2.24. Rotation by shifting

2.7 Partitions

Let $p(n)$ be the number of different partitions of n into parts. For instance,

$$5 = 4+1 = 3+2 = 3+1+1 = 2+2+1 = 2+1+1+1 = 1+1+1+1+1.$$

Thus $p(5) = 7$. In addition let $f(m,n)$ be the number of partitions of m with each part at most n. Then $p(n) = f(n,n)$. $f(m,n)$ can be defined recursively as:

$$f(1,n) = f(m,1) = 1 \text{ for all } m,n; \tag{2.6}$$
$$f(m,n) = f(m,m) \text{ for } m \le n; \tag{2.7}$$
$$f(m,m) = 1 + f(m,m-1); \tag{2.8}$$
$$f(m,n) = f(m,n-1) + f(m-n,n) \text{ for } n < m. \tag{2.9}$$

We can combine Equs. (2.7) and (2.8) into

$$f(m,n) = 1 + f(m,m-1) \text{ for } m \le n.$$

Thus, we get the recursive $f()$ in Listing 2.25 (parts.py).

```
def f(m,n):
  if (m == 1) or (n == 1):
    return 1
  elif m <= n:
    return 1 + f(m,m-1)
  else:
    return f(m,n-1) + f(m-n,n)

n = int(input("n=? "))
print(f"p({n}) = {f(n,n)}")
```

Listing 2.25. Recursive partition counting

Note that the code assumes that $n = m$, and so the user only needs to supply a single value.

Typical output:

```
> python parts.py
n=? 20
p(20) = 627

> python parts.py
n=? 39
p(39) = 31185
```

The drawback of this approach is that the recursive computation time grows exponentially with n, and so even medium size inputs (e.g. $n = 100$) take too long to return.
After some deliberation we succeeded in developing a fast iterative algorithm.

The first step is to set $f(0,n) = f(0,0) = 1$. Then Equ. (2.9), $f(m,n) = f(m,n-1) + f(m-n,n)$, becomes valid for $n = m$ since the second term on the right-hand side becomes $f(m-m,n) = f(0,n) = 1$. This makes Equ. (2.9) become $f(m,n) = f(m,m-1)+1$, which is equivalent to the combined equation for (2.7) and (2.8) above.

$f(m,n)$ can now be written as just two equations:

$$f(0,n) = 1 \text{ for all } n;$$
$$f(m,n) = f(m,n-1) + f(m-n,n) \text{ for } n \le m.$$

A standard way to implement a two-argument function like $f(m,n)$ is to use a 2D list (e.g. $p[m][n]$). This has the advantage that once a particular $f(m,n)$ value has been calculated then it no longer needs to be reevaluated (i.e. memoization). Also, a list is usually easier to iterate over rather than employing recursion.

If we study how the 2D list is filled in this problem, it becomes clear that the $f(m,n-1)$ value in $f(m,n) = f(m,n-1) + f(m-n,n)$ is never needed again after this calculation, and so the $f(m,n)$ value could be stored in $p[m][n-1]$.

More drastically, there's actually no need to store the $f()$ values in a 2D list at all; a 1D list is enough. This means that $f(m,n) = f(m,n-1) + f(m-n,n)$ can be implemented as $p[m] = p[m] + p[m-n]$. Note, that this still requires us to have two loops, one to iterate over n, and the other for m.

The resulting code is in Listing 2.26 (partiter.py).

```
n = int(input("n=? "))
p = [1]*(n+1)   # so p[n] can store a value
for i in range(2,n+1):
  for j in range(i,n+1):
    p[j] += p[j-i]
print(f"p({n}) = {p[n]}")
```

Listing 2.26. Iterative partition counting

We again assume that $n = m$, and so the user only needs to supply a single value. It also means that both loops use $n + 1$.

Some results:

```
p(127) =   3913864295              p(159) =  97662728555
p(138) =  12292341831              p(160) = 107438159466
p(149) =  37027355200
```

Srinivasa Ramanujan proved the congruences

$$p(5m + 4) \equiv 0 \pmod 5,$$
$$p(7m + 5) \equiv 0 \pmod 7,$$
$$p(11m + 6) \equiv 0 \pmod{11}.$$

Of the above values of n, 159 is of the form $7m + 5$ and the others have the form $11m + 6$. So $p(159)$ should be divisible by 7 and the other values should be divisible by 11. This is indeed the case.

A good starting point for more details on the partition function and Ramanujan's involvement is `https://en.wikipedia.org/wiki/Partition_function_(number_theory)`.

2.8 The Money Changing Problem

In how many different ways can you sum 100 cents using 1 cent coins, nickels, dimes, quarters and half-dollars (worth 1, 5, 10, 25, and 50 cents, respectively)? Let's generalize slightly and ask: in how many ways can you make n cents with US coins? Other countries have other coins, so generalize again:

A country has coins with denominations $D = \{1, d_1, d_2, ..., d_{k-1}\}$ units.
In how many ways can you make a sum of n units ?

We make sure to set $d_0 = 1$ so we can sum any amount. Let $a(n, k)$ be the number of ways to sum n units using coins from D. Then we have:

$$\begin{aligned} a(n, 0) &= 1, \\ a(n, k) &= 0 \text{ for } n < 0, \\ a(n, k) &= a(n, k - 1) + a(n - d_k, k). \end{aligned}$$

In words, the last equation states that a sum n can either use the coin d_k or not. There are $a(n, k - 1)$ ways to make n without using d_k and $a(n - d_k, k)$ ways when using d_k.

$a(n, k)$ is implemented recursively in Listing 2.27 (change.py).

```
d = [1, 5, 10, 25, 50]   # d_0 = 1, ...

def a(n, k):
```

```
if n < 0:
  return 0
elif k == 0:  # d_0 = 1
  return 1
else:
  return a(n,k-1) + a(n-d[k],k)
```

Listing 2.27. Recursive change sum

For US coins we get the results shown in Table 2.7.

n	100	200	300	400	500
a(n,k)	292	2435	9590	26517	59576

Table 2.7. Ways of summing n using US coins

We've implemented a tree recursive process, which will become exponential slow as n increases (or crashes at Python's recursion limit). Let's construct an iterative version to deal with larger n. The direct translation of Listing 2.27 would use a 2D list $a[n][k]$, but a study of how the data is built shows a similar optimization is possible as in the previous section's partitioning problem. Instead of a 2D list, we can keep overwriting a 1D list. We get Listing 2.28 (change2.py).

```
d = [1, 5, 10, 25, 50] #d_0=1
n = int(input("n? "))
print("Coins:", d)
k = len(d)
a = [1]*(n+1)
for j in range(1, k):
  for i in range(d[j], n+1):
    a[i] += a[i-d[j]]
print("Ways to pay:", a[n])
```

Listing 2.28. Iterative change sum

Listing 2.28 computes the number of ways (432699251)to sum 50000 (i.e. $500) in a fraction of a second whereas Listing 2.27 raises an error when Python's recursion limit is reached.

2.9 An Abstractly Defined Set

A set S is defined by means of three axioms:

Al. $0 \in S$
A2. $x \in S \rightarrow 2x + 1 \in S$ and $3x + 1 \in S$
A3. S is the minimal set satisfying axioms A1 and A2.

We'll use these as the basis of a recursive boolean function which recognizes if a number n belongs to S. Which of the numbers 511, 994, 995, 996, 997, 998,

999 do belong? The trees in Fig. 2.9 show that only 511, 994, and 999 are in S, since these numbers construct a path leading to 0 (and T for True).

Note that this search is the 'opposite' of the mathematical definition. The axioms builds a set starting from $x = 0$ by adding $2x + 1$ and $3x + 1$ recursively, while the search starts with the final number and attempts to reduce it to 0.

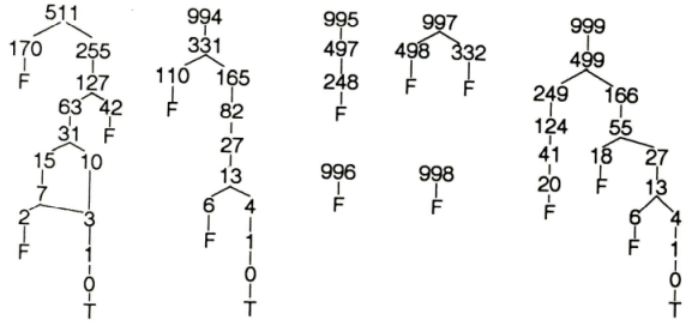


Figure 2.9. Decide if a number belongs in S

inS() in Listing 2.29 (inset.py) uses recursion to implement the tree search shown in Fig. 2.9

```
def inS(x):
  if x == 0:
    return True
  x -= 1
  if (x % 2 != 0) and (x % 3 != 0):
    return False      # nothing divisible
  elif x % 3 != 0:    # one is dvisible
    return inS(x//2)
  elif x % 2 != 0:
    return inS(x//3)
  else:               # both are divisible
    return inS(x//2) or inS(x//3)
```

Listing 2.29. Recursively check if a number is in S

In this case, there's little chance of Python's recursion limit being hit since the number of calls is proportional to a tree's longest branch, which is $O(\log_2 n)$ when it's checking $2n + 1$ reductions.

The iterative version in Listing 2.30 (insetIt.py) is much closer to the axiomatic definition.

```
n = int(input("n=? "))
s = {0}
for i in range((n+1)//2):
  if i in s:
    s.add(2*i+1)
    s.add(3*i+1)
print("in S:", n in s)
```

Listing 2.30. Iteratively check if a number is in S

It builds a Python set just big enough so that it can check if the user-supplied number is a member. The trick is to stop adding to the set once $i = (n+1) /\!/ 2$ which ensures that n will be in the set if it's valid.
This approach illustrates the trade-off between recursion and data structures. This example replaces recursive calls by set building.

2.10 Recursive Pile Splitting

A single pile of n chips, $n > 1$, is randomly split into two piles of k and $n-k$ chips, and the product $k(n-k)$ is computed. Then each pile, provided it has more than one chip, is randomly split again, and the product of the number of chips in its parts is added to the preceding product. This splitting is repeated on all the piles until every one consists of just a single chip (i.e. there are n piles).
A recursive function to find the final sum of the $n-1$ products is given in Listing 2.31 (sumprod.py).

```
def f(n):
  if n < 3:
    return n-1
  else:
    k = random.randint(1,n-1)
    return k*(n-k) + f(k) + f(n
        -k)
```

Listing 2.31. Recursive piles splitting

Table 2.8 shows the final sum when the pile starts with different sizes. Surprisingly, the sum doesn't depend on how the piles are split.

n	2	3	4	5	6	7	8	9	10
f(n)	1	3	6	10	15	21	28	36	45

Table 2.8. Sum of $n-1$ splits

Based on an examination of Table 2.8, we guess that $f(n) = \binom{n}{2}$. For a proof, imagine tying strings from each chip to every other chip. Initially there will be $n(n-1)/2$ strings. After splitting a pile into piles X, Y with x and y chips, we cut all the strings connecting those two piles, which reduces the total by xy strings. This will continue until there are n piles of chips with all the strings cut. This will always total $n(n-1)/2$, which is also the total sum of the xy products, no matter how we split the piles.

2.11 The Stern-Brocot Sequence

The online reference for this sequence is `https://oeis.org/A002487`. (It was formerly labeled N0056 in the 1973 edition of Sloane's *Handbook of Integer Sequences*.)

We define a function $f()$ on the non-negative integers by means of the recursion

$$f(0) = 0, \quad f(1) = 1, \quad f(2n) = f(n), \text{ and}$$
$$f(2n+1) = f(n) + f(n+1) \text{ for all } n > 0.$$

The Stern-Brocot sequence (or Stern's diatomic series) are the values $f(0), f(1), f(2), ...$ Incidentally, this function is also known by the name fusc(), given to it by Edgser Dijkstra in the 1970s. Let

$$n = 2^s + e_{s-1}2^{s-1} + ... + e_0$$

be the binary representation of n. If we reverse its digits (e.g. 10100 $\mapsto$ 00101), we get

$$u = e_0 2^s + e_1 2^{s-1} + ... + e_{s_1} 2 + 1.$$

It's fairly difficult to prove that $f(n) = f(u)$ for all n, but quite easy to write a program that calculates $f(n)$ and $f(u)$ and checks for equality.

$f()$ is tree recursive but the depth of the tree is only about $O(\log_2 n)$ because the recursive calls are dividing n by 2. Nevertheless, an iterative version will be faster for very large numbers (reverse1.py).

```
def f(n):
  if n < 2:
    return n
  elif (n % 2 == 0):  # even
    return f(n//2)
  else:
    return f((n-1)//2) + f((n
        +1)//2)

n = int(input("n=? "))
fn = f(n)
print(f"n: {n}; f({n}): {fn}")
u = 0
while n > 0:
  u = u*2 + n%2
  n = n//2
fu = f(u)
print(f"u: {u}; f({u}): {fu}")
print("Equal?", fn == fu)
```

Listing 2.32. Reversing using f()

To better understand the transformation, let's hand-compute one call to $f()$ in detail:

$$\begin{aligned} f(85) &= f(42) + f(43) &&= f(21) + f(21) + f(22) \\ &= 2 * f(21) + f(22) &&= 2 * f(10) + 3 * f(11) \\ &= 5 * f(5) + 3 * f(6) &&= 5 * f(2) + 8 * f(3) \\ &= 13 * f(1) + 8 * f(3) &&= 13 * f(0) + 21 * f(1) = 21. \end{aligned}$$

The function $g()$ will be useful:

$$g(n,i,j) = i * f(n) + j * f(n+1). \tag{2.10}$$

Then we have

$$\begin{aligned} g(2n,i,j) &= i * f(2n) + j * f(2n+1) \\ &= (i+j) * f(n) + j * f(n+1) = g(n,i+j,j), \\ g(2n+1,i,j) &= i * f(2n+1) + j * f(2n+2) \\ &= i * f(n) + (i+j) * f(n+1) = g(n,i,i+j). \end{aligned}$$

This means that $g()$ can be defined using two recursions:

$$\begin{aligned} g(2n,i,j) &= g(n,i+j,j) \\ g(2n+1,i,j) &= g(n,i,i+j). \end{aligned}$$

Both of these reduce the n value, so eventually we'll reach $g(0,i,j)$. The definition for $g()$ in Equ. (2.10) means that:

$$\begin{aligned} g(n,i,j) &= i * f(n) + j * f(n+1) \\ g(0,i,j) &= i * f(0) + j * f(1) = j. \end{aligned}$$

Equ. 2.10 can also be used to express $g()$ in terms of a single $f()$ call, by setting $i = 1$, and $j = 0$:

$$\begin{aligned} g(n,i,j) &= i * f(n) + j * f(n+1) \\ g(n,1,0) &= 1 * f(n) + 0 * f(n-1) \\ &= f(n). \end{aligned}$$

This lets us replace $f()$ by $g()$ which is still recursive, but tail-recursive. The resulting code is in Listing 2.33 (reverse2.py). Another step is to translate the tail-recursion into a loop, as in Listing 2.34 (reverse3.py).

```
def g(n,i,j):
  if n == 0: return j
  elif n/2 != int(n/2):
    return g((n-1)/2, i, i+j)
  else:
    return g(n/2, i+j, j)

n = int(input("n=? "))
gn = g(n,1,0); u = 0
print(f"n: {n}; g({n},1,0): {gn}")
while n > 0:
  u = u*2 + n%2; n = n//2
gu = g(u,1,0)
print(f"u: {u}; g({u},1,0): {gu}")
```

Listing 2.33. Reversing using $g()$

```
def g(n,i,j):
  while n > 0:
    if n/2 != int(n/2):
      n, i, j = (n-1)/2, i, i+j
    else:
      n, i, j = n/2, i+j, j
  return j
```

Listing 2.34. Iterative version of $g()$

2.11.0.1 The Calkin–Wilf Tree. The $f(0), f(1), f(2), ...$ values of the Stern-Brocot sequence have the unexpected feature that $f(n)/f(n+1)$ runs through all the reduced non-negative rationals exactly once.

The start of the sequence is printed out by sbSeq.py which repeatedly calls $f()$, and uses Python's fraction class to print $f(n)/f(n+1)$

```
> python sbSeq.py
n=? 40
  0  1  1  2  1  3  2  3  1  4  3  5  2  5  3  4  1  5  4  7
  3  8  5  7  2  7  5  8  3  7  4  5  1  6  5  9  4 11  7 10
0
1
1/2 2
1/3 3/2 2/3 3
1/4 4/3 3/5 5/2 2/5 5/3 3/4 4
1/5 5/4 4/7 7/3 3/8 8/5 5/7 7/2 2/7 7/5 5/8 8/3 3/7 7/4 4/5 5
1/6 6/5 5/9 9/4 4/11 11/7 7/10
```

These rationals can be arranged into an infinite complete binary tree called a Calkin–Wilf tree tree, as in Fig. 2.10.

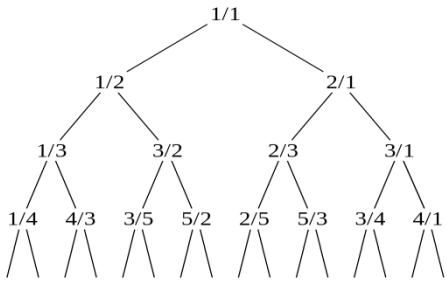


Figure 2.10. The first few levels of the Calkin-Wilf tree

The tree is rooted at the number 1, and any rational number expressed in simplest terms as the fraction a/b has as its two children the numbers $a/(a+b)$ and $(a+b)/b$. It is named after Neil Calkin and Herbert Wilf, who considered it in 'Recounting the Rationals', in the *The American Mathematical Monthly*, Vol. 107, No. 4, April 2000, online at `https://www.maths.ed.ac.uk/~v1ranick/papers/calkinwilf.pdf`.

The sequence of rationals obtained by a breadth-first traversal of the tree is known as the Calkin–Wilf sequence. Listing 2.35 (calkinwilf.py) prints out the requested n-th number in the sequence.

```
def nCalkinWilf(n):
  queue =[(1,1)]
  n -= 1
```

```
  for _ in range(n):
    numer, denom = queue.pop(0)
    leftNumer = numer; leftDenom = numer + denom
    rightNumer = numer + denom; rightDenom = denom
    queue.append((leftNumer, leftDenom))
    queue.append((rightNumer, rightDenom))
  finalNumer, finalDenom = queue.pop(0)
  if finalDenom == 1:
    return finalNumer
  else:
    return str(finalNumer) + '/' + str(finalDenom)
```

Listing 2.35. Generating the n-th value in the Calkin-Wilf sequence

For example:

```
> python calkinwilf.py
n? 5
3/2
> python calkinwilf.py
n? 12
2/5
```

The code doesn't build a tree, instead using a queue to implement a breadth-first traversal over the sequence.

2.11.0.2 Relationship to the Stern–Brocot tree. The Calkin–Wilf tree resembles the Stern–Brocot tree (see Fig. 2.11), especially at the top levels, and the same numbers appear at the same levels in both trees.

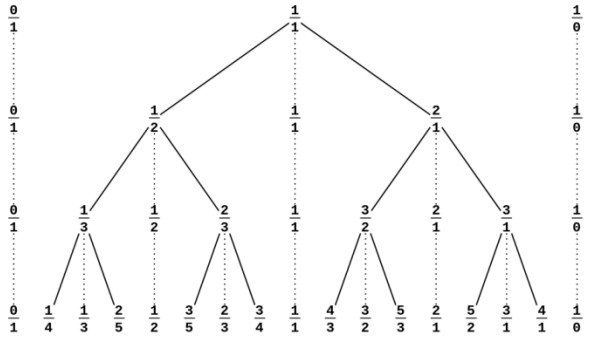


Figure 2.11. The first few levels of the Stern-Brocot tree

A crucial (and useful) difference is that Stern–Brocot is a binary search tree: its left-to-right traversal is the same as the numerical order of the numbers in it. This property is not true of Calkin–Wilf.

The tree's construction begins with the two fractions $\frac{0}{1}$, $\frac{1}{0}$ although the second isn't strictly a fraction, but think of it as representing ∞. At every subsequent level, the existing fractions are copied down (that's the meaning of the dotted lines in Fig. 2.11), and the *mediant* of adjacent fractions are inserted. For two fractions $\frac{a}{b}$ and $\frac{c}{d}$ their mediant is $\frac{a+c}{b+d}$.

Every fraction in the tree has two children. Each child is the mediant of the closest ancestor on the left and closest ancestor on the right.

One approach for building a Stern-Brocot tree is to convert an existing Calkin-Wilf tree by performing a bit-reversal permutation on the numbers at each level of the trees. However, an easier approach is to maintain a list for the current level of the Stern-Brocot tree, and keep expanding it with mediants until the desired level is reached. That is implemented in Listing 2.36 (sbTree.py). Typical output:

```
> python sbTree.py
lvls=? 3
(0, 1) (1, 3) (1, 2) (2, 3)
(1, 1) (3, 2) (2, 1) (3, 1)
(1, 0)
```

```
def expand(fracs):
  nFracs = []; i = 0
  while i < len(fracs)-1:
    nFracs.append( fracs[i])
    nFracs.append( mediant(
        fracs[i], fracs[i+1]))
    i += 1
  nFracs.append( fracs[-1])
  return nFracs

def mediant(f1, f2):
  return (f1[0] + f2[0],
          f1[1] + f2[1])

lvls = int(input("lvls=? "))
fracs = [ (0, 1), (1, 0) ]
for i in range(lvls):
  fracs = expand(fracs)
print(*fracs)
```

Listing 2.36. A Stern–Brocot tree level

An important aspect of the tree is that the mediant of two fractions is always a value between those fractions:

$$\frac{a}{b} \leq \frac{a+c}{b+d} \leq \frac{c}{d}$$

given that

$$\frac{a}{b} \leq \frac{c}{d}.$$

The two inequalities can be shown by rewriting the fractions with common denominators.

This means that the tree can be used to find increasingly accurate rational approximations to decimals (see Listing 2.37; RationalApprox.py).

```
def mediant(f1, f2):
  return (f1[0] + f2[0],
          f1[1] + f2[1])

x = float(input("x=? "))
left  = (0, 1); right = (1, 0)
best = left
bestError = abs(x)
while bestError > EPS:
  med = mediant(left, right)
  if x < toFloat(med):
    right = med     # go left
  else:
    left = med      # go right
  error = abs(toFloat(med) - x)
  if error < bestError:
    best = med; bestError = error
    print(best, end=' ')
print()
```

Listing 2.37. Finding rational approximations

Approximations for π and e:

```
> python RationalApprox.py
x=? 3.14159265358979323846264338328
(1, 1) (2, 1) (3, 1) (13, 4)
(16, 5) (19, 6) (22, 7) (179, 57)
(201, 64) (223, 71) (245, 78)
(267, 85) (289, 92) (311, 99)
(333, 106) (355, 113)

> python RationalApprox.py
x=? 2.71828182845904523536028747135
(1, 1) (2, 1) (3, 1) (5, 2)
(8, 3) (11, 4) (19, 7) (49, 18)
(68, 25) (87, 32) (106, 39) (193, 71)
(685, 252) (878, 323) (1071, 394)
```

For more on the Stern–Brocot tree see Section 4.5 of Graham [**GKP94**] and `https://en.wikipedia.org/wiki/Stern%E2%80%93Brocot_tree`.

2.12 The Best Rational Approximation

Given a real number $r > 0$, we want to find a sequence of ever better rational approximations p/q with $q \leq qmax$. Listing 2.38 (ratap.py) is probably the simplest implementation.
We measure the distance between r and p/q with $d = |r - p/q|$, with $d = r$ initially. Each time that we get a smaller distance, the associated fraction p/q is printed.

```
r,qmax = map(float, input("r
    qmax=? ").split())
p = 0; q = 1
min = r
while q < qmax:
  if p/q < r:
    p += 1
  else:
    q += 1
  d = abs(r - p/q)
  if d < min:
    min = d
    print(p, "/", q)
  if (d == 0):
    break
```

Listing 2.38. Best rational approxs

If Listing 2.38 is run with r set to one of $\tau = (1+\sqrt{5})/2 \approx 1.6180339887498948432$, $\sqrt{2} \approx 1.4142135623730955049$, or $\pi \approx 3.1415926535897932 38$, we get the approximations listed in Table 2.9.

τ	$\sqrt{2}$	π
1/1	1/1	1/1
2/1	3/2	2/1
3/2	4/3	3/1
5/3	7/5	13/4
8/5	17/12	16/5
13/8	24/17	19/6
21/13	41/29	22/7
34/21	99/70	179/57
55/34	140/99	201/64
89/55	239/169	223/71
144/89	577/408	245/78
233/144	816/577	267/85
377/233	1393/985	289/92
610/377	3363/2375	311/99
987/610	4756/3363	333/106
1597/987	8119/5741	355/113
2584/1597	19601/13860	52163/16604
4181/2584	27720/19601	52518/16717

Table 2.9. Rational approximations for $\tau, \sqrt{2}, \pi$.

There are some interesting patterns hidden inside these rationals. For τ we get pairs of successive Fibonacci numbers. For $\sqrt{2}$ the p's and q's are the solutions to $p^2 - 2q^2 = \pm 1$ and $p^2 - 2q^2 = 2$. The π approximations include the well-known 22/7 and 355/113. More importantly, the sequence of approximations is exactly the same as that produced by Listing 2.37. Is this also the case case for the sequences generated by Listing 2.37 when given values for τ and $\sqrt{2}$? The answer appears to be 'yes', but why?

We'll return to this problem when we look at continued fractions in Sec. 11.

Exercises for Sections 2.6 to 2.12

(1) Here's yet another method for rotating an array. Initially we have $V = AB$, which must be transformed into $V = BA$. Suppose that the block A is shorter than B, then we can write $B = B_0B_1$ with block B_1 being the same length as A. Blocks of equal length are easy to swap and we get $V = B_1B_0A$. Now A is in its final position and we must transform B_1B_0 into B_0B_1, but this is the same task as before but with shorter blocks B_1, B_0.

The opposite case where block B is shorter than A is treated in a similar fashion.

Write a program based on this idea.

(2) Let $b(0) = 1$ and for $n \geq 1$ let $b(n)$ denote the number of ways of writing n as a sum of powers of 2 (order does not matter). Thus

$$7 = 4+2+1 = 4+1+1+1 = 2+2+2+1 = 2+2+1+1+1$$
$$= 2+1+1+1+1+1 = 1+1+1+1+1+1+1$$

and so $b(7) = 6$. Show that

$$b(2n+1) = b(2n), \qquad b(2n) = b(2n-2) + b(n).$$

a) Write a recursive function to compute $b(n)$.

b) Write an iterative function to compute $b(n)$. For example:

$$b(132) = 31196.$$

(3) Let $r(n, m)$ be the number of partitions of n into parts, the smallest being at least m. We define $r(0, m) = 1$. Obviously $p(n) = r(n, 1)$, with $p(n)$ defined as in Sec. 2.7.

a) Show that $r(n, m) = r(n, m+1) + r(n-m, m)$

b) Write a recursive function which computes $p(n)$.

c) Write an iterative function which computes $p(n)$.

(4) Let the number of partitions of n into parts $\leq m$ be denoted by $p(n, m)$. Prove that $p(n, m) = p(n, m-1) + p(n-m, m)$ (Euler). Find boundary conditions and write a recursive function which computes p. Compare its speed with the one in Sec. 2.7.

(5) Let $p(u, n, m)$ be the number of partitions of u into n parts each $\leq m$. Show that $p(u+1, n+1, m) = p(u, n+1, m) + p(u, n, m) - p(u-m, n, m)$ (Euler). Find boundary conditions and write a function to compute p.

(6) Prior to 2002, German coins consisted of D = {1, 2, 5, 10, 50, 100, 200, 500} Pfennigs. In how many ways could you change

a) 1 DM into smaller coins? (1 DM = 1 German Mark = 100 Pfennigs.)

b) 5 DM using all the coins at least once?

(7) Russian coins consist of D = {1, 2, 3, 5, 10, 15, 20} kopeks. In how many ways can you change 1 Ruble (= 100 kopeks) into coins?

(8) Swiss coins less than 1 SF (Swiss Franc) consists of D = {1, 2, 5, 10, 20, 50} Rappen. In how many ways can you change 1 SF into smaller coins? (1 SF = 100 Rappen.)

(9) In how many ways can you change 1 English Pound into coins? A few years ago English coins were D = {0.5, 1, 2, 5, 10, 20, 50, 100} pence. How do you handle 0.5 pence? (The half-penny is no longer in circulation, and the Pound is 100 pence.)

(10) You have a set $a_1 < a_2 < ... < a_s$, of integer weights. In how many different ways can the weight n be composed from these weights? That is, write a program which finds the number C_n of solutions of

$$a_1e_1 + a_2e_2 + \cdots + a_se_s = n, \quad e_i \in \{0, 1\}.$$

(11) Solve the preceding problem if the weights can be placed on *both* pans of the scales. That is, write a program which finds the number D_n of solutions of

$$a_1e_1 + a_2e_2 + \cdots + a_se_s = n, \quad e_i \in \{-1, 0, 1\}.$$

In Exs. 10 and 11, experiment with different sets of weights, especially the set $\{1, 2, 2^2, ..., 2^{s-1}\}$ in 10, and the set $\{1, 3, 3^2, ..., 3^{s-1}\}$ in 11.

(12) A set S is defined by means of three axioms:

A1. $1 \in S$
A2. $x \in S \rightarrow 2x \in S, \quad 3x \in S, \quad 5x \in S.$
A3. S is the minimal set satisfying axioms A1 and A2.

a) Write a boolean function, which recognizes if $n \in S$.

b) Show that S contains all elements of the form $2^a3^b5^c$ with natural numbers a, b, c.

c) Write a program which prints the elements of S in order.

(13) There is a beautiful geometrical solution for the result in Sec. 2.10. Find out about it. As a hint, here's the accompanying 'Proof by picture' which illustrates a proof by induction.

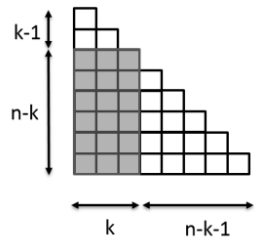


Figure 2.12. Dividing a pile of n chips

(14) The set S is the minimal set satisfying the two axioms:

A1. $0 \in S$
A2. $x \in S \rightarrow 2x + 1 \in S, \quad 3x + 1 \in S, \quad 5x + 1 \in S.$

Write recursive and iterative programs which print and count all elements of $S \leq n$.

(15) Does the set {1, 2, ..., 3000} contain a subset A of 2000 elements such that $x \in A$ implies $2x \notin A$? (Posed at the 10th Austrian-Polish Mathematics Competition (APMC) 1987; available at `https://imomath.com/index.cgi?page=collectionApo` and Kuczma [**Kuc94**].)

This problem was generalized by E.T.H. Wang in "On double-free sets of integers", *Ars Combinatoria*, 28, 1989, pp. 97-100 in the following way: a set S of integers is called double-free (D.F.) if $x \in S \to 2x \notin S$. Let $N_n = \{1,2, ...,n\}$ and $f(n) = \max\{|A| : A \subset N_n \text{ is D.F.}\}$.

Wang proved that $f(n) = \lceil n/2 \rceil + f(\lfloor n/4 \rfloor)$. Write a recursive program for $f()$ and solve the original problem.

Problems 16 to 19 refer to Sec. 2.11.

(16) Write a program which finds the three smallest n such that $f(n) = x$ for $x =$ 8, 10, 20, 30.

(17) Write an iterative program for $f(n)$. Hint: suppose we have values a, b such that $f(n) = af(m) + bf(m+1)$. Then we can find a similar formula $f(n) = a'f(m') + b'f(m'+1)$ with $m' = \lfloor m/2 \rfloor$ as follows. If m is even then $a' = a + b$, $b' = b$; if m is odd then $a' = a$, $b' = a+b$. We start with $m = n, a = 1, b = 0$ and when we reach $m = 0$ we have $f(n) = b$.

(18) A function f defined on the positive integers is given by

$$f(1) = 1, \quad f(3) = 3, \quad f(2n) = f(n),$$
$$f(4n+1) = 2f(2n+1) - f(n), \quad f(4n+3) = 3f(2n+1) - 2f(n)$$

Determine the number of positive integers ≤ 1988 for which $f(n) = n$. This was a difficult problem in the International Mathematical Olympiad XXIX (`https://www.imo-official.org/problems.aspx`), but is a fairly routine coding task.

a) Write a recursive function which computes $f(n)$.

b) Write a program which solves the problem by brute force.

c) Compute a table of $f(n)$ and guess what the function does.

d) Write a program which for input n finds the binary representations of n and $f(n)$. What does the function do?

e) Find the number of positive integers $\leq 2^k$ for which $f(n) = n$.

f) Prove that f reverses the binary representation of an odd integer.

g) Iteratively compute the array $f[k]$ for every $k \in 1..n$.

(19) Show that in Ex. 17 $f(n) = f(u)$. Hint: analyze Ex. 17 or 18f).

(20) a) Using Listing 2.29 find all set elements ≤ 10000.

b) What proportion of consecutive numbers have differences of 1?

c) What proportion of consecutive numbers have differences of 2?.

d) Find the number of elements in 1..n for $n = 1000, 2000, ..., 10000$. Observe the strange fluctuations in these ten "millenia". Does this behavior persist?

(21) In Sec. 2.6 we learned how to swap the blocks AB into BA. How would you transform ABC into CBA? That is, two nonadjacent blocks A, C must be swapped.

An efficient algorithm can be found in J .L Mohammed and C.S. Subi, "An Improved Block-Interchange Algorithm", *Journal of Algorithms* 8 (1987), 113-121.

(22) **The Frobenius Problem revisited**. Given coins with denominations $a[1], ..., a[s]$. Let $b[k]$ be the number of solutions of the equation $a[1]x_1 + ... + a[s]x_s = k$, where x_i are non-negative integers. Write a program which stores $b[1], ..., b[n]$ in a list b.

(23) Repeat Ex. 16 from "Exercises for Sections 1.1 to 1.7" in a similar way to the preceding problem.

(24) Run Listing 2.38 with r set to:

a) $\sqrt{3} \approx l.7320508076$

b) $\sqrt{7} \approx 2.2360679775$

c) $\sqrt{13} \approx 2.6457513110$

d) $\sqrt{5} \approx 3.1622776602$

e) $\sqrt{10} \approx 3.6055512754$

f) $e \approx 2.7182818285$

For each of these, also generate the sequence of approximations using Listing 2.37 to check if the two sequences are the same.

2.13 Primes

In this section we will deal with primes and their distribution.

2.13.1 How to Recognize a Prime. A natural number is a prime if it has exactly two divisors. If $n > 1$ is not a prime it can be decomposed into non-trivial factors

$$n = d_1 d_2, \quad 1 < d_1 < n, \quad 1 < d_2 < n.$$

The divisors d_1,d_2 cannot both be $> \sqrt{n}$, otherwise $d_1 d_2 >$ n. That is, a composite number n has non-trivial divisors $d \leq \sqrt{n}$. In other words:

If n has no divisors d in $1 < d \leq \sqrt{n}$, then n is a prime.

In Listing 2.39 (prime.py), isPrime() returns True or False depending on whether the supplied value is prime.

```
def isPrime(n):
  if n < 2:
    return False
  if n == 2 or n == 3:
    return True
  if n%2 == 0:
    return False
  for i in range(3, int(n**0.5)+1,
       2):
    if n%i == 0:
      return False
  return True
```

Listing 2.39. Test for primality

```
>>> from prime import *
>>> isPrime(17)
True
>>> isPrime(15)
False
```

prime.py contains two other useful prime functions: sievePrimes() and primes() (see Listing 2.40; prime.py).

```
def sievePrimes(n):
  if n < 2:
    return False
  prims = [False]*(n+1)
  for i in range(2,n+1):
    prims[i] = True
  for p in range(2,n+1):
    if (p*p <= n) and (prims[p]):
      for i in range(p*2, n+1, p):
        prims[i] = False
        p += 1
  return prims

def primes(n):
  prims = sievePrimes(n)
  return [i for i in range(len(
      prims)) if prims[i] ]
```

Listing 2.40. More tests for primality

sievePrimes() returns a list of booleans which indicate if the associated list index position is a prime. primes() converts this list into one containing only prime numbers. These functions can be called like so:

```
>>> sievePrimes(100)
[False, False, True, True,
... # many more
... True, False, False, False]
>>> primes(100)
[2, 3, 5, 7, 11, 13, 17, 19,
23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79,
83, 89, 97]
```

2.13.2 The Prime Number Theorem. The *Prime Number Theorem* (PNT) formalizes the notion that primes become less common as the integers increase. The theorem was proved independently by Jacques Hadamard and Charles Jean de la Vallé Poussin in 1896 using ideas introduced by Bernhard Riemann. The

basic distribution is:

$$\pi(x) \sim x/\ln x$$

where $\pi()$ is a prime-counting function (the number of primes less than or equal to x). It states that for large enough x, the *probability* that a random integer not greater than x is prime is very close to $1/\ln x$.

countPlot.py collects counts of primes in groups of 10000 numbers between 2 and MAX. The count/$(x/\ln x)$ values are plotted, where x is the last number in each group. As you might expect, the curve in Fig. 2.13 slowly approaches 1 as x gets larger.

More details on the PNT, including a sketch of its proof can be found at `https://en.wikipedia.org/wiki/Prime_number_theorem`.

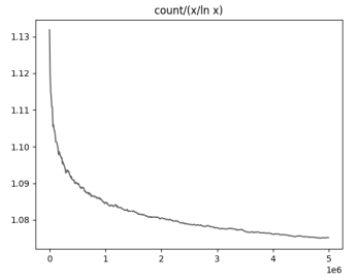


Figure 2.13. count/$(x/\ln x)$

2.13.3 The Goldbach Conjecture. The Goldbach Conjecture is a deep number theoretic problem that is still unsolved after more than 250 years. It states that every even number, starting with 6, can be represented as the sum of two odd primes.

Listing 2.41 (goldbach.py) is supplied with an even number range (e.g. 200 to 228) and reports the number of ways each of those even numbers can be represented by the sum of two odd primes.

```
def goldbackFull(g):
  x = 3; count = 0
  while x <= g-x:
    if isPrime(x):
      if isPrime(g-x):
        print(f"{x}+{g-x}",end='')
        count += 1
    x += 2
  print()
  return count
```

Listing 2.41. Goldbach counts

For example:

```
> python goldbach.py
a b? 20 24
----- 20 -----
  3+17  7+13
 Count: 2; Expect: 1.1
----- 22 -----
  3+19  5+17  11+11
 Count: 3; Expect: 1.2
----- 24 -----
  5+19  7+17  11+13
 Count: 3; Expect: 1.2
```

- Which numbers have particularly low counts?
- Which numbers seem to have particularly high counts?
- Do you see any patterns?

From around 36 on, numbers divisible by 3 seem to have a high count. Powers of 2 and numbers $2p$ with p a large prime seem to have low counts. Divisibility by small odd primes seems to raise the Goldbach count on the average. Foe example, with $n = 3 * 5 * 7 = 105$, $2n = 210$ should have a high count. Indeed:

x	200	202	204	206	208	210	212	214	216	218	220	222	224	226	228
count	8	9	14	7	7	19	6	8	13	7	9	11	7	7	12

Table 2.10. Goldbach counts for x values

The reason why multiples of 3, say, have a higher than average Goldbach count is as follows. About half of all the primes up to any given limit are ≡ 1 mod 3 and the other half are ≡ -1 mod 3. This is called the *Prime Number Theorem for Arithmetic Progressions*. Consequently, about half the sums $p + q$ of two primes are multiples of 3 while the other two residue classes mod 3 each have only about a quarter of these numbers.

A great deal of research has gone into developing probabilistic arguments in favor of the conjecture. One simplistic approach, based on the PNT, argues that an integer m selected at random has roughly a $\frac{1}{\ln m}$ chance of being prime. Thus if n is a large even integer and m is a number between 3 and $n/2$, then one might expect the probability of m and $n - m$ simultaneously being prime to be $\frac{1}{\ln(m)\ln(n-m)}$. Therefore, you might conclude that the total number of ways to write a large even integer n as the sum of two odd primes is roughly

$$\sum_{m=3}^{n/2} \frac{1}{\ln m} \frac{1}{\ln(n-m)} \approx \frac{n}{2(\ln n)^2}$$

Since $\ln n \ll \sqrt{n}$ goes to infinity as n increases, this suggests that every large even integer can be represented as the sum of two primes in many ways.

To test this heuristic, Listing 2.41 calculates $\frac{n}{2(\ln n)^2}$, and reports it as the 'Expect:' value. After some experimentation, it's clear that this approximation is somewhat inaccurate. The main problem is that it assumes that the probabilities of m and $n - m$ being prime are statistically independent, which isn't the case since they must both be odd numbers.

G.H. Hardy and J.E. Littlewood, developed much better probabilistic arguments in a paper published in *Acta Mathematica* in 1923:

$$P(2n) \sim C \left(\prod_{\substack{\text{odd prime} \\ \text{factors of n}}} \frac{p-1}{p-2} \right) \frac{2n}{(\ln 2n)^2}$$

where

$$C = 2 \prod_{\text{odd primes}} \left(1 - \frac{1}{(p-1)^2}\right) \approx 1.320324.$$

Actual Goldbach counts tend to differ from these approximations by several percent even for five-digit numbers.

goldbachComet.py plots Goldbach's comet, which reveals some interesting aspects of the counts. The graph is shown (in grayscale) in Fig. 2.14. A color version is online at colorComet.png.

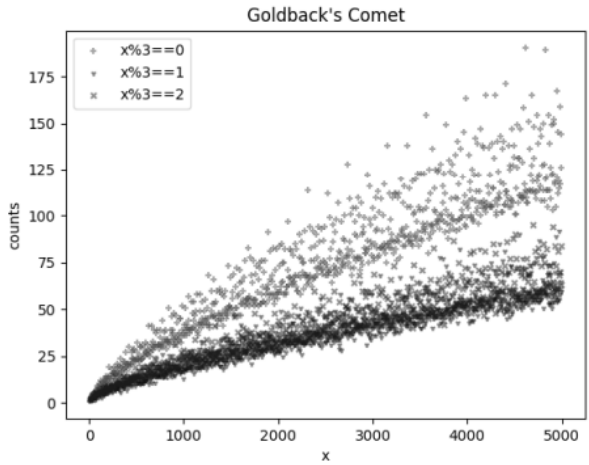


Figure 2.14. Goldbach's comet

The coloring and shape of the points is based on the value of $x/2\%3$. Red + points corresponding to 0 mod 3, black ∨ points corresponding to 1 mod 3, and blue × points for 2 mod 3. In other words, the red points are multiples of 6, the green points are multiples of 6 plus 2, and the blue points are multiples of 6 plus 4.

The graph suggests that there are tight upper and lower bounds on the number of ways of splitting an even number into two primes, which depend strongly on the number modulo 3.

Two good places to start for more information on the conjecture are `https://en.wikipedia.org/wiki/Goldbach%27s_conjecture` and `https://mathworld.wolfram.com/GoldbachConjecture.html`. We also recommend *Uncle Petros and Goldbach's Conjecture*, by Apostolos Doxiadis [**Dox10**] which, although fictional, is still an enjoyable read. Back in 2001, as part of their effort to promote this book, the publishers announced a $1 million prize for the first

person to prove Goldbach's Conjecture within the next two years. The reward went unclaimed.

2.13.4 Prime Twins. Prime couples of the form $(p, p+2)$ are called *twin primes*. For example: (3, 5), (5, 7), (11, 13), (17, 19), (29, 31), ...

One of the first discoveries when looking at twin primes was that, aside from (3,5), all of them have the form $(6n-1, 6n+1)$. This comes from noticing that any prime greater than 3 must be of the form $6n \pm 1$. To show this, note that any integer can be written as $6x + y$, where x is any integer, and y is 0,1,2,3,4, or 5. Now consider each y value individually.

When $y = 0$, $6x + y = 6x$ and is divisible by 6. When $y = 1$ there are no immediately recognizable factors, so this is a candidate for primacy. When $y = 2$, $6x+2 = 2\cdot(3x+1)$, so is not prime. For the case when $y = 3$, $6x+3 = 3\cdot(2x+1)$ and is not prime. When $y = 4$, $6x + 4 = 2 \cdot (3x + 2)$ and is not prime. When $y = 5$, $6x+5$ has no immediately recognizable factors, so is the second candidate for primacy.

So all primes can be represented as either $6n + 1$ or $6n - 1$, and twin primes, since they are separated by two, will have to be $6n - 1$ and $6n + 1$.

This also means that five is the only prime that belongs to two pairs, as every other pair is at least a multiple of 6 apart, and so can not overlap. Another way to prove this is to note that every third odd number is divisible by 3, and therefore no three successive odd numbers can be prime unless one of them is 3. Five is therefore the only prime that is part of two twin prime pairs.

Listing 2.42 (twins.py) prints all twin primes between $a > 8$ and b. x is the first integer ≥ 2, and since a has the form $6n-1$, then x is initialized to 6 * int(a/6) + 5.

```
a,b=map(int,input("a b=? ").split())
x = 6*math.trunc(a/6) + 5
while x <= b-2:
  if isPrime(x) and isPrime(x+2):
    print(x, x+2)
  x += 6
```

Listing 2.42. Finding twin primes

The following output are all the twin primes between $a = 1000000$ and $b = 1003000$:

```
1000037 1000039     1000721 1000723     1001549 1001551     1002359 1002361
1000211 1000213     1000847 1000849     1001807 1001809     1002719 1002721
1000289 1000291     1000859 1000861     1001981 1001983     1002767 1002769
1000427 1000429     1000919 1000921     1002149 1002151     1002851 1002853
1000577 1000579     1001087 1001089     1002257 1002259     1002929 1002931
1000619 1000621     1001321 1001323     1002341 1002343
1000667 1000669     1001387 1001389     1002347 1002349
```

Note that between 1000000 and 1001000 (the first millennium), there are 11 twins. When we split this interval evenly into 10 centuries, just two centuries are empty, the second and the fourth. Is this significant?

Suppose we toss 11 balls at random into 10 buckets. What is the expected number of empty buckets? Any one bucket remains empty with a probability of 0.9^{11}, so the expected number of empty buckets is $10 * 0.9^{11} = 3.138$. However, we have too little data to show that these twin primes are too evenly spread out to be random, although we might suspect it. This is why the output above covers a range of 3000 (two more millennia), making the data more likely to be random.

Indeed, the distances between twin primes are much larger than the distances between primes, and there's little dependency between them. Certainly, twin primes do become increasingly rare at larger ranges, which is in keeping with the general tendency of gaps between adjacent primes. However, it's unknown whether there are infinitely many twin primes (the *twin prime conjecture*) or if there is a largest pair.

2.13.5 Prime Triples, Quadruples. A prime triple of the form $(p, p+2, p+6)$ is a 2-4 *triple prime*. A 4-2 triple prime has the form $(p, p+4, p+6)$.

A 2-4-2 quadruple prime has the form $(p, p+2, p+6, p+8)$. A 4-2-4 quadruple prime has the form $(p, p+4, p+6, p+10)$. Listing 2.43 (quadruple.py) generates 2-4-2 quadruples.

```
a, b = map(int, input("a b=? ").split())
p = 6*math.trunc(a/6) + 5
while p <= b:
  if isPrime(p) and isPrime(p+2) and \
     isPrime(p+6) and isPrime(p+8):
    print(f"{p:5} {p+2:5} {p+6:5} {p
        +8:5}")
  p += 6
```

Listing 2.43. Finding 4-2-4 quadruple primes

The output for $a = 5$, $b = 20000$:

```
    5     7    11    13
   11    13    17    19
  101   103   107   109
  191   193   197   199
  821   823   827   829
 1481  1483  1487  1489
 1871  1873  1877  1879
 2081  2083  2087  2089
 3251  3253  3257  3259
 3461  3463  3467  3469
```

```
 5651  5653  5657  5659
 9431  9433  9437  9439
13001 13003 13007 13009
15641 15643 15647 15649
15731 15733 15737 15739
16061 16063 16067 16069
18041 18043 18047 18049
18911 18913 18917 18919
19421 19423 19427 19429
```

Apart from the first quadruple all the others end with the digits 1, 3, 7, 9, which is easy to prove (see Ex. 12). So the steps must be multiples of 10 as well

as of 6, i.e. 30-steps. That is, as soon as we find the first quadruple in an interval, we can proceed in steps of 30.

This can also be expressed as all quadruplets except (5, 7, 11, 13) are of the form $(30n + 11, 30n + 13, 30n + 17, 30n + 19)$ for some integer n. This structure is necessary to ensure that none of the four primes are divisible by 2, 3, or 5. A quadruplet of this form is also called a *prime decade*.

All prime decades have center values of the form $210n + 15$, $210n + 105$, or $210n + 195$ since the centers must be -1, 0, or +1 modulo 7.

Suppose we want all quadruples between a and b. Where do we start? With the first term of the sequence $30n$+11 which is $\geq a$, this becomes 30*trunc((a+18)/30) + 11. Check this by plugging in some values for a.

We replace the equation for p in Listing 2.43 by

```
p = 30*math.trunc((a+18)/30)+11
```

The while-test becomes:

```
while p <= b-8:
```

and the increment changes to:

```
p += 30
```

This speeds up the program considerably, although (5,7,11,13) is no longer reported of course.

2.13.6 Relatively Prime Numbers. The probability that two random numbers both have a given prime p as a factor is $1/p^2$. So, the probability that they do *not* have p as a common factor is $1 - 1/p^2$. Therefore, the probability that two numbers have no common prime factors is

$$P = (1 - 1/2^2)(1 - 1/3^2)(1 - 1/5^2)(1 - 1/7^2)(1 - 1/11^2)\cdots$$

Using

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots,$$

this can be rewritten as

$$P = \left((1 + 1/2^2 + 1/2^4 + \cdots)(1 + 1/3^2 + 1/3^4 + \cdots)\cdots\right)^{-1}.$$

By the Unique Factorization Theorem (every positive integer is expressible as the product of primes in exactly one way), this expression is equivalent to

$$P = (1 + 1/2^2 + 1/3^2 + 1/4^2 + 1/5^2 + 1/6^2 + \cdots)^{-1}.$$

In words, the expansion of the multiplications generates reciprocals whose denominators are a product of ever square power of every prime. For example, somewhere we can find $1/(2^2 \cdot 3^4 \cdot 5^2)$, which is the same as $1/(2 \cdot 3^2 \cdot 5)^2$, or $1/90^2$. Indeed, every integer will appear somewhere in P's sum as a squared denominator.

We have arrived at the Basel problem (`https://en.wikipedia.org/wiki/Basel_problem`) which asks for the precise summation of the reciprocals of the squares of the natural numbers:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots$$

Euler solved this, as we outline in Sec. 6.7.7:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

And so, the desired probability is

$$P = 6/\pi^2 \approx 61\%.$$

Listing 2.44 (coPrime.py) uses this probability to obtain an estimate for π by examining 1 million random integer pairs.

```
def estPi(nPairs):
  count = 0
  for i in range(0, nPairs):
    n1 = random.randint(1, MAX_NUM)
    n2 = random.randint(1, MAX_NUM)
    # increment the counter if both integers are coprimes
    if math.gcd(n1, n2) == 1:
      count += 1
  prob = float(count)/nPairs
  return math.sqrt(6 / prob)
```

Listing 2.44. π based on the probability of being relatively prime

The result is only correct to 2 decimal places:

```
> python coPrime.py
Generating 1000000 random pairs
 pi est: 3.1431831955327896
math.pi: 3.141592653589793
```

Recall that $\phi(n)$ = the number of integers less than n that are relatively prime to n (see Sec. 2.1.2). Then $\phi(n)/n$ = the probability that a randomly chosen integer is relatively prime to n. (This is true because any integer is relatively prime to n if and only if its remainder, when divided by n, is relatively prime to n.) The result above therefore tells us that the average value of $\phi(n)/n = 6/\pi^2$.

2.13.7 Prime Spirals. The Ulam spiral, or prime spiral, is a visualization of prime numbers devised by Stanisław Ulam in 1963, and popularized in Martin Gardner's 'Mathematical Games' column in *Scientific American* a year later (March 1964). The column, 'Patterns and Primes' is reprinted in [**Gar71**], chapter 9.

The spiral is constructed by writing the positive integers in a square spiral (see Fig. 2.15a) and then highlighting the prime numbers (see Fig. 2.15b).

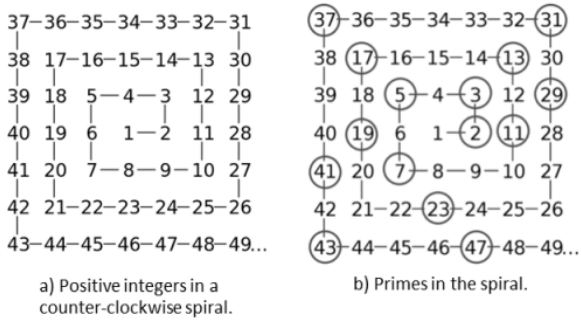


Figure 2.15. Constructing the Ulam spiral

Diagonal, vertical, and horizontal lines with a high density of primes become visible (see Fig. 2.16), which was so striking that the spiral was used as the cover illustration for that month's issue of *Scientific American*.

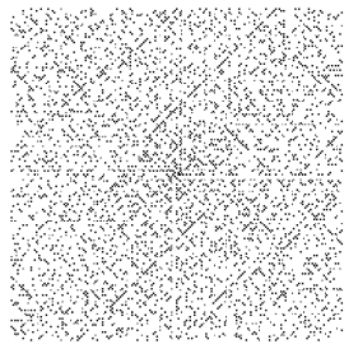


Figure 2.16. The Ulam spiral

Listing 2.16 was generated by `ulamSpiral.py` which uses isPrime() to identify the primes stored in a 2D list. The tricky part of the code is populating the array in the manner shown in Fig. 2.15a. An odd number sized array is created, so ensuring there's a single center cell. The code implements a 'state machine' based around the four states 'up', 'down', 'left', and 'right' corresponding to cell moves. The states are changed in an order that models counter-clockwise rotation, and a counter is used to decide how long each state is occupied. The counter is gradually incremented, which causes the turns to spiral outwards.

Diagonal, horizontal, and vertical lines in the number spiral correspond to polynomials of the form

$$f(n) = 4n^2 + bn + c$$

where b and c are integer constants. When b is even, the lines are diagonal, and either all the numbers are odd, or all even, depending on the value of c. This can be confirmed by manually calculating the equations for some of the diagonals around the center, as in Fig. 2.17. It's no surprise that all the primes other than 2 lie in alternate diagonals of the spiral.

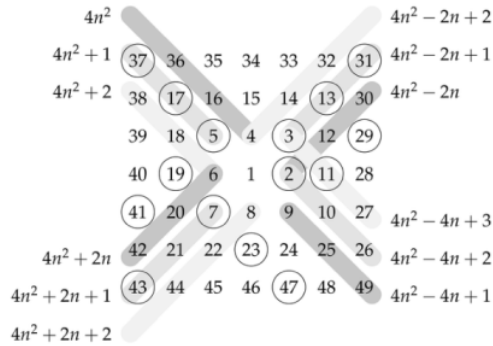


Figure 2.17. Diagonals as quadratic equations

The polynomial function can be obtained by looking at the differences between the values in typical sequences, such as the two shown in Fig. 2.18.

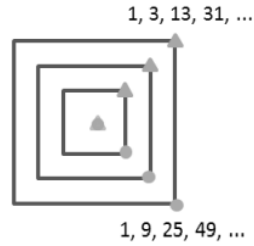


Figure 2.18. Two polynomial sequences in the Ulam spiral

The bottom right sequence in Fig. 2.18 is 1, 9, 25, 49, 81,..., whose difference sequence is 8, 16, 24, 32, ... In general, this b sequence has the form:

$$b_n = b_0 + 8n$$

b_0 is the starting value, which is 0 for this example.

This allows us to define the top-level sequence as:

$$a_n = a_0 + \sum_{k=1}^{n} b_k$$

a_0 is the starting value, which for this example is 1. It can be simplified:

$$\begin{aligned} a_n &= a_0 + \textstyle\sum_{k=1}^{n}(b_0 + 8k) \\ &= a_0 + nb_0 + 8\textstyle\sum_{k=1}^{n} k \\ &= a_0 + nb_0 + 8\left(\frac{n(n+1)}{2}\right) \\ &= a_0 + nb_0 + 4n^2 + 4 \\ &= 4n^2 + bn + c \end{aligned}$$

As to why some of the odd diagonals have a higher concentration of primes than others, consider $4n^2 + 6n + 1$ and $4n^2 + 6n + 5$ as examples. Plug in values for $n = 0, 1, 2,....$ into the first polynomial, and calculate the remainders modulo 3 to get 1, 2, 2, 1, 2, 2,.... The results for the second polynomial are 2, 0, 0, 2, 0, 0,.... This implies that in the second sequence two out of every three numbers are divisible by 3, and so not prime. However, the sequence for the first polynomial contains no numbers divisible by 3, which makes it more likely that it will have primes. It's only 'more likely' since we should also consider the remainders when the sequences are divided by other primes.

A more rigorous argument can use G.H. Hardy and J.E. Littlewood's *Conjecture F* which provides a formula for the number of primes of the form ax^2+bx+c (see `https://en.wikipedia.org/wiki/Ulam_spiral`). It implies that there's considerable variability in the prime density along different diagonals in the spiral. In particular, the density is highly sensitive to the discriminant of the polynomial, $b^2 - 16c$ (with $a = 4$).

Euler first noticed in 1772 that the quadratic polynomial

$$P(n) = n^2 - n + 41$$

is prime for the 40 integers $n = 0, 1, 2, ..., 39$, producing 41, 43, 47, 53, 61, 71, ..., 1601 respectively. This equation uses one of Euler's 'lucky' numbers – positive integers n such that for all integers k with $1 \leq k < n$, the polynomial $k^2 - k + n$ produces a prime number. Only six lucky numbers exist, namely 2, 3, 5, 11, 17 and 41. Note that they're all prime.

The spiralArr(a, N) function in `ulamSpiral.py` takes two arguments: the starting value a at the center of the 2D list and N for the size of the list. In Fig. 2.16, $a = 1$, but it's instructive to change it to one of Euler's lucky numbers. For example, Fig. 2.19 is the spiral generated when $a = 41$.

The prime density along several diagonals is markedly greater than in the earlier figure (Fig. 2.16).

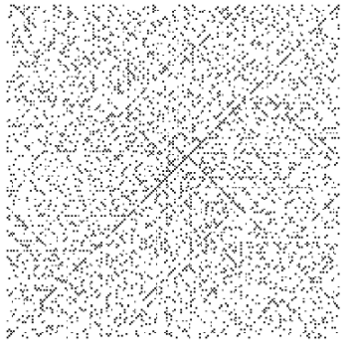


Figure 2.19. The Ulam spiral starting at 41

2.13.7.1 Prime Spiral Variants. Ulam wasn't the first to visualize primes in this way. In 1932, Laurence M. Klauber employed a triangle in which row n contains the numbers $(n-1)^2+1$ through n^2. As in the Ulam spiral, quadratic polynomials generate many primes that lie along diagonals. Also vertical lines, which correspond to numbers of the form k^2-k+M tend to have a high density of primes (see Fig. 2.20).



Figure 2.20. The Klauber triangle

Robert Sacks devised a variant of the Ulam spiral in 1994. Positive integers are plotted along an Archimedean spiral rather than Ulam's square spiral, spaced so that one perfect square occurs in each full rotation, as indicated in Fig. 2.21a.

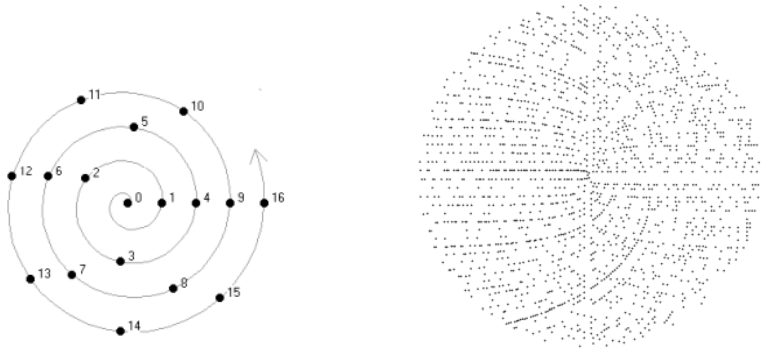
The spiral uses the following equations to generate polar coordinates (r, θ):

$$r = n^{1/2} \quad \text{and} \quad \theta = 2\pi \cdot 360 \cdot n^{1/2}.$$

n is a positive integer. The angle calculation uses $n^{1/2}$ as the number of turns of the spiral, which has to be converted into degrees, and then to radians.

sacksSpiral.py draws the spiral using turtle graphics (Fig. 2.21b), and so its polar coordinates are converted to Cartesian (x, y):

$$x = r\cos\theta, \quad y = r\sin\theta$$



(a) Integer positions (b) Drawing with a turtle

Figure 2.21. The Sacks spiral

The only prime-specific part of sacksSpiral.py is the if-test which decides whether an integer i should be drawn because it's a prime. It's interesting to replace this by other test functions to see how the spiral changes.

2.13.8 Factoring. We want to factorize an integer n, assuming that it's odd and greater than 1, so start with the trial divisor $d = 3$ (factor.py).

```
def factorsOdd(n):
  if n%2 == 0:
    print("Number must be odd")
    return []
  fs = []; d = 3
  while d*d <= n:
    while n == d*int(n/d):
      fs.append(d)
      n = n//d
    d += 2
  if n > 1:
    fs.append(n)
  return fs
```

Listing 2.45. Factorize an odd number

Let's factor some large odd numbers:

```
>>> from factor import *
>>> factorsOdd(123456789)
[3, 3, 3607, 3803]

>>> factorsOdd(999999999999)
[3, 3, 3, 7, 11, 13, 37,
 101, 9901]
```

All primes beyond 3 lie in one of two arithmetic progressions $6n - 1$ and $6n+1$. That is, from 5 onwards the trial divisor d can be incremented alternatively in steps of 2 and 4. This leads to a second version of the function shown in Listing 2.46 (factor.py).

```
def factors(n):
  fs = []
  while n == 2*int(n/2):
    fs.append(2)
    n = n//2
  while n == 3*int(n/3):
    fs.append(3)
    n = n//3
  d = 5; s = 2
  while d*d <= n:
    while n == d*int(n/d):
      fs.append(d)
      n = n//d
    d += s; s = 6-s
  if n > 1:
    fs.append(n)
  return fs
```

Listing 2.46. Factorizing any number

It first divides out the factors 2 and 3. Then it starts with $d = 5$ and step size $s = 2$. The assignment $s = 6 - s$ toggles s between 2 and 4. factors() has the advantage of supporting all positive integers:

```
>>> factors(22222222222222)
[2, 11, 239, 4649, 909091]

>>> factors(999999999999)
[3, 3, 3, 7, 11, 13, 37,
 101, 9901]
```

2.14 n as a Sum of Four Squares

Inspired by Diophantus' classical treatise *Arithmetica*, Bachet (Claude Gaspard Bachet de Méziriac) conjectured in 1621 that every natural number is the sum of at most four squares. He verified this up to $n = 351$. The first proof was given by Lagrange in 1770 (see `https://en.wikipedia.org/wiki/Lagrange%27s_four-square_theorem`). We can tell how difficult this theorem was from the fact that Euler, over the years, settled several special cases, but failed to prove rhe general form, although Lagrange's later proof did make use of Euler's four-square identity.

Listing 2.47 (fourSquares.py) finds all representations of n as $x^2 + y^2 + z^2 + u^2$ with $x \geq y \geq z \geq u$:

```
n = int(input("n=? "))
r = math.trunc(math.sqrt(n))
count = 0
for x in range(r, r//2-1, -1):
  for y in range(x, -1, -1):
    for z in range(y, -1, -1):
      for u in range(z, -1, -1):
        if x*x + y*y + z*z + u*u == n:
          print(f"{x:2} {y:2} {z:2} {u:2}   ", end='')
          count += 1
          if (count % 5 == 0):
            print()
```

Listing 2.47. Sum of four squares

For $n = 169$, 207 and 399, the program outputs:

```
> python.exe fourSquares.py
n=? 169
13  0  0  0   12  5  0  0   12  4  3  0   11  4  4  4   10  8  2  1
10  7  4  2    9  6  6  4    8  8  5  4
> python.exe fourSquares.py
n=? 207
14  3  1  1   13  6  1  1   13  5  3  2   11  9  2  1   11  7  6  1
11  6  5  5   10  9  5  1   10  7  7  3    9  9  6  3
> python.exe fourSquares.py
n=? 399
19  6  1  1   19  5  3  2   18  7  5  1   18  5  5  5   17 10  3  1
17  9  5  2   17  7  6  5   15 13  2  1   15 11  7  2   15 10  7  5
14 13  5  3   14 11  9  1   13 13  6  5   13 11 10  3   13 10  9  7
11 11 11  6
```

Jacobi's four-square theorem gives a formula (denoted by $r_4(n)$) for the number of representations of a natural number n as the sum of four squares:

$$r_4(n) = \begin{cases} 8\sum_{m|n} m & \text{if } n \text{ is odd} \\ 24\sum\limits_{\substack{m|n \\ m \text{ odd}}} m & \text{if } n \text{ is even.} \end{cases}$$

In words, the number of ways to represent n is eight times the sum of the divisors of n if n is odd and 24 times the sum of the odd divisors of n if n is even.

This produces a much larger count than that suggested by the output from fourSquares.py since two representations are considered different if their terms are in a different order or if the integer being squared is negative. For example, $r_4(1) = 8$, three of which are:

$$\begin{aligned} 1^2 + 0^2 + 0^2 + 0^2 \\ 0^2 + 1^2 + 0^2 + 0^2 \\ (-1)^2 + 0^2 + 0^2 + 0^2. \end{aligned}$$

The theorem utilizes the divisors of a number n, which can be calculated using Listing 2.48 (jacobi.py).

```
def divisors(n):
  divs = [1]
  for i in range(2,
      int(math.sqrt(n))+1):
    if n%i == 0:
      divs.extend([i, n//i])
  divs.extend([n])
  divs = list(set(divs))
  divs.sort()
  return divs
```

Listing 2.48. Divisors of a number

```
def sumDivisors(n):
  odSums = 0; evSums = 0
  for i in range(1,
      int(math.sqrt(n))+1):
    if n%i == 0:
      if i%2 == 0:
        evSums += i
      else:
        odSums += i
      if n//i != i:
        if (n//i)%2 == 0:
          evSums += (n//i)
        else:
          odSums += (n//i)
  if n%2 == 1:  # odd
    return 8*(odSums + evSums)
  else:
    return 24*odSums
```

Listing 2.49. Divisors of a number (2)

However, Jacobi's formula only makes use of the sum of the divisors, and also distinguishes between odd and even divisors. This results in a modified divisor algorithm in Listing 2.49. It generates the following sequence of counts for $n = 0$ to 10:

```
1, 8, 24, 32, 24, 48, 96, 64, 24, 104, 144
```

This sequence is listed as A000118 at OEIS (`https://oeis.org/A000118`).

An interesting view of this topic appears in Chapter 4 of Polya [**Pol20**], where it's employed as an example of induction in number theory.

Exercises for Sections 2.13 to 2.14

(1) Denote by $G(2^n)$ the Goldbach count of 2^n. Check the following two-part table (2.11 and 2.12):

n	3	4	5	6	7	8	9	10	11	12
$G(2^n)$	1	2	2	5	3	8	11	22	25	53

Table 2.11. Goldbach count of 2^n.

(2) Plot $\ln(G(2^n))$ versus n. What do you get for large n?

(3) Write a program which checks if a number e satisfies the Goldbach conjecture. That is, it must print True as soon as it finds a representation of e as a sum of two odd primes $a + b$ with $a \leq b$. Also, find the quotient $\ln(e)/\ln(a)$.

n	13	14	15	16	17	18	19	20	21	22
$G(2^n)$	76	151	244	435	749	1314	2367	4239	7471	13705

Table 2.12. Goldbach count of 2^n continued.

(4) Write a program which prints a and e (see the previous question) only if a is larger than all the a's of the preceding e's. Once again find $\ln(e)/\ln(a)$.

(5) Print the first 20 emirps. An *emirp* ('prime' spelled backwards) are primes that when reversed (in their decimal representation) become different primes. (This rules out palindromic primes.)

(6) Find all positive integer solutions (x_n, y_n) of $x^2 - dy^2 = 1$ for $d = 2, 3, 5, 7, 10$ in a sufficiently large interval. From the data, guess recurrences $x_{n+i} = f(x_n, y_n)$, $y_{n+1} = g(x_n, y_n)$ and prove them by induction.

(7) Write programs which compute prime triples of type 2-4 as well as 4-2.

(8) Find the first prime p beyond a max value where $2p + 1$ is also a prime. Use max = 1000, 10000, 100000, 1000000, 10000000, 100000000.

(9) Are there quadratic polynomials which assume unusually many prime values? Count the primes of the form $x^2 + 1, x^2 + x + 41, x^2 + x + 1$ up to $x = 4000$.

(10) Rewrite isPrime() (Listing 2.39) more efficiently as follows: starting with 5 you can proceed alternately in steps of 2 and 4. If we set initially $s = 2$ then the counter $s = 6 - s$ oscillates between 2 and 4.

(11) Rewrite the function isPrime() in Ex. 10 to start from 7, with suitably changed step sizes, and find out if there is a further speedup.

(12) A *superprime* is an integer that is prime, *and* so is every integer obtained by deleting a digit from its right-hand side. For example, 7331 is prime, and so are 733, 73, and 7.

a) What 2- and 3-digit superprimes exist?

b) Which digits of a superprime can be a 1, 2, 4, 5, 6, 8, and 0?

(13) Write programs for computing 4-2-4 prime quadruples.

(14) A *Wilson prime of order n* is a prime number p such that p^2 divides $(n-1)! \cdot (p-n)! - (-1)^n$. For example, if $n = 1$, the formula reduces to $(p-n)! + 1$ where the only known primes are 5, 13, and 563.

Generate the Wilson primes for orders $n = 1$ to 11 and $p < 11000$.

(15) Factor the Fermat number $F_5 = 2^{2^5}+1 = 4294967297$ with the two factor functions (Listings 2.45 and 2.46) and compare their run times. Use the following function for generating this Fermat number (and others):

```
def fermat(n):
  return 2**(2**n) + 1
```

(16) Compute a list of natural numbers which are not the sums of three squares (0 is allowed) and make a conjecture about the form of these numbers.

(17) Prove that the numbers of the form identified in Ex. 16 are indeed not the sums of three squares. *Remark*: it is a deep theorem that every positive integer not of the form given in the solution of Ex. 16 is the sum of three squares.

(18) a) Prove that a number of at least four digits cannot be equal to the sum of the cubes of its digits.

b) Find all the numbers with at most three digits which are equal to the sum of the cubes of their digits.

(19) *Wheels*. To factor a large number n we first divide it by 2, 3, 5, 7. From then on we can use step sizes 4, 2, 4, 2, 4, 6, 2, 6 of period 8 to get the trial divisors 11, 13, 17, 19, 23, ... which contain all the remaining primes. Use this to write a factoring function.

(20) Find the smallest integer which can be represented as a sum of three 4th powers in two different ways.

(21) Sieve out the positive integers which cannot be represented in the form: a) $x + y + xy$; b) $x + y + 2xy$ with positive integers x, y. Comment.

(22) Find all the numbers n up to 1000 so that all the numbers of the form $n - 2^k$ with $2 \le 2^k < n$ are prime. The six numbers you will find are the only known numbers with this property.

2.15 The Maximum of a Unimodal Function

A function which is strictly increasing up to a point in an interval and strictly decreasing past that point is called *unimodal* in that interval. We want to find the maximum of a unimodal function f in an interval $[x1, x3]$.

The *golden-section* search method is suitable for when we have no information about the derivative of f, and must find a maximum or a minimum. Its name is derived from the fact that it calculates new points, $x2$ and $x4$, by using ϕ, the golden ratio. This guarantees that the interval width shrinks by the same constant proportion in each step, which means that the algorithm avoids the generation of very small intervals which may cause rounding errors (at least until it gets very close to the answer).

The starting situation is illustrated in Fig. 2.22. We are supplied with the starting interval $[x1, x3]$ and must calculate two new points, $x2$ and $x4$.

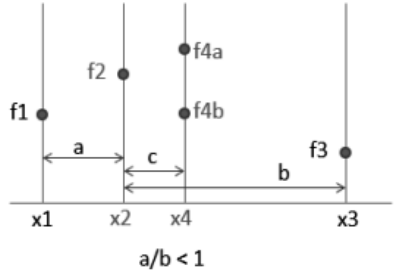


Figure 2.22. Starting a golden-section search

$x2$ will be chosen so it is a units from $x2$ and b units from $x3$, such that the intervals ratio $a/b < 1$. $x4$ will be c units from $x2$, with c determined by ensuring that the shrinking intervals ratios stay the same as the original a/b.

Each of the x points has a y value represented by $f1$, $f2$, etc. in Fig. 2.22. Note that there are two possible y-values for $x4$ – when the gradient at $f2$ is decreasing, $x4$ will have the y-value $f4b$. But when $f2$'s gradient is increasing then $x4$ will be assiged $f4a$. The two cases are illustrated in Fig. 2.23.

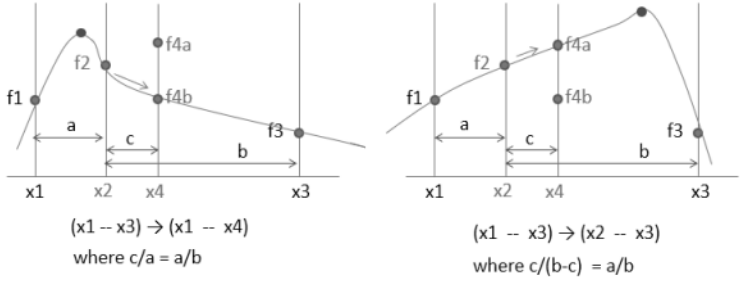


Figure 2.23. Reducing the golden-section search interval

In the first case (gradient decreasing at $f2$), then the maximum must be somewhere between $x1$ and $x2$, and so the search interval is reduced from $[x1, x3]$ to $[x1, x4]$, but with the proviso that

$$\frac{c}{a} = \frac{a}{b}.$$

In the second case (gradient increasing at $f2$), then the maximum must be somewhere between $x4$ and $x3$, and so the interval is reduced from $[x1, x3]$ to $[x2, x3]$,

but with the proviso that

$$\frac{c}{b-c} = \frac{a}{b}.$$

These two constraints on how the interval is reduced are enough for us to derive equations for calculating $x2$ and $x4$.

Eliminating c from these two simultaneous equations yields

$$\frac{b}{a}^2 - \frac{b}{a} = 1.$$

If we substitute $x = b/a$ and solve $x^2 - x - 1 = 0$, the positive root is $x = \frac{1+\sqrt{5}}{2} = \phi$. Therefore, to maintain the same interval ratio throughout the search, we must set $a/b = 1/\phi$.

The code implements this idea when calculating $x2$ and $x4$. The key lines are:

```
PHI = (math.sqrt(5) + 1) / 2
x2 = x3 - (x3-x1)/PHI
x4 = x1 + (x3-x1)/PHI
```

We can show these assignments are correct by rewriting them in terms of the intervals used in Fig. 2.22, and the required intervals ratios:

$$\frac{a}{b} = \frac{1}{\phi} \quad \text{or} \quad \frac{b}{a} = \phi \tag{2.11}$$

$$\frac{c}{a} = \frac{1}{\phi} \quad \text{or} \quad \frac{a}{c} = \phi \tag{2.12}$$

$$\frac{c}{b-c} = \frac{1}{\phi} \quad \text{or} \quad \frac{b-c}{c} = \phi \tag{2.13}$$

Considering the $x2$ assignment:

$$\begin{aligned} x2 &= x3 - (\tfrac{x3-x1}{\phi}) \\ \phi &= \tfrac{x3-x1}{x3-x2} = \tfrac{b+a}{b} \quad \text{from Fig. 2.22} \\ &= 1 + \tfrac{a}{b} \quad \text{using Equ. 2.11} \\ &= 1 + \tfrac{1}{\phi} \\ \phi^2 &= \phi + 1 \end{aligned}$$

Considering the $x4$ assignment:

$$\begin{aligned} x4 &= x1 + (\tfrac{x3-x1}{\phi}) \\ \phi &= \tfrac{x3-x1}{x4-x1} = \tfrac{b+a}{c+a} \quad \text{from Fig. 2.22} \\ &= \tfrac{a\phi+a}{\frac{a}{\phi}+a} \quad \text{using Equs. 2.11 and 2.12} \\ &= \tfrac{\phi+1}{\frac{1}{\phi}+1} = \phi\tfrac{\frac{1}{\phi}+1}{\frac{1}{\phi}+1} = \phi \end{aligned}$$

The preceding lines only caoture half of the rewrite since c can alternatively be replaced by Equ. 2.13 on the third line. We'll leave the details to you, but it uses the equality $c/a = b/(a+b)$ (a combination of Equs. 2.11 and 2.13) and ends in the same way as the $x2$ assignment rewrite, with $\phi^2 = \phi + 1$.

```
def goldSrch(fn, x1, x3):
  while abs(x3-x1) > EPS:
    x2 = x3 - (x3-x1)/PHI
    x4 = x1 + (x3-x1)/PHI
    if fn(x2) > fn(x4):
      x3 = x4
    else:
      x1 = x2
  return (x3+x1)/2

fn = lambda x: 1 - ((x-2)**2)
xMax = goldSrch(fn, 1, 5)
print(f"Maximum x: {xMax:.8f}")
```

Listing 2.50. Using the golden-section search

Listing 2.50 (goldSrch.py) uses goldSrch() to find the maximum of $y = 1-(x-2)^2$ in the interval $[1, 5]$. The result:

```
> python goldSrch.py
Maximum x: 2.00000001
```

Golden-section search was first reported by J. Kiefer in 'Sequential Minimax Search for a Maximum', *Proc. Amer. Math. Soc.*, Vol. 4, No 3, pages 502–506, June 1953. He also developed the closely related Fibonacci search technique which finds the maximum/minimum of a sequence of values that has a single local maximum/minimum. A golden-section search is approximated using Fibonacci numbers applied to the indices of an integer sequence.

Note that the bisection method is a similar algorithm, but for finding a zero of a function (see Sec. 1.5). Also, bisection narrows in on the root using only two points, while golden-section search employs four, and bisection reduces the interval by a factor of two in each step, rather than by the golden ratio.

2.16 A Very Difficult Problem

We want to show that if a, b, and q are positive integers with $a^2 + b^2 = q(ab + 1)$, then q is a perfect square.

This problem was proposed by the German Federal Republic group at the XXIX. IMO in Canberra, 1988. The Australian Problem Committee liked it a lot, but nobody could solve it, including George Szekeres and his wife, both famous problem solvers and creators. So it was passed to the four most eminent number theorists in Australia, but after six hours none of them had solved it either. To cut a long story short, the problem was chosen for the Olympiad, and eleven high school students produced complete solutions!

Suppose we are mathematicians of average ability, and have Python at our disposal, then the problem becomes comparatively simple. First, collect data, and study it for clues. When we see how to generate all the solutions output by programming, an elegant proof may suggest itself to us.

a) Due to the symmetry in a and b, we can assume that $a \le b$, and it's also easy to see that $a = b$ only when $a = b = q = 1$. So we may additionally assume that $a < b$.

b) At this stage we should write a short piece of code (see Ex. 1) to generate solutions in a small range, such as $a \le 150, b \le 1000$. We get:

a	1	2	3	4	5	6	7	8	8	9	10	27	30	112
b	1	8	27	64	125	216	343	30	512	729	1000	240	112	418
q	1	4	9	16	25	36	49	4	64	81	100	9	4	4

Table 2.13. Solutions to the difficult problem

c) An examination of Table 2.13 suggests that $(a, b, q) = (c, c^3, c^2)$. Indeed:

$$c^2 + c^6 = c^2(c^4 + 1).$$

We have found one solution for each square.

d) Now turn to the triples that produce the same q, such as when $q = 4$:

2	8	30	112	a_1	b
8	30	112	418	b	a_2
4	4	4	4	q	q

Table 2.14. Solutions when $q = 4$

The second value of each triple (a, b, q) is the first value of the next triple, which suggests the transformation

$$(a_1, b, q) \rightarrowtail (b, a_2, q).$$

We can rewrite $a^2 + b^2 = q(ab + 1)$ as an explicit diophantine equation

$$a^2 - qba + b^2 - q = 0 \tag{2.14}$$

This can be viewed as a quadratic in a, with two solutions a_1 and a_2, which satisfy

$$a_1 + a_2 = qb \tag{2.15}$$

$$a_1 \cdot a_2 = b^2 - q \tag{2.16}$$

Equ. (2.15) shows that both a_1 and a_2 are integers, and we have

$$a_2 = qb - a_1. \tag{2.17}$$

Since $q \geq 2$ and $b > a_1$, we have $a_2 >$ b. So for any b, the two solutions a_1 and a_2 of Equ. (2.14) straddle b. By symmetry, the two solutions b_1, b_2 of Equ. (2.14) for a given a also straddle a. Thus one can get ever larger pairs a and b of integers which satisfy Equ. (2.14) for a fixed q.

We shall now prove that q is a square by going downwards through this family of solutions.

The original equation $a^2 + b^2 = q(ab + 1)$ shows that ab can not be negative. Hence a and b must have the same sign (or possibly one of them is 0). Using the straddling property we can alternately replace the larger of a or b by a smaller non-negative integer. Eventually, one of the numbers must reach 0, and q will be the square of the other.

Exercises

(1) Write a program which prints the triples a, b, and q in Table 2.13.

(2) Let a and b be positive integers such that $ab + 1$ divides $a^2 + ab + b^2$. Show that $(a^2 + ab + b^2)/(ab + 1)$ is the square of an integer. Explore the situation as in the IMO problem.

(3) Consider the general term $q = (a^2 + r \cdot ab + b^2)/(s \cdot ab + t)$ with $r =$ -1, 0, or 1, and positive integers s and t. What condition between s and t must be satisfied in order that a) q is a square; b) $q \cdot t$ is a square.

(4) *Empirical Exploration.* Start with a stack of n coins, each having their 'head' face up. Now turn over the top i-substack and place it back on top for $i = 1, 2, ..., n, 1, 2, ..., n, 1, 2, ...$ For each i, check if the initial state (all 'heads') has re-occurred. Let $f(n)$ be the number of reversals until this happens. Write a program to calculate $f(n)$. Can you also discover a formula for $f(n)$?

A variant of this problem is concerned with *burnt pancakes*, where each pancake has a burnt side and all pancakes must end up with the burnt side on the bottom.

Pancake sorting is a recognized sorting type, where the aim is to put a disordered stack of pancakes of different sizes into size order by inserting a spatula

at any point in the stack and using it to to flip all the pancakes above it (see `https://en.wikipedia.org/wiki/Pancake_sorting`). The problem is notable as the topic of a paper by Microsoft founder Bill Gates, co-authored with Christos Papadimitriou.

Additional Exercises for Chapters 1 to 2

(1) If n runs from 1 to a^2, the function $f(n) = \lfloor n + \sqrt{2n} + 0.5 \rfloor$ runs up to $a^2 + a$, leaving out exactly a numbers.

a) Write a program which prints the skipped values.

b) Make a conjecture and try to prove it.

This question, and the next, comes from Essay 12 in Honsberger [**Hon70**].

(2) If n runs from 1 to some upper bound max, then $f(n) = \lfloor n + \sqrt{2n} + 0.5 \rfloor$ will skip some values.

a) Write a program which prints the skipped values.

b) Make a conjecture and try to prove it.

(3) a) For $k = 2, 3, 4, \ldots$ investigate the values left out by the functions $f_k(n) = \lfloor n + \sqrt{n/k} + 0.5 \rfloor$.

b) Try to guess a closed formula for these values.

c) Prove your conjecture.

(4) a) Find the values skipped by $f(n) = \lfloor n + \sqrt{kn} + 0.5 \rfloor$. Develop a closed formula for the skipped values and prove your conjecture.

b) Find the values skipped by $f(n, p, q) = \lfloor n + \sqrt{np/q} + 0.5 \rfloor$ for positive integers p, q.

(5) *Stirling Numbers of the Second Kind.* Let $S(n, k)$ be the number of partitions of the set {1, 2, ..., n} into exactly k nonempty subsets. Show that

$$S(n, 1) = S(n, n) = 1, \quad S(n, k) = 0 \text{ for } k > n,$$
$$S(n, k) = S(n-1, k-1) + kS(n-1, k).$$

Write recursive and iterative programs for $S(n, k)$.

(6) *Stirling Numbers of the First Kind.* Let $f(n,k)$ be the number of permutations of {1, 2, ..., n} with exactly k cycles. Show that

$$f(n,k) = f(n-1,k-1) + (n-1)f(n-1,k).$$
$$f(n,1) = (n-1)! \quad f(n,n) = 1 \quad f(n,k) = 0 \text{ for } k > n.$$

Write recursive and iterative programs for $f(n,k)$.

(7) Find all the representations of n in the form $x^3 + y^3 + z^3$ with non-negative integers x, y, z. Explain why numbers of the form $9k \pm 4$ cannot be the sums of three cubes.

(8) Find all the representations of n as a sum of four non-negative integer cubes.

(9) Write a program for listing the integers from 1 to some maximum which can be represented in the form $x^2 + 2y^2$.

(10) Show empirically that any number from 1 to 10000 can be represented as a sum of at most three triangular numbers, i.e. numbers of the form $\binom{n}{2}$. Do it by sieving.

In 1796, Gauss showed that any positive integer is the sum of three triangular numbers. It's known as the *Eureka theorem* because of what he wrote in his diary: EYPHKA! Num = $\triangle + \triangle + \triangle$

(11) *The mode* (*plateau problem*). Let a be a list of n integers sorted into increasing order. The mode m is the most frequent value. Construct a function which finds m and its frequency f. The program should also find the length of the longest plateau (stretch of equal values) for any list, even if it is not sorted.

(12) *Common elements of two sequences.* We are given two lists f and g of natural numbers sorted into increasing order. Find the terms which occur in both sequences and the number k of common terms.

(13) A sequence $g(n)$ is defined by $g(0) = 0$ and $g(n) = n - g(g(n-1))$ for $n > 0$.

a) Construct a recursive program for $g(n)$ and compute its values up to $g(40)$.

b) Write an iterative version.

c) Plot the sequence and try to guess a closed formula for $g(n)$.

d) Prove your conjecture.

This question, and the next two, come from Hofstadter [**Hof99**].

(14) A sequence $h(n)$ is defined by

$$h(0) = 0 \text{ and } h(n) = n - h(h(h(n-1))) \text{ for } n > 0.$$

a) Write a recursive program for $h(n)$ and find its values up to $h(40)$.

b) Write an iterative version.

c) Plot the sequence and make conjectures about a closed formula for $h(n)$.

(15) A sequence $q(n)$ is defined by

$$q(1) = q(2) = 1 \text{ and } q(n) = q(n - q(n-1)) + q(n - q(n-2)) \text{ for } n > 2.$$

a) Write a recursive program and compute the values of $q(n)$ up to $q(30)$.

b) Write an iterative version and compute the values up to $q(2000)$.

c) Print the integers 7, 13, 15, 18, ... that get left out. Are there infinitely many of these? (This is a hard unsolved problem.) See also Ex. 23, below.

(16) Find all 4-2-4-2-4 sextuples of primes below 50000.

(17) Find numerical evidence for the following theorem due to Euler:

Every number 2^n for $n \geq 3$ can be represented in the form $2^n = 7x^2 + y^2$ with odd x and y.

From the numerical evidence deduce a proof.

This problem comes from the 48th Moscow Mathematics Olympiad, 1985. A book containing over 60 years of these problems (with hints and solutions), edited by D. Leites, can be downloaded from `https://diendantoanhoc.org/index.php?app=core&module=attach&section=attach&attach_id=22389`. An alternative source is AoPS' Olympiad Archive at `https://artofproblemsolving.com/wiki/index.php?title=Olympiad_Archive`.

(18) Consider again Perrin's sequence v_n defined by $v_0 = 3$, $v_1 = 0$, $v_2 = 2$, $v_n = v_{n-2} + v_{n-3}, n \geq 3$ from Sec. 1.5. Write a program which for each n from 3 to 1 million, tests if $n | v_n$. If the test is positive and n is not prime, it should print n. Do you now have enough evidence for the conjecture $n | v_n \Leftrightarrow n$ is a prime?

(19) On visiting Srinivasa Ramanujan, G.H. Hardy mentioned that he took taxi #n, but this number did not look remarkable to him. "On the contrary", replied Ramanujan, "it is the smallest natural number which can be represented as a sum of two cubes in two different ways" Find n. (Next problem is a continuation.)

(20) Hardy then asked if Ramanujan knew the answer to the problem for fourth powers. After some deliberation, he replied: "I cannot find an example, I think the first such number must be large". Indeed, Euler had earlier discovered that $635318657 = x^4 + y^4 = u^4 + v^4, x < y, u < v, x \neq u$. Find x, y, u, v. Is this the smallest such number?

(21) *The smallest common element of three ordered lists.* Given are three increasingly ordered lists F, G, and H. Find the smallest common element. Since the lists are ordered, we need the smallest i, such that $F[i] = G[j] = H[k]$. Apply the algorithm to the Fibonacci sequence F, to the squares G, and to the multiples H of 9. Print i, i, k, and $F[i]$.

(22) A sequence is defined as

$$f(1) = f(2) = 1, \quad f(n) = f(f(n-1)) + f(n - f(n-1)) \text{ for } n > 2.$$

Table 2.15 shows some values of this sequence. We observe that $f(13) = \cdots = f(16) = 8$.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
f(n)	1	1	2	2	3	4	4	4	5	6	7	7	8	8	8	8	9	10	11	12	12

Table 2.15. The $f(n)$ sequence.

There is an *interval of constancy* of length 4 starting at 13. On the other hand, f(16) to f(20) is a run of length 5. Between 1 and some maximum, find:

a) the leftmost (rightmost) longest interval of constancy,

b) the leftmost (rightmost) longest run.

(23) Let's return to the sequence in Ex. 15., defined by $q(1) = q(2) = 1$ and $q(n) = q(n-q(n-1)) + q(n-q(n-2))$ for $n > 2$. Hofstadter [**Hof99**] calls its growth pattern chaotic, but that only applies to local growth. Find a simple global regularity in its growth.

(24) *McCarthy's Function.* A function G is defined on the integers as follows:

$$G(n) = \begin{cases} n - 10 & \text{for } n > 100 \\ & \text{else } G(G(n+11)). \end{cases}$$

Write a recursive program which finds $G(n)$.

(25) A function Q is defined on the integers by

$$Q(a,b) = \begin{cases} 0 & \text{if } a < b \\ Q(a-b,b) + 1 & \text{if } a \geq b. \end{cases}$$

Find $Q(a,b)$.

(26) There is a conjecture that every large integer n is either a square, or the sum of a prime and a square. Write a program which finds all the numbers from 0 to 32000 which are not representable in this way. Is the conjecture plausible? Also write a program, which for input n, prints the number of representations $n = m^2 + p$. For $n = 11$ it should print 3, since $11 = 0^2 + 11 = 2^2 + 7 = 3^2 + 2$.

(27) Find the function L defined on the integers by

$$L(n) = \begin{cases} 0 & \text{if } n = 1 \\ L(n \text{ div } 2) + 1 & \text{if } n > 1. \end{cases}$$

(28) What does Listing 2.51 (product.py) do for the inputs x, y?

```
def prod(x,y):
  if y == 1:
    return x
  elif (y%2 == 1):
    return x + prod(x, y-1)
  else:
    return prod(x+x, y//2)

x,y = map(int, input("x y=? ").split())
print(prod(x,y))
```

Listing 2.51. Not a prime ordering function

(29) Find a permutation of the set {1, 2, ..., 9} so that the number consisting of the first n digits is divisible by n for every n from 1 to 9. The most obvious coding approach is rather slow; there are several ways to speed up the search.

(30) Let B_n be the number of partitions of an n-set. The numbers B_n can be computed by means of the "Bell triangle" below. The first column is $B_0, B_1, B_2, ...$ and the last entry in each row is the next Bell number. Here $B_0 = 1$ by definition.

```
  1
  1   2
  2   3   5
  5   7  10  15
 15  20  27  37  52
 52  67  87 114 151 203
203 255 322 409 523 674 877
877
```

How is this triangle formed? Write a program which for input n prints the Bell triangle up to line n.

(31) Write a program based on $S(n,k)$, the *Stirling Numbers of the Second Kind* (see Ex. 5) which for input n prints B_n.

(32) a) Write a program which prints

```
1
2 2
3 3 3
4 4 4 4
.........
```

b) Show that $a_n = \lfloor\sqrt{2n} + 0.5\rfloor$ is a closed expression for the n-th term of this sequence when written linearly as 1, 2, 2, 3, 3, 3, 4, 4, 4, 4,....

c) It is claimed that $a_n = \lfloor\frac{1}{2}(1 + \sqrt{8n-7}))\rfloor$ and $a_n = \lceil\frac{1}{2}(\sqrt{8n+1} - 1)\rceil$ are also closed expressions for the n-th term. Check if these claims are true for $n = 1..1000$.

(33) a) Write a program which prints the sequence

```
1
2 4
5 7 9
10 12 14 16
.............
```

b) Show that $a_n = 2n - \lfloor\sqrt{2n} + 0.5\rfloor$ is a closed expression for the n-th term of the sequence when written linearly.

(34) Let X and Y be natural numbers. We say that X is contained in Y if the binary representation of Y goes over into that of X when some (or possibly no) digits are omitted. For example, $X = 1010$ is contained in $Y = 1001100$. Construct an algorithm which, for given natural numbers A, B, finds the largest C contained in A as well as in B.

This question comes from the International Collegiate Programming Contest (ICPC), 1987. Archives dating back to 2008 are online at `https://icpcarchive.github.io/`, and questions for the years 1991 to 2006 can be found in Poucher [**PR09**]. For older problems, you may need to use the Internet Archive's Wayback Machine. we had success with the link `http://icpc.baylor.edu/past/`, then clicking on the tab "Past Problems".

(35) A sequence is defined by $a_1 = 1, a_n = \lfloor\sqrt{a_1 + ... + a_{n-1}}\rfloor, n > 1$. Explore this sequence.

(36) *Ulam's lucky number.* From the list 1, 2, 3, 4,... of all positive integers remove every second number, leaving 1, 3, 5, 7, 9,.... Since 3 is the first number (above 2) that has not been used as a sieving number, we remove every third number from the remaining numbers, yielding 1, 3, 7, 9, 13, 15, 19, 21, Now every seventh number is removed, leaving 1, 3, 7, 9, 13, 15, 21, Numbers that are never removed are considered to be "lucky". Write a program that prints the lucky numbers up to some maximum.

3

Probability

Before tackling this chapter, it may be advisable to have a look at Appendix A if you're probability skills need a refresh.

3.1 A Pseudo-Random Number Generator

A pseudo-random number generator (PRNG) creates a sequence of numbers that appear to be random. The algorithm starts with a seed number, and if we let $R(t)$ represent the number returned by the PRNG in step t, then the seed is $R(0)$. An arithmetic function is applied to the seed to get the first number R(1) in the pseudo-random sequence. To get the next, we apply the function to $R(1)$, producing $R(2)$. This process continues for $R(3)$, $R(4)$, ..., for as long as we need "random" numbers.

One of the simplest algorithms is the *Lehmer PRNG* which employs the following difference equation:

$$R(t) = a \cdot R(t-1) \bmod m$$

where m is a prime number and a an integer between 1 and $m-1$.

Care must be taken when choosing values for m and a otherwise the generated sequence may be very far from random. For example, if we set $m = 13$ and $a = 5$, then the sequence produced is 5, 12, 8, 1, 5, 12, 8, ...

Listing 3.1 (lehmer.py) utilizes $m = 2^{31} - 1$ and $a = 16807$, and includes a simple visual check of the resulting randomness by grouping the generated sequence into 100 histogram buckets which should exhibit a uniform distribution.

```
n = 10000
```

```
MOD = 2**31 - 1    # a prime number
CONST = 16807      # an integer between 1 and MOD-1

def lehmer(r):
  # Compute pseudo-random number using a Lehmer PRNG
  return (CONST * r) % MOD

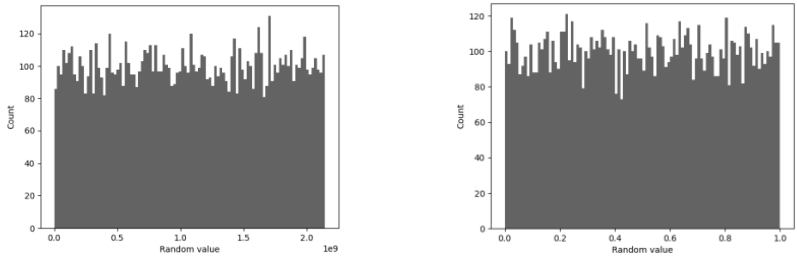
def randomSeq(n, seed):
  r = seed
  rands = []
  for i in range(n):
    # r = lehmer(r)
    r = random.random()
    rands.append(r)
  return rands

seed = datetime.now().timestamp()
print("seed:", seed)
samples = randomSeq(n, seed)
plt.hist(samples, 100)
plt.xlabel("Random value")
plt.ylabel("Count")
plt.show()
```

Listing 3.1. Display a histogram generated by the Lehmer PRNG

Note that a date timestamp is used as the seed. This will be different each time the program is called, which enhances the sequence's randomness.

Typical output is shown in Fig. 3.1a.



(a) The Lehmer PRNG (b) random.random()

Figure 3.1. Random histograms

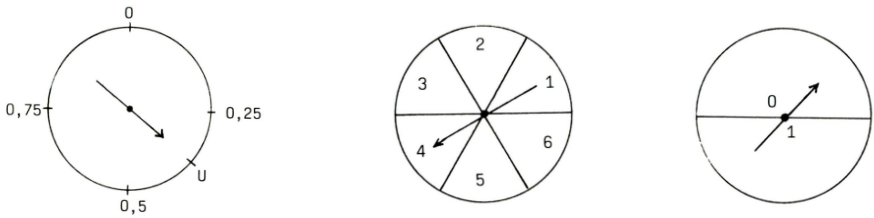
The data is fairly uniformly distributed about 100, which is a result of generating 10000 numbers and putting them into 100 buckets.

It's interesting to compare this output with Python's random(), which only requires the call to lehmer() in Listing 3.1 to be replaced by random.random(). The resulting plot is shown in Fig. 3.1b.

There's no apparent difference between Figs. 3.1b and 3.1a apart from the fact that random() generates values in the range $[0, 1)$ whereas lehmer() returns values in $[0, m-1)$.

In practice, there's really no need for us to write our own PRNG, since Python's random module already contains an excellent range of functions.

3.2 Python's PRNGs



(a) Uniform (b) A fair die (c) A fair coin

Figure 3.2. Types of spinners

Python's random module implements PRNGs for various distributions. The most important for us will be

- **random.random()**: produces a random float n uniformly distributed in the *half-open* range $0.0 \leq n < 1.0$ (Fig. 3.2a). *Uniformly distributed* means n falls into the interval $[0, 1)$ with a probability equal to the width of that interval.
- **random.uniform(a,b)**: returns a random float n uniformly distributed in the *closed* range $a \leq n \leq b$.
- **random.randint(a,b)**: returns a random integer n such that $a \leq n \leq b$. For example, random.randint(1,6) mimics the roll of a fair die (Fig. 3.2b).
- **random.choice(seq)**: returns a random element from the supplied sequence. For instance, random.choice([0,1]) replicates the tosses of a fair coin with its faces labeled as 0 and 1 (Fig. 3.2c).

For integers, there's also functions to generate a random permutation of a list, and random sampling without replacement. On the real line, there are functions to compute normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. It's well worth your time to have a quick look through the module's documentation at `https://docs.python.org/3/library/random.html`.

Almost all of the module functions depend on random(), which produces 53-bit precision floats with a period of $2^{19937} - 1$. The underlying implementation in C is of the Mersenne Twister, one of the most extensively tested PRNGs in existence (`https://en.wikipedia.org/wiki/Mersenne_Twister`). However, since it's deterministic, it's unsuitable for cryptographic purposes, for which Python offers the `secrets` module.

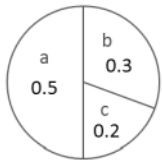


Figure 3.3. Weighted choices

A 'spinner' type that we'll need a few times is *weighted choice*. For example, we wish to choose from the set $\{a, b, c\}$ with the probability distribution $\{0.5, 0.3, 0.2\}$ (see Fig. 3.3).

This can be implemented as in Listing 3.2 (`choices.py`) using a variant of random.choices() which takes a list of weights for driving the selection. The optional keyword argument *k* allows multiple requests to be made at once.

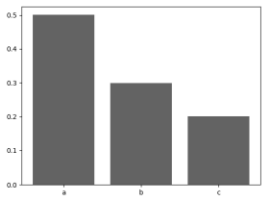


Figure 3.4. Weighted choices plotted

Listing 3.2 also illustrates the use of collections.Counter() for creating a dictionary of counts for each sample (in this case, 'a', 'b', and 'c'). The plot (Fig. 3.4) shows that the samples are distributed with the requested probabilities.

```
pop = ['a', 'b', 'c']
probs = [0.5, 0.3, 0.2]
size = 10**6
samples = random.choices(pop, probs, k=size)
      # get a million samples
countDict = collections.Counter(samples)
plt.bar(countDict.keys(),
       [v/size for v in countDict.values()])
plt.show()
```

Listing 3.2. Plotting weighted choices

3.3 Random Walks

The code in the following programs use variants of random.choice() to perform walks on a line, the plane, and in 3D space, starting at the origin.

```
def rWalk1D(n):
  ts = [0]
  y = 0
  locs = [y]
  for i in range(1, n+1):
```

```
    ts.append(i)
    y += random.choice([-1,1])
    locs.append(y)
  return ts, locs
```

Listing 3.3. Random 1D walk

In Listing 3.3 (rWalk1D.py) the choice is between -1 and 1 which forces a step of 1 unit at each iteration. Typical output is in Fig. 3.5.



Figure 3.5. Plotted random 1D walk

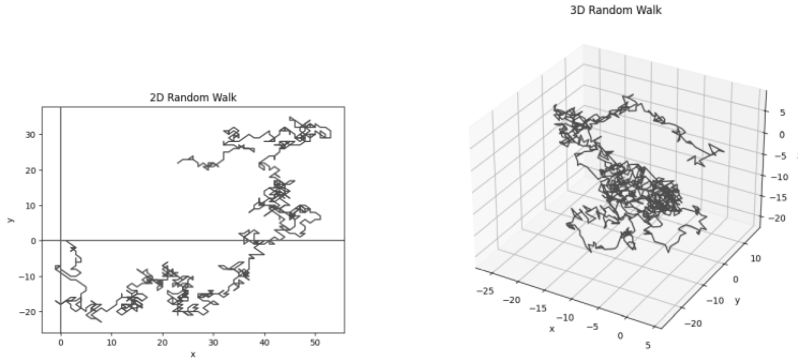
A 2D walk (rWalk2D.py) looks more interesting if the changes to the (x, y) coordinate can be -1, **0**, or 1.

```
def rWalk2D(n):
  x = 0
  xs = [x]
  y = 0
  ys = [y]
  for _ in range(1, n+1):
    x += random.choice([-1,0,1])
    xs.append(x)
    y += random.choice([-1,0,1])
    ys.append(y)
  return xs, ys
```

Listing 3.4. Random 2D walk

Typical output is shown in Fig. 3.6a.

A 3D random walk is analogous, with choice() also modifying a z coordinate. Typical output of rWalk3D.py is shown in Fig. 3.6b.



(a) 2D walk (b) 3D walk

Figure 3.6. Plotted random walks

3.4 Problems Involving Random 1D Walking

A 'frog' randomly jumps along the x-axis, starting from the $x = i$. At each time interval, it moves one integer to the right with probability p or one integer to the left with probability $q = 1 - p$ (see Fig. 3.7). The journey stops when it lands on one of the *absorbing* points $x = 0$ or $x = N$.

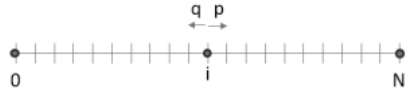


Figure 3.7. Random walk with two absorbing barriers

Let P_i be the probability that starting from $x = i$, the frog reaches $x = N$. It may move either to the right (to P_{i+1}) with a probability p or to the left (to P_{i-1}) with a probability of q. Therefore:

$$P_i = pP_{i+1} + qP_{i-1} \tag{3.1}$$

Also, $P_0 = 0$ (there's no chance of getting to N when you've been absorbed at $x = 0$) and $P_N = 1$.

Since $p + q = 1$, the left-hand side of Equ. (3.1) can be re-written as

$$pP_i + qP_i = pP_{i+1} + qP_{i-1}$$

yielding

$$P_{i+1} - P_i = \frac{q}{p}(P_i - P_{i-1})$$

In particular, $P_2 - P_1 = (q/p)(P_1 - P_0) = (q/p)P_1$ (since $P_0 = 0$), so that $P_3 - P_2 = (q/p)(P_2 - P_1) = (q/p)^2 P_1$, and more generally

$$P_{i+1} - P_i = (\frac{q}{p})^i P_1, \quad 0 < i < N.$$

Thus

$$\begin{aligned} P_{i+1} - P_1 &= \textstyle\sum_{k=1}^{i}(P_{k+1} - P_k) \\ &= \textstyle\sum_{k=1}^{i}(\frac{q}{p})^k P_1. \end{aligned}$$

giving

$$\begin{aligned} P_{i+1} &= P_1 + P_1 \textstyle\sum_{k=1}^{i}(\frac{q}{p})^k = P_1 \sum_{k=0}^{i}(\frac{q}{p})^k \\ &= \begin{cases} P_1 \dfrac{1 - (\frac{q}{p})^{i+1}}{1 - (\frac{q}{p})}, & \text{if } p \neq q; \\ P_1(i+1), & \text{if } p = q = 0.5. \end{cases} \end{aligned} \tag{3.2}$$

(Here we used the geometric series equation $\sum_{n=0}^{i} a^i = \dfrac{1 - a^{i+1}}{1 - a}$, for any number a and any integer $i \geq 1$.)

Choosing $i = N - 1$, and using the fact that $P_N = 1$, gives

$$1 = P_N = \begin{cases} P_1 \dfrac{1 - (\frac{q}{p})^N}{1 - (\frac{q}{p})}, & \text{if } p \neq q; \\ P_1 N, & \text{if } p = q = 0.5, \end{cases}$$

from which we get

$$P_1 = \begin{cases} \dfrac{1 - (\frac{q}{p})}{1 - (\frac{q}{p})^N}, & \text{if } p \neq q; \\ \dfrac{1}{N}, & \text{if } p = q = 0.5, \end{cases}$$

thus obtaining from Equ. (3.2)

$$P_i = \begin{cases} \dfrac{1 - (\frac{q}{p})^i}{1 - (\frac{q}{p})^N}, & \text{if } p \neq q; \\ \dfrac{i}{N}, & \text{if } p = q = 0.5. \end{cases} \tag{3.3}$$

3.4.1 The Gambler's Ruin Problem. Consider a gambler who starts with an initial fortune of \$1 and on each successive gamble either wins \$1 or loses \$1 with probabilities p and $q = 1 - p$ respectively. The gambler's objective is to reach a total fortune of N, without first being ruined (running out of money). If the gambler succeeds, then he's won the game and stops playing. He also stops when he's ruined, whichever happens first. There's nothing special about starting with \$1; more generally the gambler begins with i dollars where $0 < i < N$.

Let P_i be the probability that the gambler wins (i.e. reaches N) when he has i dollars. Thus $P_0 = 0$ (the gambler definitely does not win) and $P_N = 1$ (he wins).

We can rephrase this in terms of random walking. If the gambler starts with i dollars, then he is initially at $x = i$ on the line. The fortune awaits at $x = N$ and ruin at $x = 0$. This means that the outcome of a gamble is the same difference equation as for the random walk in Equ. 3.1:

$$P_i = pP_{i+1} + qP_{i-1}$$

and has the same solution: Equ. 3.3.

3.4.1.1 Checking the Equation. John starts with \$2, and $p = 0.6$. What is the probability that he reaches $N = 10$? The following values $i = 2$, $N = 10$ and $q = 1 - p = 0.4$, so $q/p = 2/3$, are plugged into Equ. 3.3:

$$P_2 = \frac{1 - (\frac{2}{3})^2}{1 - (\frac{2}{3})^{10}} = 0.57$$

We check this result experimentally using Listing 3.5 (GamTrials.py).

```
def gamble(p, start, N):
  t = 0; pos = start
  while (pos > 0) and (pos < N):
    r = random.choices(['win', '
        lose'], [p, 1-p])
    t += 1
    if r[0] == 'win': pos += 1
    else: pos -= 1
  return t, (pos == N)

p = float( input("p? "))
i, N = map(int, input("i, N? ").
    split())
ts = []; numWins = 0
for _ in range(NUM_TRIALS):
  time, isWin = gamble(p, i, N)
  if isWin: numWins += 1
  ts.append(time)
print(f"Win Prob: {(numWins/
    NUM_TRIALS):.2f}")
avg = statistics.mean(ts)
std = statistics.stdev(ts)
print(f"Average time: {avg:.1f};
    stdev: {std:.1f}")
```

Listing 3.5. The Gambler's Ruin

The user specifies values for p, i, and N, and the code run NUM_TRIALS (1000) times, recording the fraction of wins and the time taken for the game to finish (either with a win or ruin). For example, we can supply the values for John's game:

```
> python GamTrials.py
p? 0.6
i, N? 2 10
Win Prob: 0.55
Average time: 18.0; stdev: 16.3
```

The estimated winning probability is close to the theoretical value (0.55 vs. 0.57).

A graph is also drawn (see Fig. 3.8), showing the time to win/lose for each of the trials, with the average time as a red-dashed line. This illustrates that there's a very wide spread in the duration for the game, sometimes taken around 100 bets to finish.

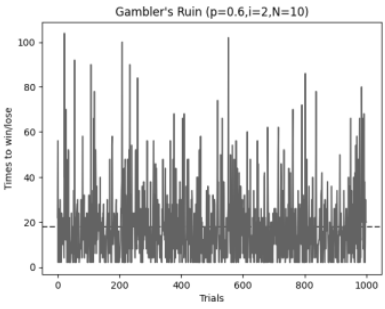


Figure 3.8. John's game; p = 0.6

If John's probability of winning is reduced even slightly below 0.5, the chances of him winning drop significantly. Equ. 3.3 gives $P_2 = 0.17$ when $p = 0.49$, and i and N are unchanged.

When Listing 3.5 is supplied with these inputs, it produces:

```
> python GamTrials.py
p? 0.49
```

```
i, N? 2 10
Win Prob: 0.16
Average time: 15.2; stdev: 18.6
```

The winning probability is again a close match to theory. Fig. 3.9 shows that although the average game duration is not much changed (from 18.0 to 15.2), the spread has become much wider, extending in some cases above 100 bets.

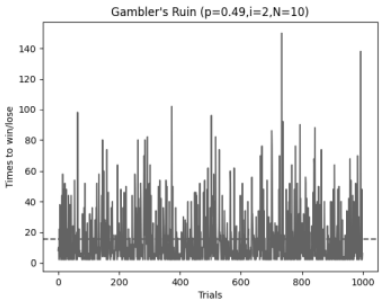


Figure 3.9. John's game; p = 0.49

3.4.1.2 Bound to Lose. Why is the gambler so unlikely to make money when the game is only slightly biased against him (i.e. when $p < q$)? We can identify two forces at work. First, the gambler's capital has random upward and downward swings from runs of good and bad luck. Secondly, and more importantly, the gambler's capital exhibits a steady, downward drift, because the negative bias means an average loss of a few cents on each $1 bet. The situation is illustrated in Fig. 3.10.

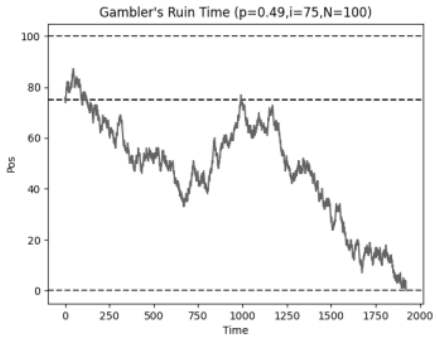


Figure 3.10. The downward drift of a $p < q$ random walk

If the gambler doesn't have a lucky, early, upward swing, then he's doomed since the required size of the upward swing becomes too large. It can be shown that as the size of the required swing grows, the odds that it occurs decrease exponentially.

Fig. 3.10 was generated by Listing 3.6 (GamTime.py).

```
def gambleTime(p, start, N):
  t = 0; ts = [t]
  pos = start; locs = [pos]
  while (pos > 0) and (pos < N):
    r = random.choices(['win', '
        lose'], [p, 1-p])
    t += 1
    ts.append(t)
    if r[0] == 'win':
      pos += 1
    else:
      pos -= 1
    locs.append(pos)
  return ts, locs
```

Listing 3.6. Timing the Gambler's Ruin

It only runs a single game for a specified p, i, and N, but plots the gambler's position changes during the course of that game. In the example above, $p = 0.49$, $i = 75$, and $N = 100$. Even though the player started very close to the winning position, the slightly biased probability was enough to lead him to ruin.

GamSurface.py attempts to visualize the drastic effects that even a slight bias against the gambler can cause, together with the player's starting position.

The code runs multiple games after the user supplies a winning position N, by varying the probability p between 0.4 and 0.6, and i in the range $[0..N]$. The winning fraction of games for NUM_TRIALS plays is collected, and plotted as a 3D surface (see Fig. 3.11).

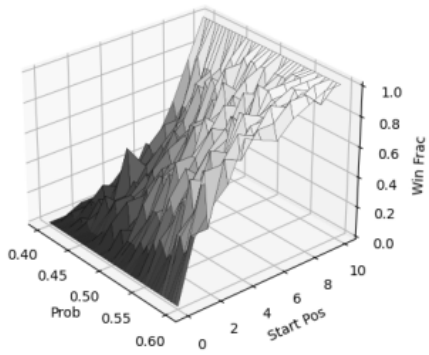
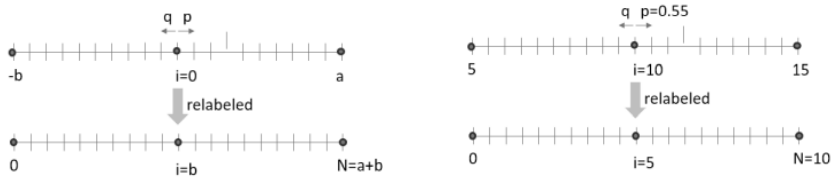


Figure 3.11. Probability and position

A small range of probabilities around 0.5 are enough to change a near-certain win to a near-certain loss, even when the gambler starts very close to the winning position. A look at the surface from different angles makes it clear that the relationship involves a hyperbolic curve, as suggested by our calculations.

3.4.2 Relabeling the Axis. Many random walk problems have the 'frog' starting at 0 and moving left to $-b$ or right to a. The question is usually phrased as 'what is the probability of reaching a before b?'.

This problem, and similar ones, can be relabeled as in Fig. 3.12a to make them more clearly variants of Equ. 3.3.



(a) Random walking from 0 (b) Playing the stock market

Figure 3.12. Number lines

The probability of reaching a before b when starting from $i = 0$ is the same as the probability of reaching $a + b$ before 0 when starting from $i = b$

$$P_b = \begin{cases} \dfrac{1-(\frac{q}{p})^b}{1-(\frac{q}{p})^{a+b}}, & \text{if } p \neq q; \\ \dfrac{b}{a+b}, & \text{if } p = q = 0.5. \end{cases}$$

As an example, imagine that Jane bought a stock share for \$10, and the stock price moves randomly with $p = 0.55$. What's the probability that Jane's stock reaches \$15 before \$5?

We want the probability that the stock goes up by 5 before going down by 5, which can be relabeled as in Fig. 3.12b.

The probability P_b, where $b = 5$:

$$P_b = \frac{1-(\frac{q}{p})^b}{1-(\frac{q}{p})^{a+b}} = \frac{1-(0.82)^5}{1-(0.82)^{10}} = 0.73$$

3.4.3 Gambling Variants. The main variant of the Gambler's Ruin problem is when there are *two* players competing against each other. There is a probability p that player A wins, and a probability of $1 - p$ that player B wins. Now suppose players A and B have have n_0 and m_0 dollars, respectively. At each trial, the loser pays the winner \$1. The game ends when one player is broke, so one player ends up with $n_0 + m_0$ dollars, and the other with nothing. This can be rephrased as a one player game (either A or B) who must reach a value $N = n_0 + m_0$ to win.

Another variation is when there is no winning value N, which permits the gambler to win any amount of money. Of course, he can still be ruined, which will stop the game. These changes are reflected in Equ. 3.3 by letting $N \to \infty$. If $p > 0.5$, then $q/p < 1$ and thus

$$\lim_{N\to\infty} P_i = 1 - (\frac{q}{p})^i > 0, \quad p > 0.5 \tag{3.4}$$

If $p \leq 0.5$, then $q/p \geq 1$ and then

$$\lim_{N\to\infty} P_i = 0, \quad p \leq 0.5 \tag{3.5}$$

Equ. (3.4) says that if $p > 0.5$, then there is a positive probability that the gambler will become infinitely rich. Equ. (3.5) says that if $p \leq 0.5$, then the gambler will certainly be ruined.

As an example, reconsider John from an earlier example (he starts with \$2, and $p = 0.6$). Under these new conditions, what's the probability that John will become disgustingly rich?

$$1 - (q/p)^i = 1 - (2/3)^2 = 5/9 = 0.56$$

Alternatively, if John started with \$1, what's the probability that he'll go broke? The probability he becomes infinitely rich is $1-(q/p)^i = 1-(q/p) = 1/3$, so the probability of ruin is 2/3.

Yet another variant is called *the Monkey on the Cliff*. A monkey is standing one step from the edge of a cliff and takes repeated independent steps: backward with probability p, or forward with probability q. Whereas the gambler moves on a line with two absorbing points (at $i = 0$ and $i = N$), the monkey only has a single end-point (the cliff edge, $i = 0$). This is actually the same as assuming that the gambler can become infinitely rich ($N \to \infty$).

3.4.4 Expected Duration of the Game. Let d_i be the expected duration of the standard game from a point when the gambler has i dollars. The boundary conditions are $d_0 = d_N = 0$, because $P_0 = 0$ and $P_N = 1$ means that the game is over.

The expected duration can be expressed as a difference equation for moving from position i to the two neighbors

$$d_i = p(1 + d_{i+1}) + q(1 + d_{i-1}) = 1 + pd_{i+1} + qd_{i-1}$$

The "1+" represents a unit time step which occurs when moving. Rearranging, and including the boundary conditions, we obtain:

$$pd_{i+1} - d_i + qd_{i-1} = -1 \quad \text{where } d_0 = 0, d_N = 0.$$

Since the right-hand side, -1, is nonzero, this is an inhomogeneous linear difference equation, which can be a little tricky to solve. The solution for $0 < i < N$

is:

$$d_i = \begin{cases} \frac{1}{q-p}(i - N\frac{1-(\frac{q}{p})^i}{1-(\frac{q}{p})^N}), & \text{if } p \neq q; \\ i(N-i), & \text{if } p = q = 0.5. \end{cases}$$

We can return to our earlier examples to see if the experimental data (e.g. in Figs. 3.8 and 3.10) tallies with the values produced by the d_i equation.

In the first case, $p = 0.6$, $i = 2$, and $N = 10$ results in $d_2 = 18$. On various runs, Listing 3.5 (GamTrials.py) reported an average time of 15.2, 15.3, and 19.6, which are fairly close, especially considering the wide spread in the game times.

Fig. 3.10 was generated with $p = 0.49$, $i = 75$, and $N = 100$ resulting in $d_{75} = 1970$, while the code reported 1917, so again fairly close agreement.

If we switch to the infinitely rich variant (i.e. $N \to \infty$), with $q > p$, the expected theoretical duration is

$$\lim_{N\to\infty} d_i = \lim_{N\to\infty} \frac{1}{q-p}(i - N\frac{1-(\frac{q}{p})^i}{1-(\frac{q}{p})^N}) = \frac{1}{q-p}(i-0) = \frac{i}{q-p},$$

since $(q/p)^N$ grows much quicker than N. This means that the gambler is certainly ruined, taking time $i/(q-p)$, on average.

In the case when $q = p$, we have

$$\lim_{N\to\infty} d_i = \lim_{N\to\infty} i(N-i) = \infty$$

So here, the gambler is still certainly ruined, but it may take forever.

3.5 Self Avoiding Random Walks

Fig. 3.13 shows a random path across a 11 x 11 square lattice taken from Donald Knuth's 'Mathematics and Computer Science: Coping with Finiteness' [**Knu96**].

A path begins at $(0, 0)$ (the lower left corner) and ends at $(10, 10)$ (the upper right corner), but is free to wander anywhere in between as long as no previously visited point is revisited.

The number of allowed alternative directions are written next to each point. For example, the 1's at the edges mean that there was only one way to go, since the other way is either already occupied or leads to a dead end. The probability that this particular path is obtained is the product of all the individual probabilities at each choice point, namely

$$\begin{aligned} \tfrac{1}{2}\cdot\tfrac{1}{2}\cdot\tfrac{1}{3}\cdot\tfrac{1}{3}\cdot\ldots\cdot\tfrac{1}{3}\cdot\tfrac{1}{1}\cdot\tfrac{1}{2} &= 2^{-34}2^{-24} \\ &= 1/48521024904413357015 04, \end{aligned}$$

or about one chance in 5×10^{21}.

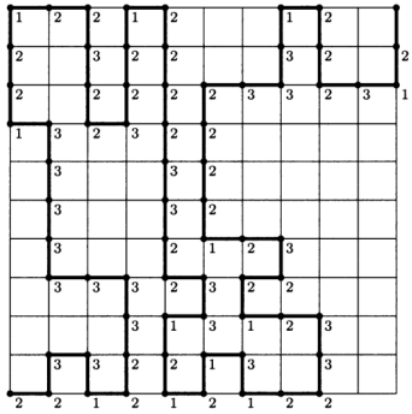


Figure 3.13. A numbered random path

How many paths are there from $(0,0)$ to $(10,10)$ which do not pass through the same point twice? With modern machines, this can be calculated, but real-size lattices quickly become too large to search in a feasible amount of time. However, physicists and chemists would very much like to know the number of self avoiding paths from $(0,0)$ to (m,n) to gain valuable insights into the structure of long molecules.

We can estimate this number by simulation. We start at $(0,0)$ and pick a path by going from node to node, choosing a branch at random at each node, until we reach $(10,10)$. If the number of branches at the i-th node is X_i, then the probability of obtaining this path is $p = 1/X$, where $X = X_1X_2...X_n$. Thus, the expectation of the random variable X is

$$E(X) = \sum_{\text{All paths}} 1 = \#\text{ of paths through the tree.}$$

This means that counting the number of paths is reduced to finding the average value N, by averaging the product of the number of choices along many paths.

Listing 3.7 (gridPaths.py) utilizes depth-first search (DFS) and three main data structures: a 2D list representing the grid, a list of coordinates for the path chosen between $(0,0)$ and $(10,10)$, and a list of visited points. The path and visited lists are not the same since backtracking may undo part of the path but those dropped points will remain in the visited list.

```
def findPath(showGrid=False):
  global grid, path, visited
  grid = [[' '
    for y in range(GRID_SIZE)]
      for x in range(GRID_SIZE)]
path=[(0,0)]; visited=[(0,0)]
nChoices = dfs(0, 0, 1)
```

```
  if showGrid:
    printPath(path)
    print(f"{len(path)}, {
        nChoices*1.0:.3g}, {1/
        nChoices:3g}")
  return path, nChoices


def dfs(r, c, nChoices):
  if atEnd(r,c):
    return nChoices
  possSteps = nextSteps(r,c)
  nSteps = len(possSteps)
  for (r1,c1) in possSteps:
    path.append((r1,c1))
    visited.append((r1,c1))
    nchs = dfs(r1, c1, nChoices*
        nSteps)
    if nchs != -1:
      return nchs
    else:
      path.remove((r1,c1))
  return -1

def atEnd(r, c):
  return (r == GRID_SIZE-1) and (
      c == GRID_SIZE-1)
```

```
def nextSteps(r, c):
  random.shuffle(DIRS)
  steps = [dirPos(r, c, dir) for
      dir in DIRS]
  return [ (r1,c1) for (r1,c1) in
       steps if isValid(r1,c1)]

def dirPos(r, c, dir):
  if dir == 'N':
    return (r-1, c)
  elif dir == 'E':
    return (r, c+1)
  elif dir == 'S':
    return (r+1, c)
  else:   # assume it's W
    return (r, c-1)

def isValid(r, c):
  if r < 0 or r >= GRID_SIZE:
    return False
  if c < 0 or c >= GRID_SIZE:
    return False
  return not ((r,c) in visited)
```

Listing 3.7. Calculating a path

One way to use the code is by calling findPath() within the Python IDE:

```
>>> from gridPaths import *
>>> findPath(True)
------------------------
| . .       . . . . . . |
| . . . .   .     . . . |
| .   . .   . . . . . . |
| . . . .   . . . . . . |
|       .   .         . |
|       .   x   . . . . |
|   . . .   . . . . . . |
| . . .     . . . . .   |
| . . . . .     .       |
| . . .   . .     . . .   |
| . . .   . .         . . |
------------------------
79, 4.3e+26, 2.32684e-27
([(0, 0), (0, 1), (1, 1), (1, 0), (2, 0), (3, 0), (3, 1),
(3, 2), (2, 2), (1, 2), (1, 3), (2, 3), (3, 3), (4, 3),
(5, 3), (6, 3), (6, 2), (6, 1), (7, 1), (7, 0), (8, 0),
  : # many more tuples, not shown
(4, 10), (5, 10), (6, 10), (6, 9), (5, 9), (5, 8), (5, 7),
```

```
(6, 7), (6, 8), (7, 8), (7, 7), (8, 7), (9, 7), (9, 8),
(9, 9), (10, 9), (10, 10)], 429767699937105647757164544)
>>>
```

Since the grid is implemented as a 2D list, the code treats $(0,0)$ as the *top-left* corner, and $(10,10)$ as the *bottom-right*. Also, the path drawing is a lot simpler than Knuth's Fig. 3.13. The three numbers (79, 4.3e+26, 2.32684e-27) are the length of the path, the number of choices made, and the probability (1/(number of choices)). The 'x' in the grid picture indicates that the path went through the center point (5,5).

Each call to findPath() will usually generate widely different results, such as a path containing only 49 points, with the number of choices reduced from 4.3e26 to 1.2e14.

The other way to use the code is to call the program from the command line, which makes it generate 10000 paths, then print the mean and standard deviations for the collected path lengths and number of choices:

```
> python gridPaths.py
ctr: 46.1%
len. mean = 55.05;  stdev = 16.74
choices. mean = 1.96e+27;  stdev = 3.9e+28
choices. median = 2.37e+18;  mode = 2.74e+15
```

The statistics for the number of choices highlight the enormous amount of variation in the results. It indicates that the mean is biased by the presence of extreme observations, which is why we've also included the median and the mode. A good course of action in a situation like this is to plot the data to see how it skews.

Knuth's results are considerably different from ours. He reports a total number of possible paths of $(1.6 \pm 0.3) \times 10^{24}$, and an average path length of 92 ± 5. The chance of hitting the center is 81 ± 10 percent of all the paths, whereas our result is around 46 percent.

Perhaps our sample size wasn't large enough, but we think it's more likely that our DFS algorithm differs in some way from Knuth's. For example, are paths that are discarded during backtracking still counted in the number of choices?

The number of paths from corner to corner on an 11 x 11 grid was computed by John Van Rosendale in 1981; the value turns out to be approximately 1.569e24.

Exercises

(1) Add plotting code to Listing 3.7 to help you track down the discrepancy between our results and Knuth's.

(2) The number of shortest paths from $(0,0)$ to $(10,10)$ is $\binom{20}{10} = 184756$. Estimate this number by means of Knuth's approach.

(3) The number of shortest lattice paths from $(0,0)$ to $(10,10)$ which may lie below or touch the line $y = x$, but not cross it, is $\frac{2}{11}\binom{20}{10} = 33592$. Estimate this number by means of Knuth's approach.

(4) An 8 x 8 chessboard can be covered by 32 2x1 dominoes in $2^4 901^2 = 12988816$ different ways. Estimate this number by means of Knuth's approach.

3.6 Finding Geometric Probabilities by Simulation

Finding exact solutions for problems involving geometric probabilities and expectations is often quite complicated, frequently requiring multiple integrals. Instead we'll repeat a random experiment multiple times to get an estimate of the probability or expectation through *simulation*.

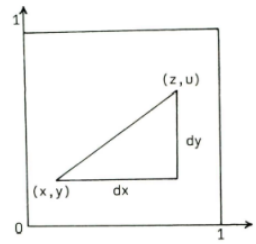


Figure 3.14. Distance between two points in a unit square

3.6.1 The Distance between Two Points in a Square. Two points are chosen at random in a unit square, and the expected distance between them is estimated. The coordinates x, y, z, u in Fig. 3.14 are chosen by means of Python's random() so that the points (x, y) and (z, u) are uniformly distributed in the square.

In Listing 3.8 (expdist1.py), we don't assign the x, y, z, u values to variables since we only need them to compute $dx$and dy. The sum variable accumulates all the n distances, and the average, sum/n, is a good estimate E_{est} of the expectation E.

```
n = int(input("n=? "))
sum = 0
for i in range(n):
  dx = random.random() - random.random()
  dy = random.random() - random.random()
  sum += math.sqrt(dx*dx + dy*dy)
print(f"expected dist == {sum/n:.6f}")
```

Listing 3.8. Points inside a square

When the program was run with $n = 10000$, typical output was $E_{est} = 0.523$.

The exact value calculated by L.A. Santalo [**San04**] is

$$E = \frac{\sqrt{2} + 2 + 5\ln(1+\sqrt{2})}{15} \approx 0.52141.$$

3.6.2 The Distance between Two Points in an Equilateral Triangle. This time the boundary values for the two points are a little more complicated. The idea is to choose two coordinates inside a unit square but count them only if they also lie inside the boxed equilateral triangle in Fig. 3.15.

If we translate the triangle so that vertex V moves to the origin O we get $y < -\sqrt{3}|x|$ for the shaded area. Now we translate back to $V = (\frac{1}{2}, \frac{1}{2}\sqrt{3})$ and get the equations for the two diagonal lines

$$y - \sqrt{3}/2 < -\sqrt{3}\,|x - \frac{1}{2}|.$$

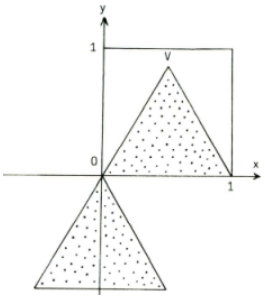


Figure 3.15. Distance inside an equilateral triangle

For $n = 10000$, Listing 3.9 (expdist2.py) gives the estimate $E_{est} = 0.362$.

```
n = int(input("n=? "))
sum = 0; d = math.sqrt(3)
for i in range(n):
  while True:
    x = random.random(); y = random.random()
    if y < d*(0.5 - abs(x - 0.5)):
      break
  while True:
    z = random.random(); u = random.random()
    if u < d*(0.5 - abs(z - 0.5)):
      break
  dist = math.sqrt((x-z)*(x-z) + (y-u)*(y-u))
  sum += dist
print(f"exp. dist. == {sum/n:.6f}")
```

Listing 3.9. Points inside a triangle

Santalo gives the exact value as

$$E = \frac{1}{5} + 3\ln\frac{3}{20} \approx 0.36479.$$

3.6.3 The Distance between Two Points in a Unit Circle.

This time it's simpler to use random.uniform(a, b), specifically random.uniform(-1,1). We generate points inside a 2x2 square centered at the origin, but only count those inside the unit circle (Fig. 3.16).

Listing 3.10 (expdist3.py) is based on this idea.

```
n = int(input("n=? "))
sum = 0
for i in range(n):
```

```
  while True:
    x = random.uniform(-1,1)
    y = random.uniform(-1,1)
    if x*x + y*y <= 1:
      break
  while True:
    z = random.uniform(-1,1)
    u = random.uniform(-1,1)
    if z*z + u*u <= 1:
      break
  dist = math.sqrt((x-z)*(x-z) + (y-u)*(y-u))
  sum += dist
print(f"exp. dist. == {sum/n:.6f}")
```

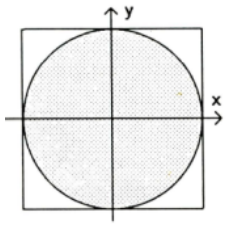


Figure 3.16. Distance inside a unit circle

Listing 3.10. Points inside a circle

For $n = 10000$, Listing 3.10 produces $E_{est} = 0.905$. Santalo gives the exact value as

$$E = \frac{128}{45\pi} \approx 0.90541.$$

3.6.4 Buffon's Needle. Buffon's needle is considered the first problem in geometric probability, dating from 1777.

In the plane, consider an infinite set of parallel lines spaced a units apart. Onto the plane we throw "at random" a curve C of arbitrary shape (Fig. 3.17). (Buffon used a needle of length a.)

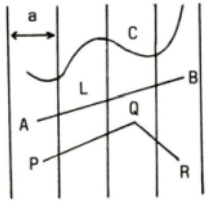


Figure 3.17. Curves and parallel lines

Let S be the number of intersections with the parallels, and we want to find $E(S)$, the expected number of intersections. For a line segment AB of length L we set $E(S) = f(L)$ with an as yet unknown function f. Now throw the broken line PQR onto the parallels. If PQ has S_1 intersections and QR has S_2 intersections, then $S = S_1 + S_2$. In other words, the expectation of the sum of two random variables is the sum of their expectations, even if those variables are not independent. Thus we have $E(S) = E(S_1 + S_2) = E(S_1) + E(S_2)$, or

$$f(L_1 + L_2) = f(L_1) + f(L_2). \tag{3.6}$$

Also, f is additive, and increasing, so must be linear

$$f(L) = L * f(1). \tag{3.7}$$

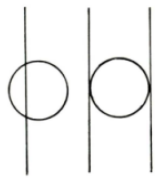
The additive property for expectations tells us that the expected number of intersection with any polygon can be expressed as Equ. (3.7), where L is the length of the polygon. Since any "normal" curve can be approximated by a broken line to any accuracy, then Equ. (3.7) is actually valid for a curve of any shape and length

L. So take a circle of diameter a and length $L = \pi a$, which has the useful property that it always has two intersections (Fig. 3.18a) with the parallels, i.e.

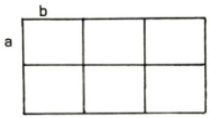
$$f(\pi a) = \pi a f(1) = 2 \quad \rightarrow \quad f(1) = \frac{2}{\pi a}$$

Combining this with Equ. (3.7) produces:

$$f(L) = \frac{2L}{\pi a} \tag{3.8}$$



(a) Circles intersecting lines



(b) Space divided into rectangles

Figure 3.18. Buffon's needles

Now suppose the plane is divided into congruent rectangles with sides a, b (see Fig. 3.18b). Onto this plane we throw a curve of length L. If there are S_1 intersections with the verticals and S_2 intersections with the horizontals, with $S = S_1 + S_2$, then

$$E(S) = E(S_1) + E(S_2) = \frac{2L}{\pi a} + \frac{2L}{\pi b} = \frac{2L}{\pi}(\frac{1}{a} + \frac{1}{b}).$$

For $a = b = L = 1$, we get

$$E(S) = 4/\pi, \text{ or } \pi = \frac{4}{E(S)}$$

If we estimate $E(S)$ by means of

$$E_{\text{est}} = \frac{\text{intersections}}{\text{throws}} = \frac{S}{T}$$

then we get an estimate for π

$$\pi_{\text{est}} = \frac{4T}{S}.$$

3.6.4.1 Implementing Buffon's Needle. Fig. 3.19 illustrates a typical situation after dropping a needle of unit length onto a unit square. We imagine that the center of the needle is at (x, y) relative to the bottom-left corner of the square, and the needle is rotated clockwise from the vertical by a radians.

Throwing this needle "at random" can be coded using:

```
x = random.random()
y = random.random()
angle = math.pi*random.random()
```

The coordinates of the center of the needle are random numbers with a uniform distribution between 0 and 1, and its angle with the northern direction, counted to the right, is randomly between 0 and π.

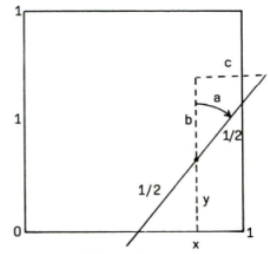


Figure 3.19. A needle intersecting a square

The throwing area can be considered to be an infinite grid of unit squares, but its periodic pattern ensures that the result will be the same if we pick a center with a uniform distribution from a single square.

The generated plot (Fig. 3.20) shows that buffon.py converges (slowly) towards a single probability. It produces a poor estimate for π of 3.14398 after one million throws, correct only to 2 decimal places.

The needle's horizontal and vertical intersections are observed by checking if its end points are in the same square or not. This is done by truncating the x- and y- coordinates to integers. One slight complication is that if a needle is off the left side (or bottom) of the square, then its x- (or y-) coordinate should be truncated to -1. This behavior is implemented by truncL().

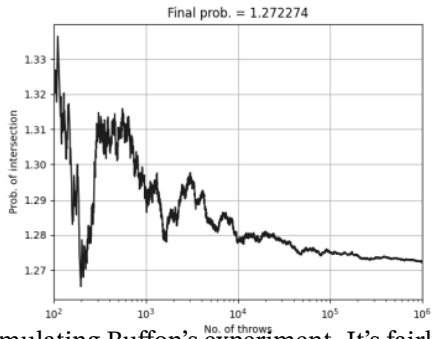


Figure 3.20. Plot of needle intersection probs

buffon.py is really only useful for simulating Buffon's experiment. It's fairly worthless for computing π since it has to use π in its calculations. Nevertheless, this problem has some interesting applications in stereology, as explained by Solomon [**Sol78**].

A better needle experiment is described in Ex. 7 below.

Exercises

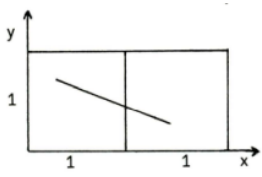
(1) Listing 3.11 (expdist3b.py) shows another way of selecting pairs of points in the unit circle for calculating their average distance apart. Is this method faster than the one in Listing 3.10?

```
n = int(input("n=? "))
sum = 0
for i in range(n):
```

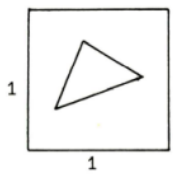
```
  while True:
    x = random.random();
    y = random.random()
    if x*x + y*y <= 1:
      break
  while True:
    z = random.random();
    u = random.random()
    if z*z + u*u <= 1:
      break
  if random.randint(0,1) == 0:
    z = -z
  if random.randint(0,1) == 0:
    u = -u
  dist = math.sqrt(
    (x-z)*(x-z) + (y-u)*(y-u))
  sum += dist
print(f"exp. dist. v2 == {sum/n:.6f}")
```

Listing 3.11. Points inside a circle (2nd approach)

(2) A point is chosen at random in each of the two squares in Fig. 3.21a. Find by simulation the mean distance between the two. (It is possible to calculate $E \approx 1.08814$.)



(a) Distance between two points



(b) Three points forming an obtuse triangle

Figure 3.21. Selecting points

(3) Three points are chosen at random inside the unit square in Fig. 3.21b. Let P be the probability that they form an obtuse-angled triangle, and obtain a value for P by simulation. It can be shown, though with great labor, that $P = 97/150 + \pi/40 \approx 0.725$.

(4) Four points, P_1, P_2, P_3, P_4, are chosen at random inside a unit cube. Find by simulation the expected volume of the tetrahedron $P_1P_2P_3P_4$. In this case we don't know the exact value of the expected volume.

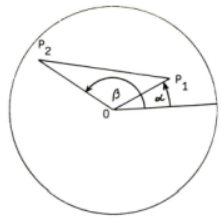
(5) Abby suggests to Carl: Let's choose two points P and Q at random inside a unit circle. If $|PQ| \leq 1$ then I win, otherwise you win. What's Abby's chance of winning? Estimate the answer using either Listing 3.10 or Listing 3.11, but theory predicts that Abby will win with probability $1 - (3\sqrt{3}/4)/\pi \approx 0.5865$.

(6) Pick two random angles, α and β, then measure out random distances in their directions from the origin to P_1 and P_2 (see Fig. 3.22a). Find the expected distance between P_1 and P_2, which will *not* agree with the distance calculated in Ex. 3. Why not?

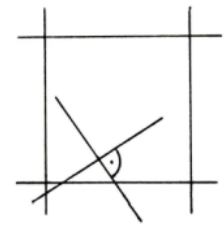
If, instead of averaging the distance

$$|P_1P_2| = \sqrt{r_1^2 + r_2^2 - 2r_1r_2\cos(\beta - \alpha)},$$

we find the average of the quantity $\sqrt{r_1 + r_2 - 2\sqrt{r_1r_2}\cos(\beta - \alpha)}$, we do get the same answer as in Ex. 3. Why?



(a) Distance between two points inside a circle



(b) A cross intersecting a rectangle

Figure 3.22. Buffon exercises

(7) Two needles of unit length are tied together at their centers to form a cross, then thrown onto a chessboard made up of unit squares (see Fig. 3.22b). This will give us a better estimate of π than using one needle, which you should check by writing a program similar to `buffon.py`.

(8) Find the mean distance between two points chosen at random in the unit cube.

Theory predicts (see 'E2629', David P. Robbins and Theodore S. Bolis, *American Mathematical Monthly* 1978, p. 277-278, `https://www.jstor.org/stable/i315089`):

$$E = \frac{4 + 17\sqrt{2} - 6\sqrt{3} + 21\ln(1 + \sqrt{2}) + 42\ln(2 + \sqrt{3}) - 7\pi}{105} \approx 0.661707.$$

(9) **Friends Meeting**. Suppose two friends decide to meet between 8 pm and 9 pm. When one of them arrives at the meeting spot, he waits for 10 minutes for the other to show up. For instance, if Friend 1 arrives at 8:40, he'll wait until 8:50 for Friend 2. Friend 1 doesn't know if Friend 2 had shown up earlier and left (and the same is true for Friend 2). However, both friends know that if they arrive at a time near to 9pm, say 8:55, that they only need to wait until 9:00, not 9:05.

What's the probability that the two friends meet?

3.7 The Bertrand Paradox

Joseph Bertrand introduced this problem in *Calcul des probabilités* in 1889: draw an equilateral triangle inside a circle, then add a chord to the circle at random. What is the probability that the chord is longer than a side of the triangle?

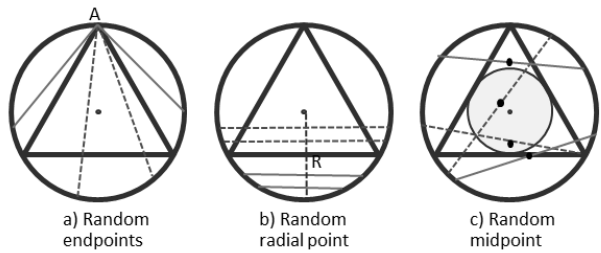


Figure 3.23. Three ways to select chords

The problem statement is vague enough that Bertrand produces three different, valid, answers:

(1) *The random endpoints method*. Choose a random start point for the chord, called A, anywhere on the circumference. The other end can be placed anywhere else on the circumference, samples of which are shown in Fig. 3.23a as dashed and solid lines.

The equilateral triangle can be oriented in any direction, so position it to ensure that one vertex coincides with A. Now it's clear that only those chords that cut across the triangle are longer than its side (the dashed lines).

The angle of the equilateral triangle at A is 60°, and all the chords lie within a 180° range, so the chances of drawing a chord larger than the side of the triangle must be $\frac{1}{3}$.

(2) *The random radial point method*. Choose a point at random on the radius line R of the circle, and construct a chord through this point, perpendicular to the radius. Samples of possible chords are shown in Fig. 3.23b.

The equilateral triangle can be oriented in any direction, so position it to ensure that one side is parallel to the chords. Now it's clear that a chord will only be longer than this side if it's nearer the center of the circle (the dashed lines) than the side.

The triangle's side bisects the radius line, therefore the probability that a random chord on the line is longer than that side is $\frac{1}{2}$.

(3) *The random midpoint method.* Choose a point anywhere within the circle and construct a chord with that point as its midpoint. Possible chords are shown in Fig. 3.23c as dashed and solid lines.

A chord will only be longer than a triangle side if its midpoint lies within the circle inscribed inside the triangle. This pale-gray circle has a radius 1/2 that of the larger circle, so its area is one fourth the area of the larger one ($\frac{1}{2}^2$). Therefore the probability a chord is longer than a side of the triangle is equal to the chance its midpoint is inside the small circle's area, $\frac{1}{4}$.

Two aspects of the preceding arguments may be a little unclear: why the radius line R is bisected by the triangle side in Fig. 3.23b, and why the inscribed circle in Fig. 3.23c is half the radius of the main circle. Fig. 3.24 shows why these two statements are true.

A standard theorem for circles is that when two angles are subtended by the same chord, then the angle at the center is twice the angle at the circumference. This is the case for chord GH in Fig. 3.24, which subtends angles at C and F. So angle GCH is twice that of GFH, and since GFH is 60°, then GCH is 120°. The triangle GCH is isosceles (the lines GC and HC are the same length), so the angles CGH and CHG must be the same, which implies they are each 30°. Switching to trigonometry, we can calculate the length $CD = CH\sin(30) = \frac{R}{2}$. CE's length is R, which means that GH bisects CE at D.

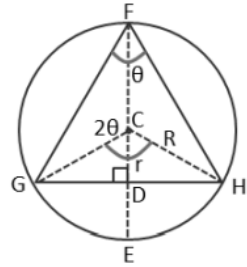


Figure 3.24. An equilateral triangle inside a circle

The inscribed circle in Fig. 3.23c has the equilateral triangle's sides as tangents, which means that its radius is equal to CD in Fig. 3.24, which we have just determined to be $\frac{R}{2}$.

We began by mentioning that Bertrand's problem statement was 'vague enough' to allow three different solutions. The vagueness resides in the meaning of choosing a chord "at random". The three solutions correspond to different selection methods.

Martin Gardner's article on the problem (see Chapter 19 in [**Gar61**]) suggests informative analogies for these methods:

(1) Two spinners are mounted at the center of a circle. They rotate independently. We spin them, mark the two points at which they stop, and connect the points with a straight line.

(2) A large circle is chalked on the sidewalk. We roll a broom handle toward it from some distance away, until the handle stops somewhere on the circle.

(3) We paint a circle with molasses and wait until a fly lands on it; then we draw a chord whose midpoint is the fly.

The difference in the selection methods can be visualized by drawing the generated chords and their midpoints. Listing 3.12 (bertrand.py) produces the six circular diagrams in Fig. 3.25, two for each method. The circles along the top row show the distribution of the chords, while the bottom row highlights the midpoints of those chords.

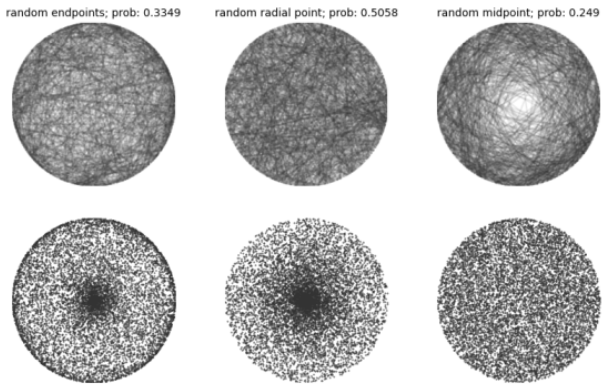


Figure 3.25. Plots of the three Bertrand methods

The diagrams are created using matplotlib's support for drawing circles and lines. We won't explain that code since it's not really relevant to the math problem, but we will show the implementation of the three chord generation methods. This code is based on a numpy solution by Christian Hill at `https://scipython.com/blog/bertrands-paradox/`.

```
def bertrand1():
  chords = []
  midPts = []
  for i in range(NUM_CHORDS):
    angle1 = random.random()*2*math.pi
```

```
    angle2 = random.random()*2*math.pi
    chord = ( R*math.cos(angle1), R*math.sin(angle1),
              R*math.cos(angle2), R*math.sin(angle2) )
    chords.append(chord)
    midPt = ((chord[0]+chord[2])/2, (chord[1]+chord[3])/2)
    midPts.append(midPt)
  return "random endpoints", chords, midPts

def bertrand2():
  angles = [ random.random()*2*math.pi
                              for i in range(NUM_CHORDS)]
  radii =  [ random.random()*R for i in range(NUM_CHORDS)]
  midPts = []
  for i in range(NUM_CHORDS):
    midPt = (radii[i] * math.cos(angles[i]),
             radii[i] * math.sin(angles[i]))
    midPts.append(midPt)
  chords = getChords(midPts)
  return "random radial point", chords, midPts

def getChords(midPts):
  # Return the chords with the provided midPts
  chords = [(0,0,0,0) for i in range(NUM_CHORDS)]
  for i, (x0, y0) in enumerate(midPts):
    m = -x0/y0
    c = y0 + x0**2/y0
    A, B, C = m**2+1, 2*m*c, c**2-R**2
    d = math.sqrt(B**2 - 4*A*C)
    xEnd1 = (-B + d)/ 2 / A
    xEnd2 = (-B - d)/ 2 / A
    yEnd1 = m*xEnd1 + c
    yEnd2 = m*xEnd2 + c
    chords[i] = (xEnd1, yEnd1, xEnd2, yEnd2)
  return chords

def bertrand3():
  angles = [ random.random()*2*math.pi
                             for i in range(NUM_CHORDS)]
  radii =  [ math.sqrt(random.random())*R
                             for i in range(NUM_CHORDS)]
  midPts = []
  for i in range(NUM_CHORDS):
    midPt = (radii[i] * math.cos(angles[i]),
             radii[i] * math.sin(angles[i]))
    midPts.append(midPt)
  chords = getChords(midPts)
  return "random midpoint", chords, midPts
```

Listing 3.12. The Bertrand methods

getChords() is a little complicated since it solves a quadratic equation involving the intersection of a line (representing a chord) and a circle (see Fig. 3.26).

The equation for the line, $y = mx + c$, is squared to give $y^2 = m^2x^2 + c^2 + 2mxc$, then used to replace y^2 in the circle's equation $x^2 + y^2 = R^2$. After a little rearranging, the resulting quadratic in x is $(m^2 + 1)x^2 + 2mcx + (c^2 - R^2) = 0$ which is solved in the usual way.

The drawing code keeps a tally of the number of chords whose length exceeds the triangle's side, which is used to calculate a probability for each method. These are included in the titles in Fig. 3.25, and approximate the theoretical results.

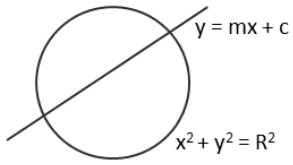


Figure 3.26. Equations of a line and a circle

3.8 The Broken Stick Problem

Suppose we randomly break a stick in two places. What's the probability that we can form a triangle using the pieces?

We'll see that there are two different valid answers, depending on the exact meaning of 'randomly break'.

3.8.1 Independent Breaks. The first scenario relies on the two breaks being independent. You could imagine this as the breaks being made by two blind-folded people at the same time.

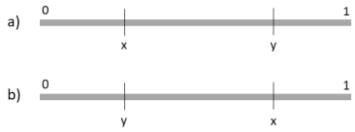


Figure 3.27. Two positions for x and y

We can assume, without loss of generality, that the stick is of unit length. Let the first cut be at position x and the second at y, which allows for two interpretations shown in Fig. 3.27.

The *triangle inequality* condition for making a triangle from three lengths a, b, and c applies to each of the three sides:

$$\begin{aligned} a + b &> c \\ b + c &> a \\ c + a &> b \end{aligned}$$

In Fig. 3.27a), the three lengths are x, $y - x$, and $1 - y$. Plugging these into the triangle inequalities produces:

$$\begin{aligned} x + y - x &> 1 - y \\ y - x + 1 - y &> x \\ 1 - y + x &> y - x \end{aligned}$$

These can be simplified to $y > \frac{1}{2}$, $x < \frac{1}{2}$, and $y - x < \frac{1}{2}$, which can be visualized as lines defining regions inside a unit square (see Fig. 3.28a). The unit square comes from the fact that x and y can have any value in the range $[0, 1]$. The region A where all three inequalities are true equals the probability $\frac{1}{8}$.

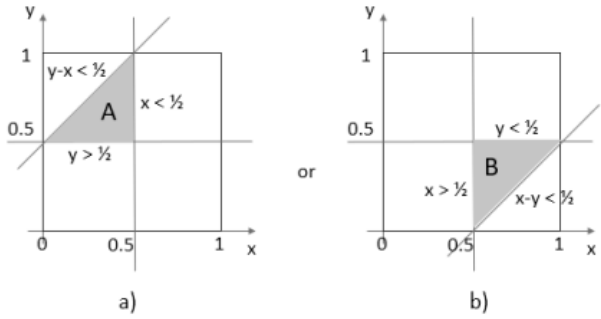


Figure 3.28. Probabilities as areas

We now repeat this process for Fig. 3.27b. The three lengths are y, $x - y$, and $1 - x$. Substituted into the triangle inequalities, we get:

$$\begin{aligned} y + x - y &> 1 - x \\ x - y + 1 = x &> y \\ 1 - x + y &> x - y \end{aligned}$$

These can be simplified to $x > \frac{1}{2}$, $y < \frac{1}{2}$, and $x - y < \frac{1}{2}$ which defines a region B in Fig. 3.28b which equals the probability $\frac{1}{8}$.

The total probability of obtaining a triangle is a sum of the two regions, $\frac{1}{4}$.

Listing 3.13 (sticks.py) simulates this process of triangularity testing, with the crucial part being the generation of the x and y lengths, which are independent.

```
def isTriangle(a, b, c):
  return (a + b > c) and (a + c > b) and (b + c > a)

numTris = 0
for i in range(NUM_TRIALS):
  x = random.uniform(0, 1)
  y = random.uniform(0, 1)    # 1st scenario
  # y = random.uniform(x, 1)  # 2nd scenario
  if x > y:
    x, y = y, x
  if isTriangle(x, y-x, 1-y):
    numTris += 1
```

```
print(f"Estimated prob = {(numTris/NUM_TRIALS):0.6f}")
```

Listing 3.13. Making a triangle (scenario 1)

The output confirms the theoretical probability:

```
> python sticks.py
Estimated prob = 0.249740
```

3.8.2 Dependent Breaks. Breaking the stick in a dependent manner can be achieved by carrying out the cuts in order: first you break off and take away the left piece, then you break the remaining piece into two. If we call the two break positions x and y again, then we end up with three pieces like those in Fig. 3.27a, but y's position depends on x's.

The lengths are x, $y - x$, and $1 - y$, and applying the triangle inequality, we get $x < \frac{1}{2}$, $y > \frac{1}{2}$, and $y - x < \frac{1}{2}$ (the same inequalities as for region A in Fig. 3.28a). The last equation is also rearranged to $y < x + \frac{1}{2}$. The first equation states that x must range between 0 and $\frac{1}{2}$ and the second and third equations imply that y must range between $\frac{1}{2}$ and $x + \frac{1}{2}$.

Therefore, the three pieces can form a triangle if x is in the interval $(0, \frac{1}{2})$ and y is in $(\frac{1}{2}, x + \frac{1}{2})$. Furthermore, y is guaranteed to be in the interval $(x, 1)$, and y's required interval $(\frac{1}{2}, x + \frac{1}{2})$ must be smaller.

Thus, for a given x, the *probability* of y being in the required range is the ratio of the length of its required range $(x + \frac{1}{2} - \frac{1}{2} = x)$ to the length of its guaranteed range $(1 - x)$. The resulting probability equation is $\frac{x}{1-x}$.

Since x can vary between 0 and $\frac{1}{2}$. The total probability is the sum of all the possible probabilities, which we can obtain by integrating the probability equation over x's limits.

$$P(\text{triangle}) = \int_0^{\frac{1}{2}} \frac{x}{1-x} dx$$

The integral is

$$[\, 1 - x - \ln|1 - x| \,]_0^{\frac{1}{2}} = \ln 2 - \frac{1}{2} \approx 0.1931$$

Checking out this result with Listing 3.13 requires the replacement of the line of code that generates y by:

```
y = random.uniform(x, 1)
```

This makes it quite explicit that y's value depends on x's. When the modified program is run, it returns a result similar to our calculated probability:

```
> python sticks.py
Estimated prob = 0.191490
```

3.9 The Monty Hall Problem

Monty Hall hosts a game show in which a contestant wins whatever is behind a door that is chosen from three closed doors. A car is behind one and a goat is behind each of the other two.

Once the contestant has chosen a door, Monty doesn't open it but instead opens a different door to reveal a goat. Monty then gives the contestant the choice of staying with their original choice or switching to the remaining unopened door. Should the contestant stay or switch to maximize the chance of winning the car?

Most people would think that the chance of winning after the goat is revealed is 50%, and so switching won't make a difference. They're wrong.

There are some rules that determine which door Monty can open after the contestant has made a choice.

- Monty knows which door hides the car.
- Monty doesn't open the door chosen by the contestant.
- Monty always offers the contestant a chance to switch.
- If more than one 'goat' doors could be opened, Monty chooses one at random.

Before we start coding, let's see how the problem can be represented with probabilities.

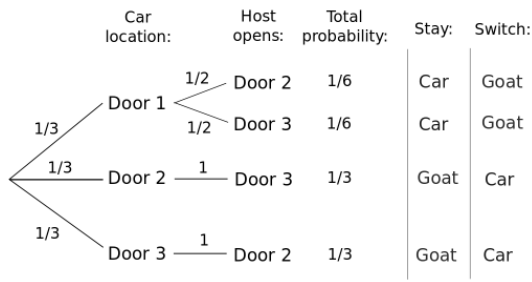


Figure 3.29. The probability of every outcome

The probabilities are summarized by the decision tree in Fig. 3.29. It highlights two places where our intuition may fail us – first there are *four* scenarios to consider, and they do *not* have equal probabilities of occurring.

We now consider the outcome if the contestant switches. The result is positive (win the car) in only the last two cases, but their total probability is 2/3, so it is worthwhile to change.

Listing 3.14 (montyhall.py) simulates a large number of games and keeps track of the probabilities of winning with, and without, switching doors.

```
trials = range(NUM_ITERS)
round1 = range(NUM_DOORS)  # door choices in round 1
stayCount, chgCount = 0, 0
stayProbs = []; chgProbs = []
for i in trials:
  carDoor = random.choice(round1)
  playerPick1 = random.choice(round1)
  montyChoices = [n for n in round1
                  if n!= carDoor and n!= playerPick1]
  goatDoor = random.choice(montyChoices) # what Monty opens
  round2 = [n for n in round1
                  if n!= playerPick1 and n!=goatDoor]
  playerPick2 = random.choice(round2)
  if playerPick1 == carDoor:
    stayCount += 1    # would win by staying
  if playerPick2 == carDoor:
    chgCount += 1     # would win by switching
  p1 = stayCount/(i+1)
  p2 = chgCount/(i+1)
  stayProbs.append(p1)
  chgProbs.append(p2)
```

Listing 3.14. Simulating the Monty Hall problem

Fig. 3.30 shows that the switch and stay probabilities gradually approach 2/3 and 1/3.

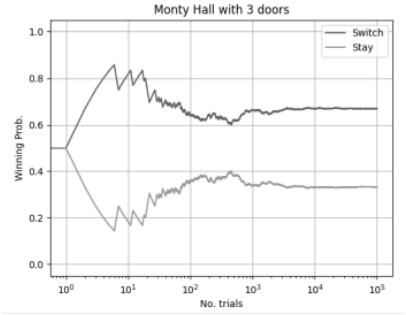


Figure 3.30. Monty Hall simulated

3.9.0.1 History. The earliest puzzle related to the Monty Hall problem is Bertrand's box paradox, posed by Joseph Bertrand in 1889 in his *Calcul des probabilités*. There are three boxes: one containing two gold coins, a box with two silver coins, and a box with one of each. After choosing a box at random and withdrawing one coin at random, which happens to be gold, what's the probability that the other coin is gold?

A somewhat better known antecedent is the Three Prisoners problem, first published in Martin Gardner's "Mathematical Games" column in *Scientific American* in 1959 [**Gar61**].

3.9.0.2 A Generalization. We finish with a variant of the puzzle suggested in Blom's excellent *Problems and Snapshots from the World of Probability* [**BHS93**]: person A is invited by person B to play a game with an urn containing w white balls and b black balls ($w + b \geq 3$). Person A is blindfolded before beginning, then asked to choose between two strategies:

(1) Player A draws a ball at random. If it's white, then he wins the game, otherwise he loses.

(2) Player A draws a ball and throws it away. Player B removes a black ball. A again draws a ball. If it's white, he wins the game, otherwise he loses.

Which strategy is best for Player A? Clearly, if A chooses Strategy 1, he wins with probability

$$P_1 = \frac{w}{w+b}.$$

If A chooses Strategy 2, then there are two paths to consider – A throws away a white ball or a black ball. In each of these, B always removes a black ball before A tries again. This means that there are now two less balls (i.e. $w + b - 2$), but in case 1 there are $w - 1$ white balls, while in case 2 there are w. This can be expressed as:

$$\begin{aligned} P_2 &= \frac{w}{w+b} \cdot \frac{w-1}{w+b-2} + \frac{b}{w+b} \cdot \frac{w}{w+b-2} \\ &= \frac{w}{w+b}\left(1 + \frac{1}{w+b-2}\right) \end{aligned}$$

Since $P_2 > P_1$, Strategy 2 is the best one for A.

In Monty Hall, we have $w = 1$, $b = 2$, and so $P_1 = 1/3$ and $P_2 = 2/3$.

3.10 Random Choice of an s-Subset from an n-Set

An s-sized sample is chosen at random from an n-sized population if each sample of that size has the same probability $1/\binom{n}{s}$ of being chosen. This is an interesting statistical problem, and we'll look at four elementary algorithms that address it.

3.10.1 Algorithm 1. Listing 3.15 (`sample1.py`) implements one of the more elegant solutions. For each i from 1 to n it calls randint(0, $n - 1$). If the result is $< s$, then it prints i and decreases s by 1. For each call it also decreases n by 1.

```
s,n = map(int, input("s n=? ").split())
i = 1
while n > 0:
  if random.randint(0, n-1) < s:
    s -= 1
    print(i, end=' ')
  i += 1; n -= 1
```

Listing 3.15. Sampling from a set

Examples:

```
> python sample1.py
s n=? 10 100
5 17 36 49 53 72 77 84 95 100
> python sample1.py
s n=? 5 300
3 43 90 159 206
```

The algorithm has time complexity $O(n)$ since randint() is called n times, and has the advantage that the sample is printed in increasing order.

If n is the number of elements still to be considered and s the number of elements still to be chosen, then the next number will be printed with probability s/n. This isn't easy to understand, and it helps to trace the first few iterations of the code's while-loop.

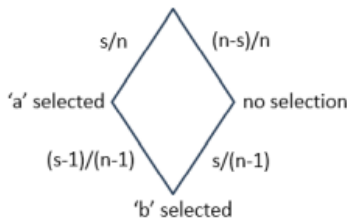


Figure 3.31. Probabilities for reaching the second selection

The first selection is obviously printed with the correct probability $p_1 = s/n$. It requires more thought to see that the next selected value is printed with the same likelihood. Consider the two paths through Fig. 3.31. The left path is the probability for getting $\{a, b\}$ while the right-hand path is for $\{b\}$ (i.e. no selection was made during the first loop). We combine the paths in Fig. 3.31 to get:

$$p_2 = \frac{s}{n} * \frac{s-1}{n-1} + \frac{n-s}{n} * \frac{s}{n-1} = \frac{s}{n(n-1)} * (s-1+n-s) = \frac{s}{n}.$$

3.10.2 Algorithms 2 and 3. Listing 3.16 (sample2.py) is more efficient than algorithm 1 if s is a lot smaller than n, and it's also perhaps easier to understand.

```
s,n = map(int, input("s n=? ").split())
x = [0]*n
for i in range(s):
  while True:
    r = random.randint(0,n-1)
    if x[r] == 0: break
  print(r, end=' '); x[r] = 1
```

Listing 3.16. Sampling from a set (2/3)

```
> python sample2.py
s n=? 10 100
13 92 67 12 64 1 44 86 97 6
```

It chooses a random number r from 0 to $n-1$, and checks if it's already been chosen. If it has then $x[r] = 1$, which causes the code to repeat the selection. Otherwise r is printed, and $x[r]$ is set to 1 to record that r has been chosen.

One drawback is that the numbers are printed in their selected, random order.

sample3.py implements a one-line change to Listing 3.16 to print the s elements in sorted order:

```
print(*[i for i in range(n) if x[i] == 1])
```

The result:

```
> python sample3.py
s n=? 10 100
4 6 7 11 13 25 33 66 75 84
```

3.10.3 Algorithm 4. Listing 3.17 (sample4.py) employs a technique corresponding to drawing an s-sized sample from n numbers *without* replacement.

```
s,n = map(int, input("s n=? ").
    split())
x = [i for i in range(n)]
for i in range(s):
  r = i + random.randint(0, n-1-i)
  x[i], x[r] = x[r], x[i]
  print(x[i], end=' ')
```

```
> python sample4.py
s n=? 10 100
78 63 73 94 47 55 61 90 81 13
```

Listing 3.17. Sampling from a set (4)

It chooses a random number r from 0 to $n-1$ and exchanges $x[r]$ with $x[0]$. Then it selects a random number r from **1** to $n-1$ and exchanges $x[r]$ with $x[1]$,..., and continues in this way until it picks a random number r from **s-1** to $n-1$ and exchanges $x[r]$ with $x[s-1]$. The resulting subsequence in $x[0]$ to $x[s-1]$ is out of order.

The standard reference for these algorithms in Vol. 2 of *The Art of Computer Programming*, [**Knu97**], specifically Section 3.4.2.

3.10.4 Python's sample(). Python's random module contains a sample() function which can return a k length list of elements chosen from a supplied sequence, implementing sampling without replacement in a similar way to Listing 3.17.

To choose a sample from a range of integers, it's especially fast and space efficient to use range(), as in:

```
>>> import random
>>> random.sample(range(10000000), 10)
```

```
[1554429, 5899448, 1217734, 7432044, 1568457, 8183569,
 4173744, 7896302, 6848341, 1777236]
```

Exercises

(1) **Generating a random permutation**. n numbers are stored in the list x. Write a program which can permute the numbers while ensuring that all the possible permutations have the same probability of occurring. There is a suitable function for doing this in the random module, but you'll benefit from coding a version yourself.

(2) Generate a random permutation of x, and record the length k of the longest "saw-tooth" subsequence $x[p+1] < x[p+2] > x[p+3] < ... > x[p+k]$ (with teeth going upwards and the sequence ending with a last downward value).

3.11 Coin Tosses in Software

In this and the next two sections we'll examine some unusual methods for generating random digits.

3.11.1 Generating Random Binary Digits. We compute $z = \log_b a$ with $1 < a < b$. By definition this is the solution x for

$$b^x = a, \quad 0 < x < 1.$$

Let

$$x = 0.d_1d_2d_3... = d_1/2 + d_2/4 + d_3/8 + ...$$

be the binary representation of x. Then

$$b^{d_1/2+d_2/4+d_3/8+...} = a. \tag{3.9}$$

Square both sides and update a to $a * a$:

$$b^{d_1+d_2/2+d_3/4+...} = a.$$

If $a < b$ then $d_1 = 0$. Otherwise $d_1 = 1$ so update a to a/b. In both cases we get

$$b^{d_2/2+d_3/4+...} = a. \tag{3.10}$$

This is again Equ. (3.9) with the first digit shaved off. Thus we have a simple algorithm for printing successive binary digits of x which becomes Listing 3.18 (genBin.py).

```
def genBinary(a, b, i):
  while i >= 1:
    a = a*a
    if a < b:
      print("0", end='')
    else:
      a = a/b
      print("1", end='')
    i = i-1
  print()
```

Typical output:

```
> python genBin.py
a b? 2 10
01001101000100000100110101 0
0001001111101111001111111111
    : # many more lines
00111010100010000000111010 11
01111110001110010100011
```

Listing 3.18. Generating random binary digits

It's possible to check this output by remembering that $b^x = a$. So we need to calculate x in $10^x = 2$, or $x = \log_{10} 2$, and convert it to binary. We've included a decimalToBinary() function in decBin.py which translates decimal fractions to binary; it's used like so:

```
>>> import math
>>> from decBin import *
>>> decimalToBinary( math.log10(2), 50)
'.01001101000100000100110101000010011111011110011111'
```

This confirms (at least for the first 50 digits) that Listing 3.18 is producing the correct output.

Instead of 2 and 10 we could use other numbers as the inputs a and b for Listing 3.18, provided they generate a long fractional binary form (so 1 and 2 aren't suitable; try them and see).

genBin2.py is a variant of Listing 3.18. After every 1000 digits of output it prints the total number of "ones" and "zeros" produced so far. This is a simple way to check the quality of the randomness, since the two sums should be about equal.

With $a = 2$ and $b = 10$ and $n = 10000$, we get the following:

```
> python genBin2.py
a b n? 2 10 10000
1s: 504  0s: 496
1s: 997  0s: 1003
1s: 1498  0s: 1502
1s: 1996  0s: 2004
1s: 2517  0s: 2483
1s: 3016  0s: 2984
1s: 3537  0s: 3463
1s: 4036  0s: 3964
1s: 4554  0s: 4446
1s: 5057  0s: 4943
```

3.11.2 Generating Random Decimal Digits. In this subsection we'll produce random decimals by calculating the reciprocal of a prime (e.g. 1/17, 1/2063), and extract the repeated decimal sequence in the result. Naturally, not all prime reciprocals are ideal: for example, $1/3 = 0.\overline{3}$ since the repeated decimal is just a single number. $1/7 = 0.\overline{142857}$ isn't much better, only supplying six numbers.

We want to choose a prime p whose reciprocal has a decimal expansion with a *maximal period* $p - 1$, such as 2063, 10463, 20087, 100667, and 2040287. For example, 1/2063 produces a repeated decimal 2062 digits long.

findRepSeq() in Listing 3.19 (repFrac.py) finds the repeating decimal sequence in the rational supplied as a separate numerator and denominator.

```
def findRepSeq(numer, denom):
  divResult = ""
  remPosMap = {}
  rem = numer % denom
  while ((rem != 0) and \
         (rem not in remPosMap)):
    remPosMap[rem] = len(divResult)
    rem = rem * 10
    quotient = rem // denom
    divResult += str(quotient)
    rem = rem % denom
  if (rem == 0):
    return None
  else:
    return divResult[remPosMap[rem]:]

if __name__ == "__main__":
  numer, denom = map(int, input("numer
       denom=? ").split())
  decimal.getcontext().prec = 100
  print("Fraction:",D(numer)/D(denom))
  seq = findRepSeq(numer, denom)
  if (seq == None):
    print("No repeating seq")
  else:
    print("Repeating seq length:", len
        (seq))
    print("Repeating seq is", seq)
    c = collections.Counter(seq)
    cSort = dict(sorted(c.items()))
    print(cSort)
```

Listing 3.19. Finding a fraction's repeated decimal

For example:

```
> python repFrac.py
numer denom=? 2 3
Fraction: 0.666666666666
6666666666666666666666
6666666666666666666666
6666666666666666666666
6666666666666667
Repeating seq length: 1
Repeating seq is 6
{'6': 1}

> python repFrac.py
numer denom=? 1 7
Fraction: 0.142857142857
1428571428571428571428
1428571428571428571428
1428571428571428571428
1428571428571429
Repeating seq length: 6
Repeating seq is 142857
{'1':1, '2':1, '4':1, '5':1,
 '7':1, '8':1}
```

The while-loop generates a quotient and new remainder when the current remainder is divided by the denominator, and the quotient is added to the end of a division result string.

The unusual aspect of the code is the presence of a map which uses the remainders as keys, and the current length of the division result as each key's value. The map is employed to check if the current remainder has appeared previously, in which case the decimal sequence is about to repeat. The loop terminates and

the map value for the remainder is used to return the slice of the divResult string that contains the repeating decimal.

The top-level of the program prints the repeated decimal, and also its length and a tally of the digits used in the sequence.

It's known that for any prime p excepting 2 and 5, the decimal expansion of $1/p$ repeats with a period that divides $p-1$. In other words, the period could be as large as $p-1$, but no larger, and if it's less than $p-1$, then it's a divisor of $p-1$.

Listing 3.20 (primePeriods.py) checks this by generating all the primes below 3000 and uses findRepSeq() to calculate their repeated decimals. The output reports the length of each repeated sequence, and stars those whose lengths are maximal.

```
ps = primes(3000)
i = 0
for p in ps:
  seq = findRepSeq(1, p)  # find seq for 1/p
  if seq == None:
    print(f"{p:4d}:{0:<4d}  ", end = '')
  else:
    if len(seq) == p-1:
      print(f"{p:4d}:{len(seq):<4d}* ", end = '')
    else:
      print(f"{p:4d}:{len(seq):<4d}  ", end = '')
  i += 1
  if i%6 == 0:
    print()
print()
```

Listing 3.20. Prime reciprocals' repeated decimal lengths

Partial output:

```
> python primePeriods.py
   2:0        3:1        5:0        7:6   *   11:2       13:6
  17:16  *   19:18  *   23:22  *   29:28  *   31:15      37:3
  41:5       43:21      47:46  *   53:13      59:58  *   61:60  *
  67:33      71:35      73:8       79:13      83:41      89:44
  97:96  *  101:4      103:34     107:53     109:108 *  113:112 *
 127:42     131:130 *  137:8      139:46     149:148 *  151:75
    :  # many lines not shown
2803:1401  2819:2818* 2833:2832* 2837:709   2843:1421  2851:2850*
2857:408   2861:2860* 2879:1439  2887:2886* 2897:2896* 2903:2902*
2909:2908* 2917:1458  2927:2926* 2939:2938* 2953:984   2957:1478
2963:1481  2969:371   2971:2970* 2999:1499
```

It's possible to check the starred primes since they're sequence A073761 at OEIS (`https://oeis.org/A073761`).

As to why the decimal expansion of $1/p$ repeats with a period that divides $p-1$, consider

$$1/7 = 0.\overline{142857}$$

with a period of 6. Now multiply it by 10^6:

$$10^6 \times 1/7 = 142857.\overline{142857}$$

Subtract the original

$$(10^6 - 1) \times 1/7 = 142857$$

And so,

$$1/7 = 142857/(10^6 - 1)$$

The denominator (7) is one more than a power of 10 on the right-hand side, and that power is the period of the decimal expansion.

Rearrange the equation:

$$10^6 - 1 = 142857 \times 7$$

More generally,

$$10^n - 1 = a \times d$$

where a is some multiple, d the denominator of the original fraction, and n is the period of the decimal expansion.

Since $a \times d \equiv 0 \pmod{d}$ then

$$10^n - 1 \equiv 0 \pmod{d}$$

or

$$10^n \equiv 1 \pmod{d}$$

Excluding the primes 2 and 5, the decimal expansion of $1/p$ has a repeating part with period n such that

$$10^n \equiv 1 \pmod{p}$$

or

$$p|10^n - 1,$$

This gives us the smallest n such that $10^n - 1$ is divisible by p. How do we know there exists any such integer n? Fermat's little theorem tells us that

$$p|10^{p-1} - 1$$

and so we could set $n = p - 1$, though this may not be the smallest such n. But, not only does n exist, we know that it is at most $p - 1$. The repeating part of $1/p$ is no more than $p - 1$.

We can employ modular arithmetic to determine the length of the period for prime reciprocals. Since $10^n \equiv 1 \pmod{d}$ we can just look for the smallest $n > 0$ satisfying this.

For example, for 1/13, $d = 13$, so try $10^2 \equiv 9 \pmod{13}$, $10^3 \equiv -1 \pmod{13}$, ..., until $10^6 \equiv 1 \pmod{13}$. The repeating part of 1/13 is 6 digits long.

The output from Listing 3.20 listed 2939 as a prime with a maximal period. The details can be obtained by calling `repFrac.py`:

```
> python repFrac.py
numer denom=? 1 2939
Fraction: 0.000340251786321878189860496767608029942157196325280707
772371554950663490983327662470227968696835658387 21
Repeating seq length: 2938
Repeating seq is 00034025178632187818986049676760802 99
     : # many more lines
47056822048315753657706702960190541
{'0': 293, '1': 294, '2': 294, '3': 294, '4': 294,
 '5': 294, '6': 294, '7': 294, '8': 294, '9': 293}
```

The sequence is the correct length, and the distribution of the digits is fairly uniform, which might be useful.

Exercise

The output above suggests that the frequencies of the digits in a maximal period differ by at most 1. Prove that if division by p has period $p - 1$ then this will always be so. Similarly, the numbers of occurrences of pairs, triples etc. of digits differ by at most 1. Write programs for checking this for pairs and triples.

3.12 Shift Registers: More Coin Tosses

Figs. 3.32 to 3.34 show six shift registers, each filled by a vector of binary digits.



Figure 3.32. Shift registers (a) and (b)

The registers generate a stream of bits as follows: first the modulo 2 sum s of the tapped cells (indicated by arrows leading from these cells) is formed. Then the initial vector moves one place to the left and the resulting empty cell is filled by s.

Take for instance the shift register (c) in Fig. 3.33.

The register can assume 2^7 or 128 states, but 0000000 is a special state that cannot be reached from a non-zero vector. We have tapped the cells so that the vector will pass through all of these before the initial vector is regenerated.



Figure 3.33. Shift registers (c) and (d)



Figure 3.34. Shift registers (e) and (f)

These six registers can also be described by linear difference equations, and so are sometimes called *linear shift registers* (LSRs). The registers in Figs. 3.32 to 3.34 can be defined by

$$x_n = x_{n-1} + x_{n-4}, \quad x_n = x_{n-5} + x_{n-6}, \quad x_n = x_{n-6} + x_{n-7},$$
$$x_n = x_{n-5} + x_{n-9}, \quad x_n = x_{n-7} + x_{n-10}, \quad x_n = x_{n-9} + x_{n-11},$$

respectively. All of them employ addition mod 2.

Listing 3.21 (shift.py) encodes a shift register as a list.

```
def shifts(regs,fn,r1,r2,numShifts):
  initRegs = regs[:]
  print(*regs, sep='', end = ' ')
  for i in range(numShifts):
    if i%7 == 0: print()
    xn = fn(regs[r1], regs[r2])
    regs = regs[1:]+[xn]
    if regs == initRegs:
      print(*regs,sep='',end = '* ')
      print(i+1)
    else:
      print(*regs,sep='',end = ' ')
  print()

if __name__ == "__main__":
  shifts([0, 1, 1, 0, 1, 0, 0],
         xor, -6, -7, 134)
```

Listing 3.21. Implementing a shift register

```
> python shift.py
0110100 1101001 1010010 0100101
1001011 0010111 0101110 1011101
0111011 1110111 1101110 1011100
0111001 1110011 1100110
  : # more lines not shown
1100011 1000110 0001101 0011010
0110100* 127
1101001 1010010 0100101 1001011
0010111 0101110 1011101
```

The top-level call in Listing 3.21 evaluates shift register (c) in Fig. 3.33, and is equivalent to the LSR equation $x_n = x_{n-6} + x_{n-7}$. 134 shifts are printed.

The starred register 0110100 in the output indicates that the initial state has been re-entered, in this case after a period of 127.

The following execute the shift registers (a) and (b) in Fig. 3.32:

```
>>> from shift import *

>>> shifts([1,0,1,0], xor, -1, -4, 20)
1010 0101 1011 0110 1100 1001 0010 0100 1000 0001
0011 0111 1111 1110 1101 1010* 15
0101 1011 0110 1100 1001

>>> shifts([1,0,1,1,1,0], xor, -5, -6, 70)
101110 011101 111011 110110 101100 011001 110011
100110 001101 011010 110101 101010 010101 101011
010111 101111 011111 111111 111110 111100 111000
  : # more lines not shown
001110 011100 111001 110010 100100 001001 010010
100101 001011 010110 101101 011011 110111 101110* 63
011101 111011 110110 101100 011001 110011 100110
```

Exercises

(1) Show that all the shift registers in Figs. 3.32 to 3.34 generate sequences with maximal periods.

(2) Show that the binary LSR-sequences are *purely periodic* in the same sense as Fig. 2.3a.

(3) In the sequence 1983113835952... each digit after the fifth is the modulo 10 sum of the preceding four digits. Does the sequence contain the word

a) 1234; b) 3269; c) 5198; d) 1983 a second time; e) 1357?

Parts a) to d) can be answered without the aid of Python, but coding might be useful for (e). Experiment with different initial values. How does the period length depend on the initial values?

(4) The two LSRs in Fig. 3.35 generate streams of binary and base 5 digits with maximal periods. Check this with Python. Initially the cells are loaded with a non-zero vector.

We call the two streams x and y, and can use x to transform y into a sequence of decimal digits by means of the following algorithm: if $x_n = 0$, leave y_n unchanged. If $x_n = 1$, set $y_n = y_n + 5$. What's the period of the resulting sequence of decimal digits?

(5) The LSRs defined by the recurrences $x_n = x_{n-14} + x_{n-17} \pmod 2$ and $y_n = 4y_{n-1} + 3y_{n-7} \pmod 5$ generate sequences with periods $2^{17} - 1$ and $5^7 - 1$



(a) Binary LSR: $x_n = x_{n-3} + x_{n-5} \pmod 2$

(b) Base-5 LSR: $y_n = 4y_{n-4} + y_{n-5} \pmod 5$

Figure 3.35. Shift registers for Ex. 4

respectively. If they're combined in the same way as in Ex. 4, what is the period of the resulting sequence?

Additional Remarks about LSRs. The quality of the random digits generated by LSRs goes up (i.e. they satisfy more standard tests) as the order of the corresponding difference equations increases.

The recursion given for $n \geq 55$ by $x_n = (x_{n-24} + x_{n-55}) \mod m$, is recommended by Knuth as a very good generator of random digits in an even base (Vol. 2, Sec. 3.3.2, [**Knu97**]) when $x_0, ..., x_{54}$ are not all even initially. For $m = 2^e$ the length of the period is

$$2^f(2^{55} - 1), 0 \leq f < e.$$

Listing 3.22 (randig10.py) implements the $m = 10$ case. It generates n random decimal digits and counts the frequencies of the digits.

```
SIZE = 55
n = int(input("n=? "))
freqs = [0]*10
x = [random.randint(0,9)
       for i in range(SIZE)]
j = 23; k = SIZE-1
for i in range(n):
  d = (x[k]+x[j])%10
  freqs[d] += 1; x[k] = d
  j = (j-1)%SIZE
  k = (k-1)%SIZE
for i in range(10):
  print(i, freqs[i])
```

```
> python randig10.py
n=? 10000
0 978
1 982
2 991
3 1041
4 987
5 1000
6 959
7 998
8 1045
9 1019
```

Listing 3.22. A Knuth generator

We initially fill x[0],..., x[54] with (low quality) random digits, and set $j = 23$ and $k = 54$. The evaluation progresses counterclockwise around the circle

in Fig. 3.36, adding the pairs mod 10 and reducing j and k by 1. When j or k reaches 0, the index jumps to 54. The results of the additions are (high quality) decimal random digits.

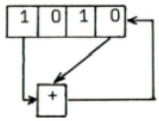


Figure 3.36. The Knuth LSR generator

The digit frequencies reported in the output are spread around 1000, but not too tightly, which is an indicator of the randomness of the generated numbers.

3.13 Random Sequence Generation by Cellular Automata

Since randomness is tantamount to total unpredictability, and it's well known that simple quadratic difference equations can lead to unpredictable, and even chaotic behavior, we now turn to nonlinear shift registers (NLSR) or "cellular automata". For more details on this approach, see S. Wolfram, 'Random Sequence Generation', *Advances in Appl. Math.* 7 (1986), pp.123-169.

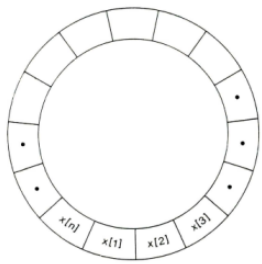


Figure 3.37. A CA buffer

A simple, well studied and very good generator of coin tosses is

$$y_n = x_{n-1} + x_n + x_{n+1} + x_n x_{n+1} \pmod 2$$

This can be written more simply and efficiently as

$$y_n = x_{n-1} \textbf{ xor } (x_n \textbf{ or } x_{n+1}).$$

It transforms an infinite sequence (x_n) into another sequence (y_n), which we'll implement with the help of a circular buffer (see Fig. 3.37).

For $i = 1, 3, ..., n-2$ we use the recurrence

$$y[i] = x[i-1] \textbf{ xor } (x[i] \textbf{ or } x[i+1]),$$

but

$$y[0] = x[n-1] \textbf{ xor } (x[0] \textbf{ or } x[1]),$$
$$y[n-1] = x[n-2] \textbf{ xor } (x[n-1] \textbf{ or } x[0]).$$

Listing 3.23 (`orxor1.py`) is based on these ideas.

```
n,max = map(int, input("n max=? ").
    split())
x = [random.choice([True, False]) for
    i in range(n)]
y = [False]*n
for _ in range(max):
  y[0] = x[n-1] ^ (x[0] or x[1])
  y[n-1] = x[n-2] ^ (x[n-1] or x[0])
  for i in range(1,n-2):
    y[i] = x[i-1] ^ (x[i] or x[i+1])
  for i in range(n):
    print(int(y[i]), end=' ')
  print()
  for i in range(n):
    x[i] = y[i]
```

Listing 3.23. A CA generator

Typical output:

```
> python orxor1.py
n max=? 15 10
0 0 0 1 1 1 0 1 1 0 0 0 1 0 0
0 0 1 1 0 0 0 1 0 1 0 1 1 0 0
0 1 1 0 1 0 1 1 0 1 0 1 0 0 0
1 1 0 0 1 0 1 0 0 1 0 1 1 0 0
1 0 1 1 1 0 1 1 1 1 0 1 0 0 1
0 0 1 0 0 0 1 0 0 0 0 1 1 0 1
1 1 1 1 0 1 1 1 0 0 1 1 0 0 1
0 0 0 0 0 1 0 0 1 1 1 0 1 0 1
1 0 0 0 1 1 1 1 1 0 0 0 1 0 1
0 1 0 1 1 0 0 0 0 1 0 1 1 0 1
```

The list x is filled by n random booleans. Then the sequence y is computed from x, and copied back into x.

Exercise

Explore the similar rule $y_i = x_{i-1}$ **xor** $(x_i$ **or** $(1 - x_{i+1}))$, or

$$y_i = (1 + x_{i-1} + x_{i+1} + x_i x_{i+1}) \pmod 2.$$

3.14 The Period Finding Problem

Given a function f which maps a finite domain D of integers into itself, and an arbitrary starting point $x \in D$, the sequence $x_0 = z$, $x_1 = f(x_0)$, $x_2 = f(x_1)$, $x_3 = f(x_2)$, ... is ultimately periodic. That is, for some t and c we have $t + c$ distinct values $x_0, x_1, ..., x_{t+c-1}$, but $x_{t+c} = x_t$. This implies that $x_{i+c} = x_i$ for all $i > t$. The integer c is the *cycle length* and t is the *tail length*.

This situation is illustrated by the graph in Fig. 3.38. The first t elements (6 in this case) are unique, and comprise the *tail*, while the next c elements (12) form the *cycle*. For example $x_{18} = x_6$, $x_{19} = x_7$, and so on.

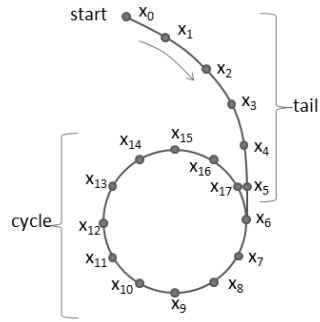


Figure 3.38. A tail and cycle

The problem of finding the unique pair (t, c) is called the *cycle problem*, and arises when analyzing random number generators that produce a new number by applying a function to the preceding value in the sequence, as is the case for the Lehmer PRNG in Sec. 3.1.

One simple solution for the cycle problem is due to R. W. Floyd (see `https://en.wikipedia.org/wiki/Cycle_detection`). We work with two variables, called `slow` and `fast`, which traverse the nodes of the graph at different speeds (see Listing 3.24 (fcycle.py)).

```
def findCycle(fn, start):
  slow = start; fast = start
  cycleLen = 0; tailLen = 0
  # let fast catch up to slow
  while True:
    slow = fn(slow)
    fast = fn(fn(fast))
    cycleLen += 1
    if slow == fast:
      break

  if slow == start:  # no tail
    print("pure cycle")
  else:  # slow goes round cycle
    cycleLen = 0
    while True:
      slow = fn(slow)
      cycleLen += 1
      if slow == fast:
        break
    # calculated cycle len

    # set fast back to start to
        calculate tail
    fast = start
    tailLen = 0
    while True:
      fast = fn(fast)
      slow = fn(slow)
      tailLen += 1
      if fast == slow:
        break
  return cycleLen, tailLen
```

Listing 3.24. Finding a cycle

fcycle() has three main parts. In part 1, `fast` proceeds twice as fast as `slow`, and when they catch up to each other it must be somewhere in the cycle.

In part 2, if they've met at the start node then this graph has no tail, and represents a *pure cycle*. Otherwise, `slow` is sent once more around the cycle in order to measure its length.

Part 3 has `fast` measure the length of the tail. It is sent back to the start node, and now both `fast` and `slow` move at the same unit speed. When they meet again it must be at the first cycle node after the end of the tail (e.g. at x_6 in Fig. 3.38). This is the tricky part of the algorithm, but can be proved by considering all the possible combinations of odd and even lengths for t and c.

All of the node moves are done by calling fn() which returns the next node in the graph. For example, the graph in Fig. 3.38 is defined using:

```
def dLoop(x):
  if x < 17:
    return x+1
  else:
    return (x%17)+6
```

Listing 3.25. The dLoop() function

The top-level call to findCycle() in this case is:

```
c, t = findCycle( dLoop, 0)
print("dLoop. Cycle len:", c, " tail len:", t)
```

The result is:

```
dLoop. Cycle len: 12  tail len: 6
```

3.15 Shrinking Squares

We label the four successive vertices of a square with four non-negative real numbers a, b, c, d (see the first square in Fig. 3.39).

Then each midpoint is labeled with the absolute value of the difference between the labels of its neighbors, and this process is repeated until we get to (0,0,0,0).

Ignoring the pretty nested squares, we are repeatedly applying the transformation

$$T : (a, b, c, d) \mapsto$$
$$(|a - b|, |b - c|, |e - d|, |d - a|).$$

This is implemented in Listing 3.26 (square.py), which prints the successive values of the tuple, and the number of steps until the process stops.

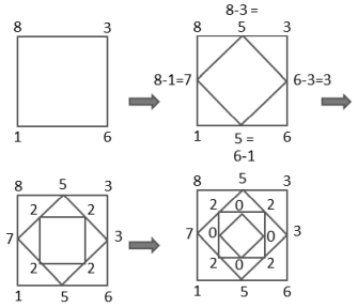


Figure 3.39. Labeling midpoint vertices

```
a,b,c,d = map(float, input("a b c
    d=? ").split())
count = 0
while (a+b+c+d) != 0:
  a1 = abs(a-b); b1 = abs(b-c)
  c1 = abs(c-d); d1 = abs(d-a)
  a = a1; b = b1; c = c1; d = d1
  count += 1
  print(f"{a:10.5f} {b:10.5f} {c
      :10.5f} {d:10.5f}")
print("count =", count)
```

Listing 3.26. Reducing a 4-tuple

```
> python square.py
a b c d=? 1 6 3 8
5.00000 3.00000 5.00000 7.00000
2.00000 2.00000 2.00000 2.00000
0.00000 0.00000 0.00000 0.00000
count = 3
```

After playing with Listing 3.26 for some time, the impression is that we always reach $(0, 0, 0, 0)$ after not too many steps. But there is an exception.

The differences in consecutive 4-tuples form a geometric sequence, which can be made explicit by labeling the values as $\{1, \quad t, \quad t^2, \quad t^3\}$ where t is some number > 1. Note that the first three differences, $t-1, t^2-t$ and t^3-t^2, advance by the same $t-1$ factor:

$$1(t-1), \quad t(t-1), \quad t^2(t-1)$$

To make the fourth difference advance in the same way, we require that

$$\begin{aligned} t^3 - 1 &= (t-1)t^3 \\ (t-1)(t^2+t-1) &= (t-1)t^3 \\ t^2+t-1 &= t^3 \\ t^3 - t^2 - t - 1 = 0 \end{aligned}$$

Now the absolute differences reproduces the original configuration, multiplied by $t-1$:

$$t-1, \quad (t-1)(t+1), \quad (t-1)t^2, \quad (t-1)t^3$$

which means that the algorithm will never reach $(0, 0, 0, 0)$ and never stop.

It turns out that the only initial values for which the algorithm does not stop are those obtained from the above by an affine mapping $y = ax + b$ applied to each component of the 4-tuple, or by reversing the order in which the numbers are written.

This problem can be solved using matrices and eigenvectors, as detailed in Problem 29 'Hustle Off to Buffalo' in M. Gardner, *Riddles of the Sphinx* [**Gar88**]. See also p.80 of R. Honsberger, *Ingenuity in Mathematics* [**Hon70**]. We'll instead find the roots for the equation $t^3 - t^2 - t - 1 = 0$ using our bisection code from Listing 1.25.

```
>>> from bis import *
>>> def fn(t):
  return t*t*t - t*t - t - 1

>>> fn(1)
-2
>>> fn(2)
1
>>> fn(3)
14
>>> bis(fn, 1, 2)
1.8392867553047836
```

For visual confirmation, we can plot the equation using plotEq.py:

```
> python plotEq.py
x equation? x*x*x - x*x - x -1
a b=? 1 2
```

The plot is shown in Fig. 3.40.

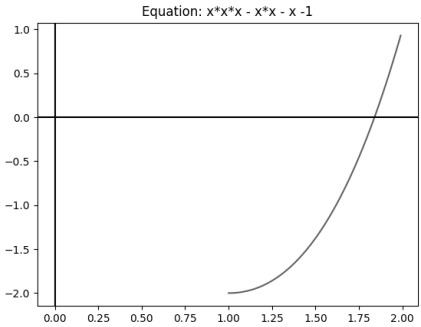


Figure 3.40. Plot of $x^3 - x^2 - x - 1$

3.15.1 Transforming n-tuples. Let's consider the more general mapping of an n-tuple containing rationals $x_0, x_1, ..., x_{n-1}$.

$$T : (x_0, x_1, \cdots, x_{n-1}) \rightarrowtail (|x_0 - x_1|, |x_1 - x_2|, \cdots, |x_{n-1} - x_0|).$$

After the first step the n-tuple will only contain positive numbers. Also for any factor $c > 0$, $(x_0, x_1, ..., x_{n-1})$ and $(cx_0, cx_1, ..., cx_{n-1})$ have the same life expectancy or cycling behavior. This allows us to replace the tuple's rationals by positive integers by multiplying them by a suitable c to create integers which are pairwise co-prime.

Listing 3.27 (polygon.py) starts with n random integers from $\{0, 1, ..., 99\}$ and repeatedly applies the T mapping until the n-tuple contains only zeros.

```
n = int(input("n=? "))
xs = [random.randint(1,99)
          for i in range(n)]
count = 0
while sum(xs) != 0:
  count += 1
  t = xs[0]
  for i in range(n-1):
    xs[i] = abs(xs[i] - xs[i+1])
  xs[n-1] = abs(xs[n-1] - t)
  for i in range(n):
    print(f"{xs[i]:4}", end ='')
  print();

print("count =" ,count)
```

Listing 3.27. Reducing polygon values to 0s

```
> python polygon.py
n=? 8
  52  66  21  36  62  51  33   1
  14  45  15  26  11  18  32  51
  31  30  11  15   7  14  19  37
   1  19   4   8   7   5  18   6
  18  15   4   1   2  13  12   5
   3  11   3   1  11   1   7  13
   8   8   2  10  10   6   6  10
   0   6   8   0   4   0   4   2
   6   2   8   4   4   4   2   2
   4   6   4   0   0   2   0   4
   2   2   4   0   2   2   4   0
   0   2   4   2   0   2   4   2
   2   2   2   2   2   2   2   2
   0   0   0   0   0   0   0   0
count = 14
```

By experimenting with this program we made the following discoveries:

a) The algorithm always stops if n is a power of 2, as in the example above.

b) For $n \neq 2^r$ (e.g. $n = 7$) the algorithm almost never stops as it enters a cycle.

c) In a cycle, eventually only two numbers survive: 0 and some positive integer.

d) A consequence of c) is that the initial n-tuple may as well just use some combination of 0s and 1s. Indeed, this would allow us to replace the absolute difference operation by modulo arithmetic since

$$|a - b| = (a + b) \bmod 2 = a \text{ xor } b.$$

The expression $(a + b) \bmod 2$ is useful in algebraic work, since addition mod 2 has the algebraic properties of ordinary addition, whereas expressions involving absolute values can seldom be simplified. For example, we can use the additivity of $T \bmod 2$, in the following

$$\begin{aligned} T(x_0 + y_0, x_1 + y_1, ..., x_{n-1} + y_{n-1}) = \\ T(x_0, x_1, ..., x_{n-1}) + T(y_0, y_1, ..., y_{n-1}) \end{aligned}$$

to state that if c is the cycle length of both n-tuples on the right, then it's also the cycle length, or a multiple of the cycle length, of the left-hand tuple.

e) If n is odd, then $(1, 1, 0, ..., 0)$ always appears in the cycle.

We can use d) and e) to efficiently determine the maximal cycle length $c(n)$ for odd n. For any n-tuple x, the elements of T_x have the sum to 0 mod 2. Hence the n-tuples of a cycle must all sum to 0 mod 2. Every such n-tuple is the element-wise sum modulo 2 of n-tuples with two neighboring 1's and 0's in all other places. This, and the remark we made after d), imply that the cycle length of $(1, 1, 0, ..., 0)$ is $c(n)$.

Listing 3.28 (cycle1.py) finds $c(n)$ for odd n by applying T until the n-tuple reappears.

```
n = int(input("n=? "))
xs = [0]*n; xs[0] = 1; xs[1] = 1
print(*xs); c = 0
while True:
  while True:
    c += 1; t = xs[0]
    for i in range(n-1):
      xs[i] = xs[i] ^ xs[i+1]
    xs[n-1] = xs[n-1] ^ t; print(*xs)
    if (xs[0] + xs[1]) == 2: break
  if sum(xs[2:n]) == 0: break
print("cycle length c =", c)
```

Example:

```
> python cycle1.py
n=? 7
1 1 0 0 0 0 0
0 1 0 0 0 0 1
1 1 0 0 0 1 1
0 1 0 0 1 0 0
1 1 0 1 1 0 0
0 1 1 0 1 0 1
1 0 1 1 1 1 1
1 1 0 0 0 0 0
cycle length c = 7
```

Listing 3.28. Finding cycles for odd n sized tuples

Various cycle lengths for different n are shown in Tables 3.1 and 3.2.

n	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
c(n)	3	15	7	63	341	819	15	255	9709	63	2047	25575	13797	475107	31

Table 3.1. Cycle lengths for n

n	33	35	37	39	41	43	45	47	49	51
c(n)	1023	4095	3233097	4095	41943	5461	4095	8388607	2097151	255

Table 3.2. Cycle lengths for n (continued).

f) Our tables suggests that $n|c(n)$.

cycle2.py finds cycles for even n by adapting the "hare-and-turtle" algorithm that we employed in Sec. 3.14.

Examples:

```
> python cycle2.py
n=? 80
Catchup: 240 ; Cycle len: 240 ; Tail len: 13

> python cycle2.py
n=? 160
Catchup: 480 ; Cycle len: 480 ; Tail len: 31

> python cycle2.py
n=? 31
pure cycle
Catchup: 31 ; Cycle len: 31 ; Tail len: 0
```

With cycle2.py, we make the following discoveries:

g) $c(2n) = 2c(n)$ (e.g. compare the first two examples from above),

h) The number of catch-up steps is always equal to the cycle length.

i) The tail length can vary from 0 (which defines a pure cycle) to 2^r, where r is the maximum exponent, such that $2^r|n$.

j) $c(n) = n$ for $n = 2^r - 1$ (e.g. see the last example above).

For odd n we have $c(2^r n) = 2^r \cdot c(n) = 2^r n \cdot q(n)$, but we need more information about $q(n)$. Very often we have

$$q(n) = 2^m - 1 \quad \text{and} \quad c(n) = n(2^m - 1). \tag{3.11}$$

The first two exceptions are $n = 23$ and $n = 35$:

```
> python cycle2.py
n=? 23
pure cycle
Catchup: 2047 ; Cycle len: 2047 ; Tail len: 0

> python cycle2.py
n=? 35
pure cycle
Catchup: 4095 ; Cycle len: 4095 ; Tail len: 0
```

$c(23) = 2^{11} - 1$ and $c(35) = 2^{12} - 1$. Thus $q(23)|(2^{11} - 1)$ and $q(35)|(2^{12} - 1)$.

What about m in Equ. (3.11)? Its value appears to be an order of 2 mod $c(n)$, i.e. the smallest m such that $2^m \equiv 1 \bmod c(n)$. That is

$$c(n)|(2^m - 1).$$

Finding proofs for most of our discoveries is quite difficult.

Exercises

(1) Show that we always arrive at an all-zero n-tuple for $n = 4$ and $n = 2^r$.

(2) Show that within a cycle just two numbers survive: 0 and some other number $a > 0$.

(3) Show that for odd n the n-tuple $(1, 1, 0, ..., 0)$ always appears in a cycle, and $(1, 0, ..., 0)$ never does.

(4) Show that $c(2n) = 2c(n)$.

(5) Show that $c(n) = n$ for $n = 2^r - 1$.

(6) Show that $n|c(n)$ for all n.

(7) Suppose $t = 1.8392867553047836$. How many steps do you need to reduce $(1, t, t^2, t^3)$ to $(0, ..., 0)$?

(8) *Bulgarian Solitaire.* a) Start with a triangular number $t = n(n+1)/2$ of cards distributed into stacks of any number k. A move consists in taking one card from each stack and forming a new stack with those k cards. One-card stacks will disappear during this move. You win the game as soon as the table has n stacks containing $1, 2, 3, ..., n$ cards respectively.

Write a program which plays the game. Count the number of moves, and try to find empirically the maximum number of moves until you win.

b) Investigate the case when t is not a triangular number, and a win is no longer possible. Develop some theorems about the cyclic structure of the game.

Additional Exercises for Chapters 1 to 3

(1) Write a function to represent the graph in Fig. 3.41.

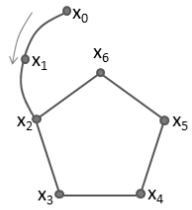


Figure 3.41. Another tail and cycle

Calculate its cycle and tail lengths using Listing 3.24 (fcycle.py).

(2) Incorporate the Lehmer PRNG from Listing 3.1 into Listing 3.24 to calculate its cycle length c and tail length t. Note that its default values for MOD and CONST will cause Listing 3.24 to take a long time to finish, so you might want to start with smaller numbers for those constants. What do the reported c and t values mean?

(3) We define an infinite binary sequence as follows: start with 0 and repeatedly replace each 0 by 001 and each 1 by 0.

a) Write a program that prints this sequence.

b) What is the 10000-th bit of the sequence?

c) What is the proportion of zeros among the first 10000 bits of the sequence?

d) Can you show that the sequence is not periodic?

e) Develop a formula for the place numbers of the 1's that is, for the sequence 3, 6, 10, 13, 17,.... Also find a formula for the place numbers of the 0's.

(4) Start with $a[0], a[1], ..., a[n]$ and extend it using $a[n+1] = \max_{0 \le i \le n}(a[i] + a[n-i])$. You will find that the differences ultimately become periodic. For instance, start with $a[i] =$ random.randint(10) for $i = 1, 2, ..., 9$.

(5) *An unsolved problem due to L.E. Dickson.* Given k integers $a[0] < a[1] < ... < a[k-1]$ define $a[n]$ for $n > k-1$ as the least integer greater than $a[n-1]$ which is not of the form $a[i] + a[j]$, $i, j < n$.

Is the sequence of differences $a[n] - a[n-1]$ eventually periodic? Take $k = 2, a[0] = 1, a[1] = 6$. Can you detect a periodicity in the sequence of differences? What about the set $\{1, 4, 9, 16, 25\}$? Is our cycle finding algorithm of any use here?

(6) Start with the digit 1 and repeatedly apply the replacement rule T:

$$0 \to 0000, \quad 1 \to 1321, \quad 2 \to 0021, \quad 3 \to 1300.$$

We get $T(1) = 1321$, $T^2(1) = T(1321) = T(1)T(3)T(2)T(1) = 1321130000211321$, ..., so that the sequence $T^{n+1}(1)$ has $T^n(1)$ as its initial segment. Thus, if we start with 1 and apply T repeatedly, we get longer and longer initial segments of an infinite sequence.

a) Show that this sequence does not change when T is applied.

b) Check if there is a period in the first 30,000 digits.

c) Show that this sequence is not periodic.

Remark. By placing a dot in the infinite sequence, e.g. $t = 1.321130000211321...$, we get the infinite expansion in some base ≥ 4. Then one can prove by advanced techniques that t is either rational (if periodic) or transcendental. We cannot get an algebraic irrational like $\sqrt{2}$ by fixed replacement rules, which means that t must be transcendental.

(7) *Numbers with the equisum property.* Write a program which generates random words with $2n$ decimal digits and checks if the sum of the first n digits is equal to the sum of the last n digits. Estimate the corresponding probability $p(n)$, and also $\sqrt{n}p(n)$. Guess a good approximation for $p(n)$.

(8) For the last time, we return to the sequence defined by

$$a_0 = 3, a_1 = 0, a_2 = 2,$$
$$a_n = a_{n-2} + a_{n-3}, n > 2.$$

One can show that n prime $\to n | a_n$ is always true. Show that $n | a_n$ for $n = 271441 = 521^2$. This may be the smallest counterexample to the conjecture $n | a_n \to n$ prime. There is no counterexample up to $n = 140000$. With matrices we can speed up the program from $O(n^2)$ to $O(n \log n)$, resulting in a huge saving in computation time.

(9) Write a program to find the smallest n with the following property: in the binary representation of $1/n$ the bit patterns of the binary representations of all the numbers from 1 to 1990 occur somewhere. n can also be found by Olympiad-level hard thinking plus some number theory.

(10) A die is rolled repeatedly until the sum $X_1+X_2+...+X_N$ of the points surpasses 100. Find by simulation

a) the most probable value of N

b) the most probable sum.

(11) *The Newton–Pepys problem.* In 1693 Samuel Pepys and Isaac Newton corresponded over the following problem: which of the following three propositions has the greatest chance of success?

(a) Six fair dice are tossed independently and at least one 6 appears.
(b) Twelve fair dice are tossed independently and at least two 6s appear.
(c) Eighteen fair dice are tossed independently and at least three 6s appear.

Pepys initially thought that outcome (c) had the highest probability of occurring, but Newton correctly concluded that outcome (a) is actually more likely. Write a program to confirm Newton's answer.

(12) *Sicherman dice.* Suppose that you have two six-sided dice, one with faces labeled 1, 3, 4, 5, 6, and 8 and the other with faces labeled 1, 2, 2, 3, 3, and 4. Compare the probabilities of occurrence of each of the values of the sum of the dice with those for a standard pair of dice.

Sicherman dice produce sums with the same frequency as regular dice (2 with probability 1/36, 3 with probability 2/36, and so on).

(13) *Three-sum analysis.* Calculate the probability that no triple among n random 32-bit integers sums to 0, and give an approximate estimate for n equal to 1000, 2000, and 4000. Hint: we utilized random.getrandbits(32) in our solution.

Give an approximate formula for the expected number of such triples (as a function of n), and run experiments to validate your estimate.

More information can be found at `https://en.wikipedia.org/wiki/3SUM`.

(14) Estimate the probability that a random triangle is obtuse. Approach this as a geometric problem, where the probability is approximated by testing many randomly generated triangles.

For a triangle to be obtuse, you'll recall from high school geometry that it must have an interior angle greater than 90°.

The crucial formula is the cosine law. If we denote the three interior angles of the triangle by A, B, and C and the lengths of the sides opposite those angles by a, b, and c, respectively, then we have

$$a^2 = b^2 + c^2 - 2bc\cos(A)$$

In other words:

$$\cos(A) = \frac{b^2 + c^2 - a^2}{2bc}$$

The final part of the puzzle is to recall that the cosine of an acute angle, i.e. an angle in the interval $(0, 90°)$, is positive, while the cosine of an angle in the interval $(90°, 180°)$ is negative.

This problem comes from *Digital Dice* by Paul Nahin [**Nah13**].

4

Statistics

4.1 The Subset Sum Problem (Part 1)

The subset sum problem (SSP) involves deciding whether any subset of the integers in a set S sum precisely to T. For example, if $S = \{5, 6, 2, 3\}$ and $T = 5$, then the answer is 'yes' since one possible subset is $\{2, 3\}$.

We'll consider a variation of this problem, where we want all the subsets with a sum $\leq T$. For instance, using the previous S and T, the subsets are $\{\{\}, \{2\}, \{3\}, \{5\}, \{2, 3\}\}$. We're focusing on this version since it'll be useful for several examples in this chapter.

Listing 4.1 (sumSubsets.py) is a recursive solution, which is both relatively easy to understand and also suffers from exponential running time as the set size increases.

```
def sumSubsetsR(nums, tot):
  combs = sumSubsets(nums, tot, 0, [])
  return sorted(combs, key=sum)

def sumSubsets(nums, tot, idx, comb):
  sumC = sum(comb)
  if sumC == tot:
    return [comb]
  elif sumC > tot:
    return []
  else:
    combs = [comb]
    for i in range(idx, len(nums)):
      combs.extend(sumSubsets(nums,tot,i+1,comb+[nums[i]]))
    return combs
```

Listing 4.1. A recursive subset sum

A typical call would be:

```
>>> from sumSubsets import *
>>> sumSubsetsR([5, 6, 2, 3], 5)
[[], [2], [3], [5], [2, 3]]
```

Note that although this is a set problem, we've decided to code things using lists. This allows us to include multiple copies of a value in S if necessary.

The crucial lines in sumSubsets() are its else-branch which builds a list of 'subsets' containing both the current list (comb) and every version where one or more numbers have been added by recursively calling sumSubsets().

As we've seen before, the standard way to speed up recursive code is to utilize memoization data structures (typically lists) to record subcalculations so they don't need to be repeated. sumSubsetsD() in sumSubsets.py maintains a Python dictionary whose key/value pairs are a sum and a list of lists that add up to that sum. This allows the code to be rewritten to be iterative, with a running time of $O(n^2)$, but with a potentially large memory overhead for the dictionaries. Such functions are usually classified as having *pseudo-polynomial* running time since the big-Oh calculation doesn't capture all the costs related to the problem size.

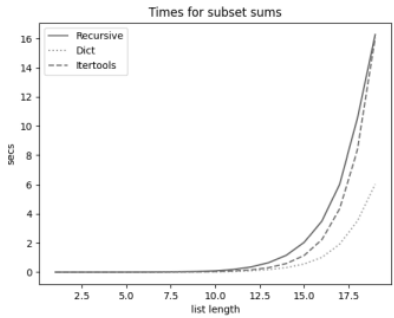


Figure 4.1. Subsets sum running times

Our last subset-sum solution utilizes Python's `itertools.combinations(seq, t)` which generates t-length subsequences of elements from seq. This is almost what we want, but we also have to generate all the smaller subsequences for $0 \leq x < t$, as shown in Listing 4.2 (sumSubsets.py).

```
def sumSubsetsC(nums, tot):
  combs = [list(comb) for r in range(len(nums)+1)
                for comb in itertools.combinations(nums, r)
                   if sum(comb) <= tot]
  return sorted(combs, key=sum)
```

Listing 4.2. An itertools subset sum

subsTimer.py uses the timeit module to time these three functions when given a list of 1's (e.g. [1], [1,1], [1, 1, 1], up to 20 1's) and tasked with generating

all the sublists that sum to 8 or less. The plot of their running times is shown in Fig. 4.1.

The most surprising thing about Fig. 4.1 is that Python's itertools.combinations() has a similar exponential running time as sumSubsetsR(), but this may be due to our use-case, which calls it $t + 1$ times for each list.

4.1.1 The Number of Subset Sums. One way to speed up our code is to simplify the coding problem! Often the actual sublists aren't required, just the total number. For instance:

```
>>> from countSums import *
>>> countSumsR([5, 6, 2, 3], 5)
5
```

Listing 4.3 (countSums.py) is recursive, with three boundary cases and an else-branch expressing the idea that the number of subsets involves adding together two subtotals – one where the current number in the list is not used, and the other where it is (which means deducting it from the required total).

```
def countSumsR(nums, tot):
  return countSums(nums, tot, len(nums))

def countSums(nums, tot, n):
  if tot < 0:
    return 0
  elif (tot == 0) or (n == 0):
    return 1
  else:
    return countSums(nums, tot, n-1) + \
           countSums(nums, tot-nums[n-1], n-1)
```

Listing 4.3. A recursive subset sum counter

In the same way as before, we can augment recursion with memoization, b this time we can utilize a 2D list which has a fixed size (unlike the dictionary in sumSubsetsD()).

Listing 4.4 (countSums.py) again illustrates that memoization usually permits recursion to be replaced by iteration, which in this case fills the table row-by-row.

```
def countSums2D(nums, tot):
  n = len(nums)
  tots = [[0 for j in range(tot+1)] for i in range(n+1)]
  for j in range(tot+1):     # set first row
    tots[0][j] = 1
  for i in range(1, n+1):    # set d's for other rows
    d = nums[i-1]
    for j in range(d):
      tots[i][j] = tots[i-1][j]
```

```
    for j in range(d,tot+1):
      tots[i][j] = tots[i-1][j] + tots[i-1][j-d]
  return tots[n][tot]
```

Listing 4.4. An iterative 2D subset sum counter

A close look at how the table is filled shows that the code only utilizes the current and previous rows, so the table could easily be made smaller. A further trick is to notice that the current row is filled left-to-right, so we only actually need to store one row. These observations result in Listing 4.5 (countSums.py).

```
def countSums1D(nums, tot):
  n = len(nums)
  tots = [1]*(tot+1)
  for i in range(1,n+1):
    d = nums[i-1]
    for j in range(tot, d-1, -1):
      tots[j] += tots[j-d]
  return tots[tot]
```

Listing 4.5. An iterative 1D subset sum counter

And so we conclude with a function that has linear running time, although we haven't actually quite finished yet – we'll return to this topic one more time in Sec. 5.9.

4.2 Matched Pairs

We'll study the simple but important problem of matched pairs by considering three examples.

4.2.1 Marijuana vs. Tobacco. The data in Table 4.1 comes from a study where nine individuals had their mental performance tested 15 minutes after smoking an ordinary cigarette (row X) and 15 minutes after consuming a marijuana cigarette (row Y), with positive numbers denoting improvements. To deal with the possible effect of smoking different substances in a certain order, a flip of a coin for each subject decided which type of cigarette they smoked first. The D row measures how much more positive the effects of marijuana were compared to tobacco.

The table shows that different people experienced different effects, but can we nevertheless infer something about the effects on the group, or are the fluctuations in mental performance just random? To deal with such a possibility, we formulate two hypotheses. The first is:

> **The null hypothesis** H: The fluctuations shown in Table 4.1 *are* just random. There's no difference in the effects produced by the two types of cigarette.

X	-1	-1	-3	3	-3	-3	2	4	10	$\sum X = 8$
Y	1	-3	-7	-3	-9	5	-6	-7	-17	$\sum Y = -46$
D = Y-X	2	-2	-4	-6	-6	8	-8	-11	-27	$\sum D = -54$
$\lvert D\rvert = d_k$	2	2	4	6	6	8	8	11	27	$sum\lvert D\rvert = 74$

Table 4.1. Marijuana study. X = tobacco; Y = marijuana. Source: *Science* 162, 1234-1242.

However, the experimenters, based on anecdotal evidence and the many negative differences in the D row, suspected that marijuana had a more detrimental effect than ordinary cigarettes. So they decided to test:

> **The alternative** A: Mental performance is *lower* after smoking marijuana than after smoking tobacco.

Hypothesis H can be given a precise meaning by making it equivalent to assigning $\pm$ sign differences by tossing a fair coin. Thus, with n people we would generate 2^n equiprobable cases.

Next, we choose a *test statistic* T which expresses quantitatively the amount of evidence for A contained in the differences D. We always define T so that a smaller value means that A is more likely to be true. Suppose the observed value of T is t. We must compute

$$P = P(T \le t|H) = \text{probability that } T \le t \text{ if } H \text{ is true.}$$

P is the *observed significance level*, or *P-value*, of the test, and the smaller the P-value the stronger the evidence for A.

We set $T = \#$ of positive signs in the D row, since as the number of positives decreases so A becomes more likely to be true (i.e. that marijuana is detrimental).

The observed value of T is $t = 2$ for a sample size of 9. There are $\binom{9}{0}+\binom{9}{1}+\binom{9}{2}$ $= 1 + 9 + 36 = 46$ cases which are favorable to event T ≤ 2. Thus,

$$P = P(T \le 2|H) = 46/2^9 = 9\%$$

Most statisticians only consider a P-value to be significant if it's 5% or less (and often less than 1% for small sample sizes, as here). The other, more fundamental problem is that this sign test doesn't make good use of the data. We should try to capture the fact that the few positive differences in the D row are much smaller on average than the negative differences.

A better statistic would be to measure the *size* of the positive differences vs. random differences for the nine subjects. This means that the null hypothesis would now be that all 2^9 differences are equally possible.

A quick look in row D identifies $T \le 10$, the sum of the positive differences. This is converted into the number of cases that give a total score of ≤ 10 based

on the sample data. For this we'll use the $|D|$ row since we're ignoring sign, and can employ subsetSumD() to find all the subsets which sum to 10 or less.

```
>>> sumSubsetsD([2,2,4,6,6,8,8,11,27], 10)
[[], [2], [2], [2, 2], [4], [2, 4], [2, 4], [6], [6], [2, 2, 4], [2, 6],
[2, 6], [2,6], [2, 6], [8], [8], [2, 2, 6], [4, 6], [2, 2, 6], [4, 6],
[2, 8], [2, 8], [2, 8], [2, 8]]
>>> len( sumSubsetsD([2,2,4,6,6,8,8,11,27], 10))
24
```

There are 24 ways to get positive scores of 10 or less. Thus

$$P(T \leq 10|H) = \frac{24}{512} = \frac{3}{64} = 4.6875\%.$$

This P-value is technically significant, but should really be taken as a sign to redo the experiment with a larger sample size.

What we have just done manually, and via the Python prompt, can of course be more fully automated. `mpairs.py` loads a text file containing matching pair data (e.g. see `marijuana.txt`), and analyzes the table to create a call to sumSubsetsD() like the one above. This program can also be used to calculate the results in the next example.

4.2.2 IQ vs. Weight. We would like to investigate the mental development of identical twins in relation to their weights at birth. The IQs of 12 pairs were measured years later and compared with their weights at birth. The X row in Table 4.2 holds the IQs of the heavier twin while the Y row are for the IQs of the lighter twin.

X	100	124	108	91	100	91	79	80	95	104	100	119
Y	101	123	106	97	106	84	70	70	84	92	85	104
D = Y - X	1	-1	-2	6	6	-7	-9	-10	-11	-12	-15	-15
$\|D\| = d_k$	1	1	2	6	6	7	9	10	11	12	15	15

Table 4.2. IQs of twins. X = heavy; Y = light. Source: *Child Development*, vol. 38, 623-629.

Our hypotheses:

- H: The heavy and the light twins develop the same IQ.
- A: The heavy twin usually develops a higher IQ.

Recall that we must define T so that a smaller value means that A is more likely to be true. We do that by summing the positive values in the D row of Table 4.2 which occur when the lighter twin has a higher IQ. The sum is $T = 1+6+6 = 13$.

Under H all 2^{12} (4096) subsets of the differences have the same probability, and the favorable cases for A are those subsets of d_k with sum to $T \leq 13$. We again find these by calling sumSubsetsD():

```
>>> sumSubsetsD([1,1,2,6,6,7,9,10,11,12,15,15], 13)
[[], [1], [1], [1, 1], [2], [1, 2], [1, 2], [1, 1, 2], [6], [6],
[1, 6], [1, 6], [1,6], [1, 6], [7], [1, 1, 6], [2, 6], [1, 1, 6],
  : # more lines, not shown
[1, 11], [12], [1, 6, 6], [1, 6, 6], [6, 7], [6, 7], [1,1, 2, 9],
[1, 2, 10], [1, 2, 10], [1, 1,11], [2, 11], [1, 12], [1, 12]]
>>> len(sumSubsetsD([1,1,2,6,6,7,9,10,11,12,15,15], 13))
60
```

There are 4096 possible cases, and 60 cases favorable for T $\leq$ 13. Thus

$$P = P(T \leq 13|H) = \frac{60}{4096} = \frac{15}{1024} = 1.465\%.$$

This P-value is strong evidence for A – a better nourished twin develops a higher IQ.

4.2.3 Cross vs. Self Fertilization. Finally, we consider a classic experiment by Charles Darwin which appears in his *The Effects of Cross and Self Fertilisation in the Vegetable Kingdom* (`https://www.gutenberg.org/ebooks/4346`).

He took 15 pairs of seeds of the same plant and planted them in 15 pots. One seed of each pair was produced by cross fertilization, the other by self fertilization. For pot #i he measured the height x_i of the cross fertilized plant and the height y_i of the self fertilized plant. For the difference $z_i = x_i - y_i$ he got (in 1/8-ths of an inch):

$$6, 8, 14, 16, 23, 24, 28, 29, 41, -48, 49, 56, 60, -67, 75.$$

Our hypotheses:

- H: There is no difference between the two kinds of seed.
- A: Cross fertilized seeds develop taller plants.

If H is true then we have 2^{15} possible and equiprobable cases.

Once again we define T so that a smaller value means that A is more likely to be true. We sum the negative values in the data row, which indicate when the self fertilized plants are taller than the cross fertilized ones. The sum of the negative differences is $t = 115$ (|-48 + -67|).

In the two previous examples, we used sumSubsetsD() at this stage, but these types of problem don't need the subsets only their total number. This allows us to switch to countSum1D(), which would actually be essential if the problem's sample size was bigger.

```
>>> from countSums import *
>>> countSums1D([6,8,14,16,23,24,28,29,41,48,49,56,60,67,75], 115)
863
```

We can now calculate P:

$$P = \frac{863}{2^{15}} = \frac{863}{32768} = 2.63\%.$$

This suggests that there is significant evidence for A: cross fertilized seeds do indeed develop into taller plants.

4.3 P-value by Simulation

Let's return to the marijuana example. If we plot Y versus X (Fig. 4.2) with `plotPairs.py` we see that there seems to be little relationship between the performances after smoking the two types of cigarette. This suggests that's there not much to be gained by considering the data in pairs.

Another approach is to treat the data (-3, 5, 10, -17, -3, -7, 3, -3, 4, -7, -3, -9, 2, -6, -1, 1, -1, -3) as random selections from an 'Effects of smoking on performance' probability spinner. The observed subset X due to tobacco smoking has sum $S = \sum X = 8$, while the subset Y due to marijuana smoking has sum $T = \sum Y = -46$.

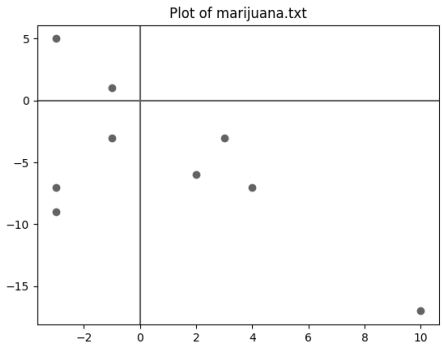


Figure 4.2. Performance affected by smoking (X=tobacco; Y=marijuana)

Our hypotheses:

- H: The large difference in the two sums is due to chance.
- A: The marijuana subset has lower performance than the tobacco subset.

If H is true, then X (the tobacco sample) is just a random 9-subset of the 18 data samples. So the question arises: how likely is it that a randomly chosen 9-subset of the data has a sum $S \geq 8$?

We can answer that through simulation – repeatedly draw a random 9-sample from the data and count how many times it sums to ≥ 8.

`permSim.py` utilizes code from Listing 3.10, packaged up as a sample() function.

To save some typing the tobacco and marijuana data is read from `marijuana.txt`:

```
2 9
-1 -1 -3  3 -3 -3  2  4  10
 1 -3 -7 -3 -9  5 -6 -7 -17
```

The simulation runs for NUM_TRIALS (32000) iterations, and reports the following:

```
> python permSim.py
fnm? marijuana.txt
x, y sums: 8 -46
count == 538; P == 0.016813
```

Over multiple runs, $P = P(S \geq 8|H)$ seems to vary between 1.5% and 1.7%.

4.3.0.1 Median and Confidence Interval. It's possible to extract more statistics from the simulation by analyzing the spread of P-values returned by multiple calls to permSim.py.

permSimStats.py generates 24 P-values, which it summarizes in a a stem-and-leaf plot. From these numbers we can extract an error estimate for the median P-value.

```
> python permSimStats.py
fnm? marijuana.txt
x, y sums: 8 -46
[1.69375, 1.74375, 1.50625, 1.740625, 1.6, 1.615625, 1.49375,
1.653125, 1.60625, 1.5875, 1.496875, 1.65625, 1.684375, 1.60625,
1.653125, 1.465625, 1.490625, 1.590625, 1.659375, 1.50625,
1.659375, 1.575, 1.484375, 1.6]
stem   leaf
   0 |
   1 | 444445555566666666666677
   2 |
```

We throw away the lower and upper quarters of the stem-and-leaf data. What remains is the *interquartile range* IQR $= 1.6 - 1.5 = 0.1$, and the median $C = (1.6 + 1.6)/2 = 1.6$.

McGill, Tukey and Larsen, in their 'Variations of Box Plots', *The American Statistician*, Vol. 32, No. 1 (Feb., 1978), pp. 12-16, developed a rule of thumb for estimating the true median with a 95% confidence interval:

$$C \pm \frac{1.57\ \text{IQR}}{\sqrt{N}} = 16 \pm (1.57 * 0.1)/\sqrt{24} \approx 1.6 \pm 0.03.$$

4.4 P-value by Computation

Let's consider the marijuana example one last time. The Null hypothesis is that it makes no difference what kind of cigarette a person smokes. If this were so then all 18 data samples could be regarded as values of the same random variable. In that case, the subset of 9 values for tobacco (the X data) would be just a randomly picked subset of 9 out of the 18. However, the sum of these 9 values is 8, which

seems rather large. To decide how likely it is that this happens just by chance, let's check all $\binom{18}{9}$ or 48620 9-element subsets of the smoking data set:

$$(-3, 5, 10, -17, -3, -7, 3, -3, 4, -7, -3, -9, 2, -6, -1, 1, -1, -3).$$

permCalc.py traverses the 9-subsets in lexicographic order and counts those which sum to 8 or more.

How do we generate all k-subsets of an n-set {1, 2, ..., n}? We start with {1, 2, ..., k}, and the next subset is found by scanning the current one from right to left to locate the rightmost index that has not yet reached its maximum. This element is incremented by one, and all positions to its right are reset to their lowest values.

In the following call, k is the size of the subsets, while tot is the sum that must be equaled or exceeded.

```
> python permCalc.py
k tot=? 9 8
count == 766
```

Exactly 766 9-subsets have a sum of 8 or more, so we have a P-value = 766/48620 = 1.5755%. This is the "exact" significance level for this data, and will be the value that permSim.py converges upon as the number of repetitions are increased towards ∞.

permCalc.py's traversals of the subsets has exponential running time, so is really only useful for small search spaces. For example, the number of 18-subsets of a 36-set is 9075135300. However, large sizes can be estimated using the approximation

$$\binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi n}}$$

which determines $\binom{36}{18}$ to be around 9138368253.

In Sec. 4.6 we'll deal with this subset counting task more efficiently. Alternatively, for very large problems, we could use permSim.py from the preceding section. It gives practically the same significance level. For instance, permCalc.py reports the P-value to be 1.5755% while permSim.py estimates it to be around 1.6%.

Exercises for Sections 4.1 to 4.4

(1) Should hospitals allow parents to stay with their sick children (the rooming-in model) to shorten their stay? 25 pairs of children were considered, with relevant variables nearly the same in all cases, except that the parents of one child in each pair were allowed to stay at the hospital. Table 4.3 shows the number of days each child stayed in the hospital; X was the no-parents group, Y the rooming-in group.

Find D = X - Y and compute the P-value with `countSums.py`. The data in Table 4.3 can be found in sickDays.txt.

X	29	44	18	15	15	7	15	11	12	12	7	11	7	22	7	14	28	31	18	16	10	11	29	8	23
Y	29	32	16	8	8	7	14	8	7	12	7	8	7	17	6	19	12	8	8	6	10	8	21	17	15

Table 4.3. Days in hospital. Source: Fernandez-Jung, F., Dissertation FU Berlin, 1983.

(2) Use `permSim.py` to analyze the data in Table 4.3 and find a 95% confidence interval for its P-value.

(3) Table 4.4 shows the frequency of one letter words per 1000 words in eight essays by A. Hamilton and seven by J. Madison.

Hamilton	24	21	23	24	33	28	28	37
Madison	20	27	19	30	11	17	27	

Table 4.4. Word frequencies: F. Mosteller and D. L. Wallace, *Inference and Disputed Authorship: The Federalist*. 1964, p. 248.

Our hypotheses:

- H: These are samples from the same distribution.
- A: These are samples from different distributions.

Find the P-value with `permSim.py`. The data is stored in `essays.txt`.

(4) Apply `permCalc.py` to Table 4.4. Find the P-value.

(5) Write a program that selects a random subset from the fourth row of the smoking data in Table 4.1 with sum T. It should increment a counter whenever $T \leq 10$ or $T \geq 64$, and repeat the selection task NUM_TRIALS times. At the end, it should estimate the P-value by calculating count/NUM_TRIALS/2 and compare the result with the exact value.

4.5 The Two Sample Problem of Wilcoxon

The Wilcoxon rank-sum test (also known as the Wilcoxon-Mann-Whitney test) determines whether two samples are drawn from the same distribution. It's best employed when we can determine the size of any difference between pairs of observations, as well as the direction of the difference.

For those with some statistics background, this may sound like Student's T-test, but that's used when both groups are normally distributed; Wilcoxon makes no assumptions about the underlying distribution.

The two names for the test are historical. Frank Wilcoxon proposed it in 1945, but a thorough analysis, including the recurrence relation we use below,

appeared in a 1947 article by Henry Mann and Donald Ransom Whitney: 'On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other', *The Annals of Mathematical Statistics*, Vol. 18, No.1 (Mar., 1947), pp. 50-60.

4.5.1 Data Analysis. We're going to examine the relationship between the heights and ages of death of the 35 US presidents who died of natural causes, based on data obtained from `https://potus.com/presidential-facts/`. The details are stored in CSV form in `presidents.csv`:

```
No,Name,Age,Height
1,George Washington,67,188
2,John Adams,90,170
3,Thomas Jefferson,83,189
4,James Madison,85,163
  : # more lines, not shown
38,Gerald R. Ford,93,183
40,Ronald Reagan,93,185
41,George H.W. Bush,94,188
```

The data is loaded and analyzed by `readPresidents.py` which generates a stem-leaf plot of the ages, a division of the ages into 'short' and 'tall' presidents, where 'short' is deemed to be < 173cm, a binary rank string, and two box-plots shown in Fig. 4.3.

The printed output:

```
> python readPresidents.py
Loading: presidents.csv
Keys: ['No', 'Name', 'Age', 'Height']
No. of records: 35

Presidential Ages
stem   leaf
   5 | 377
   6 | 003344567778
   7 | 0112347889
   8 | 01358
   9 | 00334

Short Ages: [67, 79, 80, 85, 90]
Tall Ages: [53, 57, 57, 60, 60, 63, 63, 64, 64, 65,
66, 67, 67, 68, 70, 71, 71, 72, 73, 74, 77, 78, 78,
81, 83, 88, 90, 93, 93, 94]

Ages as binary:
35 : 00000000000010000000000011001010000
Short Positions (from 1):
```

```
13 25 26 29 31
Rank sum: 124
```

The box-plots suggest that the 'short' and 'tall' groups might be two different populations, which we're going to investigate.

The binary rank string:

```
00000000000010000000000011001010000
```

is obtained by sorting the ages into ascending order, and representing a 'tall' president by 0 and a 'short' one by 1. It also indicates quite strongly that short presidents live longer. So we'll consider two hypotheses:

- H : Height has nothing to do with the age of death. The pattern is just a random fluctuation.
- A : Short presidents on average do live longer.

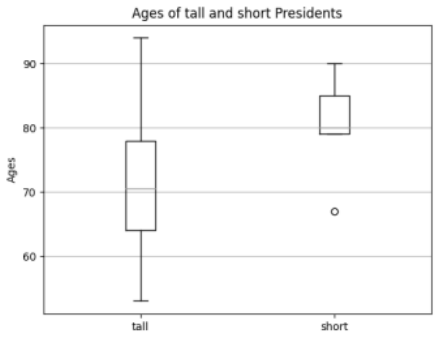


Figure 4.3. Box plot of presidential ages

readPresidents.py finishes by printing the positions of the 'short' presidents in the binary string (counting from the left, and starting at 1), and also the *rank sum* RS, the sum of those positions (124).

In the next two subsections we'll determine the P-value $= P(RS \geq 124|H)$. Our first approach approximates an answer through simulation, then we'll obtain an exact value using the Mann-Whitney recurrence relation.

4.5.2 An Approximate P-value. simupres.py chooses m of the ranks 1 to n at random and sums them. This is repeated NUM_TRIALS times, and a counter is incremented whenever the event RS $\geq$ max occurs.

When supplied with $n = 35$, $m = 5$, max $= 124$:

```
> python simupres.py
n m max=? 35 5 124
count == 1804; P == 0.05637500
```

We could also find a confidence interval for the P-value as we did in Sec. 4.3.

4.5.3 Analyzing the Binary String. Before we tackle the large binary string of presidential ages, let's look at the analysis steps using something smaller. 01001 is a 5-bit string with 1's in the second and fifth positions, so its rank sum RS $= 2 + 5 = 7$. We'll determine the exact P-value by finding the number of binary strings with 3 zeros and 2 ones with an RS ≥ 7.

1	2	3	4	5	RS
0	1	0	0	1	7
0	0	1	0	1	8
0	0	0	1	1	9
0	0	1	1	0	7

Table 4.5. All the 5-bit strings with RS ≥ 7

As shown in Table 4.5; there are 4 strings that qualify.

The total number of 5-bit strings with 3 zeros and 2 ones is $\binom{5}{2} = 10$, so the exact P-value $= 4/10 = 0.4$.

This manual approach isn't feasible for larger binary strings, such as the presidential ages, which is why Mann-Whitney developed the recurrence relation $w(u, m, n)$ to count the number of strings with a RS $\geq$ some value.

Before we can use $w(u, m, n)$, we need to determine initial values for its variables. Two of them are simple: m is the number of 1's in the string and n is the number of 0's. u is more complex, based on the *z sequence* for the string.

01001 can be encoded by a sequence $\{z_1, z_2\}$ of integers which count the number of zeros to the right of the first and second 1's in the string, counting to its end. So for 01001, $z_1 = 2$ and $z_2 = 0$. As another example, 11000 would be represented by $z_1 = 3$ and $z_2 = 3$. Note, that the length of the sequence equals the number of 1's in the string.

z_1 and z_2 can be converted into string positions for the 1's with the formula:

$$p_i = i + (n - z_i)$$

n is the number of 0's in the string ($n = 3$ here). So the position of the first 1 in 01001 is at $p_1 = 1 + (3 - z_i) = 1 + (3 - 2) = 2$ and the second 1 is at $p_2 = 2 + (3 - z_2) = 2 + (3 - 0) = 5$. Calculating the string's rank sum is now easy:

$$RS = \sum_{i=1}^{m} p_i = \sum_{i=1}^{m} (i - (n - z_i)) = \frac{m(m+1)}{2} + mn - \sum_{i=1}^{n} z_i$$

m is the number of 1's in the binary string ($m = 2$).

Recall that for 01001 we want to find the number of binary words with 3 zeros and 2 ones with rank sum RS ≥ 7, so substitute these into the RS equation

$$\frac{2 \cdot 3}{2} + 2 \cdot 3 - \sum_{i=1}^{n} z_i \geq 7$$

$$9 - \sum_{i=1}^{n} z_i \geq 7$$

$$\sum_{i=1}^{n} z_i \leq 2$$

This gives us an upper bound for $\sum_{i=1}^{n} z_i$. Mann-Whitney calls this u, the number of times a 1 precedes a 0 in the binary string. A look back at Table 4.5 shows that $u \leq 2$ for strings of this type.

$u = 2$ becomes the first argument of the recurrence relation $w(u, m, n)$ which calculates the number of strings containing n 0's and m 1's where a 1 precedes a 0 u times.

In this example, $m = 2$ and $n = 3$. However, n cannot exceed u, so can be reduced to $n = 2$. This reduction is welcome since it shortens the number of rewrites in the recursion. So our aim is to evaluate $w(2, 2, 2)$.

The three base cases for $w()$ are:

$$w(u, m, n) = 0 \text{ for } u < 0, \quad \text{and} \quad w(0, m, n) = w(u, m, 0) = 1,$$

In the recursive case, a string counted by $w(u, m, n)$ must deal with a sequence of 0's and 1's. When a 0 is met, the n count is decremented. When a 1 is met, the m count is decremented, and the remaining number of 0's is deducted from u. In other words:

$$w(u, m, n) = w(u, m, n - 1) + w(u - n, m - 1, n);$$

This will be true even for $m = 1$ if we set $w(u, 0, n) = 1$ for non-negative u and n, which takes the number of base cases to 4.

The base cases deal with u, m, or $n = 0$, which is equivalent to a count of 1. However, if u becomes negative it only signals termination of the search, so the base case returns 0.

Fig. 4.4 shows the evaluation of w(2,2,2) using the above definitions. It returns 4, the number of 5-bit strings where RS ≥ 7, and matches our manual calculation back in Table 4.5.

Now we'll apply the same analysis to the president ages string. Recall the details generated by `readPresidents.py`:

```
Ages as binary:
35 : 00000000000010000000000011001010000
Short Positions (from 1):
```

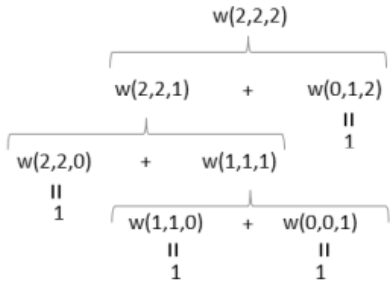


Figure 4.4. Evaluation of w(2,2,2)

```
13 25 26 29 31
Sum: 124
```

Once we've worked out the arguments for $w(u, m, n)$, we'll (happily) pass them to a program to do the counting.

The rank sum RS for the string is $13 + 25 + 26 + 29 + 31 = 124$, so we want to calculate the number of 35-bit strings where RS ≥ 124.

We start with the RS equation:

$$RS = \frac{m(m+1)}{2} + mn - \sum_{i=1}^{n} z_i$$

m is the number of 1's in the binary string ($m = 5$), and n the number of 0's ($n = 30$). Substitute these values into the equation:

$$\frac{5 \cdot 6}{2} + 5 \cdot 30 - \sum_{i=1}^{30} z_i \geq 124$$

$$165 - \sum_{i=1}^{30} z_i \geq 124$$

$$\sum_{i=1}^{30} z_i \leq 41$$

In $w(u, m, n)$, $u = 41$. m is the number of 1's in the string, so $m = 5$. n is the number of 0's, so $n = 30$, which is already less than u so doesn't change.

The straightforward translation of the $w()$ relations yields Listing 4.6 (wilcorec.py).

```
def w(u,m,n):
  if u < 0:
    return 0
  elif (u == 0) or (m == 0) or (n == 0):
    return 1
```

```
  else:
    return w(u, m, n-1) + w(u-n, m-1, n)
```

Listing 4.6. A recursive implementation of w()

Its calculation:

```
> python wilcorec.py
u m n=? 41 5 30
count = 18909
```

The total number of binary strings with 30 zeros and 5 ones is $\binom{35}{5}$, which is:

```
>>> import math
>>> math.comb(35,5)
324632
```

So the exact P-value = 18909/324632 = 0.05824749. By simulation we got P = 0.05637500.

4.5.4 Faster Implementations. Listing 4.7 (wiliter1.py) is a translation of $w()$ to use iteration.

```
u,m,n = map(int, input("u m n=? ").split())
w = [[[0 for y in range(n+1)]
       for x in range(m+1)] for z in range(u+1)]
for z in range(u+1):
  for x in range(m+1): w[z][x][0] = 1
for z in range(u+1):
  for y in range(n+1): w[z][0][y] = 1
for x in range(m+1):
  for y in range(n+1): w[0][x][y] = 1
for x in range(1,m+1):
  for y in range(1,n+1):
    for z in range(1,u+1):
      if z < y:
        w[z][x][y] = w[z][x][y-1]
      else:
        w[z][x][y] = w[z][x][y-1] + w[z-y][x-1][y]
print("w =", w[u][m][n])
```

Listing 4.7. An iterative w() using lots of memory

For larger problems it's considerably faster than the recursive function, with time complexity proportional to $u * m * n$.

Unfortunately it requires $(u + 1) * (m + 1) * (n + 1)$ cells for the 3D list $w[u][m][n]$, but this storage can be reduced considerably.

The algorithm computes $w(z, x, y)$ for $y = 0, 1, 2, \ldots$. At the start, the layer $y = 0$ is set up by assigning the value $w = 1$ to each lattice point (z, x) with $0 \le z \le u$, $0 \le x \le m$ (this follows the axis labeling in Fig. 4.5).

We can now compute the w-values at the lattice points for layers $y = 1, 2, ..., n$. Each computed value is stored in the current y layer, so the key insight is that we only need a single y layer. Thus, the recursion

$$w(z, x, y) = w(z, x, y-1) + w(z-y, z-1, y)$$

reduces to

$$w(z, x) = w(z, x) + w(z-y, x-1) \quad \text{for } z \geq y;$$

Note that the y index needs to be retained for the $z - y$ calculation, but there's no need to generate y layers. These changes are represented by Fig. 4.6.

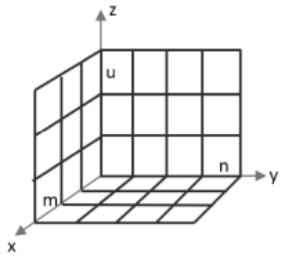


Figure 4.5. The 3D space used by w(u,m,n)

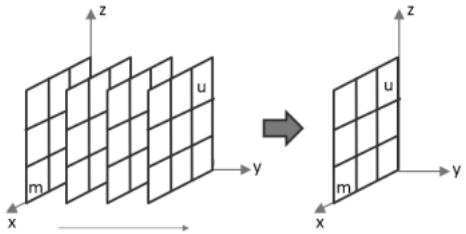


Figure 4.6. The 2D space used by w(u,m)

Listing 4.8 (wiliter2.py) implements the new algorithm.

```
u,m,n = map(int, input("u m n=? ").split())
w = [[0 for x in range(m+1)] for z in range(u+1)]
for z in range(u+1):
  for x in range(m+1): w[z][x] = 1
for y in range(1,n+1):
  for x in range(1,m+1):
    for z in range(y,u+1):
      w [z][x] = w[z][x] + w[z-y][x-1]
print("w =", w[u][m])
```

Listing 4.8. An iterative w() using less memory

4.6 The General Two Sample Test

In the presidents example we lost some information by replacing ages by ranks, so let's repeat the analysis using ages.

Suppose we take a random 5-subset of the 35 ages. What is the probability that the sum s of these ages is $\geq 67 + 79 + 80 + 85 + 90 = 401$? Let $q(n, k, s)$ be

the number of k-subsets of an n-set $\{d_1, d_2, ..., d_n\}$ of non-negative integers, whose sum is $\leq s$. Then

$$\begin{array}{rcll} q(n,0,s) & = & 1 & \text{for } s \geq 0, \\ q(n,k,s) & = & 0 & \text{for } n < k \text{ or } s < 0, \end{array} \tag{4.1}$$

and

$$q(n,k,s) = q(n-1,k,s) + q(n-1,k-1,s-d_n). \tag{4.2}$$

In words, Equ. (4.2) states that there are $q(n-1,k,s)$ subsets without d_n and $q(n-1,k-1,s-d_n)$ subsets with d_n.

To use this recursive definition, we'll need to 'reflect' the ages by subtraction (x' = maximum age $- x$) to make the ages of the short presidents small. Instead of $s \geq 67+79+80+85+90 = 401$ we'll have $s \leq 4+9+14+15+27 = 69$ (the maximum age is 94).

Equs. (4.1) and (4.2) become the $q()$ function in Listing 4.9 (TwoSample.py).

```
def q(n, k, s):
  if (k == 0) and (s >= 0):
    return 1
  elif (n < k) or (s < 0):
    return 0
  else:
    return q(n-1, k, s) + q(n-1, k-1, s-d[n-1])
```

Listing 4.9. Implementation of $q()$

The output:

```
> python TwoSample.py
n=35, k=5, s=69
P = 7.48%
```

We note that rank testing of the same data gave us a P-value which was too good (around 5.82%). In general that approach tends to overestimate tiny differences since every value gets the same weight.

Exercises for Sections 4.5 to 4.6

(1) Prove that $w(u,m,n) = w(u,n,m)$.

(2) Write iterative programs for the Two Sample Test imitating Listings 4.7 and 4.8.

(3) Find the P-value of the hypothesis in Ex. 3 at the end of Sec. 4.4 by means of Listings 4.8 and TwoSample.py.

(4) Test the presidents data with permCalc.py. You should step through all the 5-subsets of the 35 samples and count those with sums ≥ 401. Compare with the output of TwoSample.py.

(5) Apply the Two Sample test to the marijuana example. It should give the same result as permSim.py, but faster. However, you should first turn the data into non-negative integers by adding 17 to each number. The input will still be $n = 18$ and $k = 9$, but $s = 9 * 17 - 46 = 107$. To gain even more speed you should use the programs suggested in Ex. 2 above.

4.7 Kendall's Rank Correlation

Kendall's rank correlation (also known as Kendall's τ (Tau)) is used to measure the ordinal association between two sets of data. The correlation will be high when the values in the two sets have similar rank positions, and low when their ranks are dissimilar.

We'll explain how this statistic is applied by considering cancer deaths associated with the Hanford site, a decommissioned nuclear production complex operated by the US government in Hanford, Washington. Although its reactors were shut down between 1964 and 1971, engineers sent to clean up the installation discovered 54 million gallons of highly radioactive sludge. Until 1965 this waste had been stored in open pits, which leaked into the Columbia river that flows through Oregon into the Pacific. Cleanup activity are still ongoing to this day, with over 10,000 workers employed on various tasks.

Table 4.6 shows how an exposure index X compares to cancer mortality Y (per 100000 man years for 1959-1964) for nine Oregon districts. Note that the data is sorted by the exposure index.

X	1.25	1.62	2.49	2.57	3.41	3.83	6.41	8.34	11.64
Y	113.5	137.5	147.5	130.1	129.9	162.3	177.9	210.3	207.5

Table 4.6. 'Exposure and Mortality', *Journal of Environmental Health*, v. 27, 1965, 883-897.

If we replace each cancer mortality value by its rank, we get the permutation

$$p = 145326798$$

The first tabulated Y value has rank 1, the second has rank 4, the third rank 5, etc. Is this a 'random permutation'? Obviously, 123456789 and 987654321 are not random, but what about 145326798? Kendall's Tau can be used to answer this question.

Consider the $\binom{9}{2} = 36$ pairs of ranks that can be picked from p. A pair (i, j) with $i < j$ is called *concordant*, while a pair with $i > j$ is *discordant*. 123456789 has no discordants, while 987654321 has 36, but on average, we'd expect 18 concordant and 18 discordant pairs in a random permutation. We'll use the number of discordants as a way to measure 'extremeness'.

$p = 145326798$ has 30 concordants and 6 discordants, as calculated in Table 4.7.

Exposure Index	Cancer mortality rank	Concordants	Discordants
1	1	8	0
2	4	5	2
3	5	4	2
4	3	4	1
5	2	4	0
6	6	3	0
7	7	2	0
8	9	0	1
9	8	0	0
	Sum	30	6

Table 4.7. Calculating concordants and discordants for cance mortality

To count the number of concordant pairs, start with the first row of the 'Cancer mortality' column and count how many ranks below it are larger. Then move to the next row and repeat the process. To count the number of discordant pairs, start again at the top of the 'Cancer mortality' column and count how many ranks below it are *smaller*. Repeat this for every row. The 'Concordants' and 'Discordants' totals are 30 and 6 respectively, which together equals the total number of ways of selecting two pairs; in this case $\binom{9}{2} = 36$.

Carrying out a similar task for $p' = 897623541$ results in 6 concordants and 30 discordants, which is to be expected since

$$P(\text{discords}(p) \leq 6|H) = P(\text{discords}(p) \geq 30|H),$$

where

- H: It is just a random fluctuation.
- A: p has too few (or too many) discordants.

We calculate this probability in cancer.py. It generates NUM_TRIALS random 9-permutations, totals their discordants, and tests if discords(p) ≤ 6 or ≥ 30. For each occurrence of this event a counter is incremented, so that by the end the counter will record the number of 9-permutations with too many or too few discordants. This is converted to a probability by division by 9!. It's also divided by two since we're examining both ends of the distribution so the probability range is doubled.

Typical output when $n = 9$, min $= 6$, max$= 30$:

```
> python cancer.py
n min max=? 9 6 30
count == 396; P == 0.00619
```

This is saying that there's a high significance that assumption A is true.

4.7.1 Defining Kendall using Recursion. Let $p(n,k)$ be the number of n-permutations with at most k discordants. Our aim is to derive a recursive formula for $p(n,k)$ that can be readily translated into a recursive function.

Suppose we have an $(n-1)$-permutation $X_1X_2...X_{n-1}$ of {1, 2, ..., n-1}, and we want to insert the element n to make it an n-permutation. There are n possible insertion points, as indicated by the boxes in:

$$\boxed{n-1}\,X_i\,\boxed{n-2}\,X_2\,...\boxed{i}\,X_{n-i}...\boxed{2}\,X_{n-2}\,\boxed{1}\,X_{n-1}\,\boxed{0}.$$

If we insert element n into location i, then it will contributes i discordants. Our aim is to obtain $p(n,k)$, so the $p()$ value for the old $(n-1)$-permutation must be $p(n-1,k-i)$. There are n alternatives like this that can produce $p(n,k)$, leading to the formula:

$$p(n,k) = p(n-1,k) + p(n-1,k-1) + ... + p(n-1,k-n+1). \tag{4.3}$$

If we substitute $k-1$ in place of k in Equ. (4.3), we get

$$p(n,k-1) = p(n-1,k-1) + p(n-1,k-2) + ... + p(n-1,k-n). \tag{4.4}$$

The right-hand side of Equ. (4.4) is almost the same as the right-hand side of Equ. (4.3), which means that (4.3) can be simplified by substituting in the left-hand side of (4.4), giving:

$$p(n,k) = p(n-1,k) + p(n,k-1) - p(n-1,k-n). \tag{4.5}$$

Recalling that $p(n,k)$ is the number of n-permutations with at most k discordants, we specify three base cases:

$$\begin{aligned} p(n,0) &= p(1,k) = 1 \text{ for } n \geq 1, k \geq 0, \\ p(n,j) &= 0 \text{ for } j < 0 \end{aligned}$$

The recursive case in Equ. (4.5) can also be simplified when $k < n$ since the third term will have a negative argument and so contribute 0. Therefore:

$$p(n,k) = p(n-1,k) + p(n,k-1) \text{ for } k < n$$

These three base cases, and two recursive rules, become Listing 4.10 from kendallr.py.

```
def numDiscs(n,k):
  if (k == 0) or (n == 1):
    return 1
  elif k < n:
    return numDiscs(n-1, k) + numDiscs(n, k-1)
  else:
```

```
    return numDiscs(n-1,k) + numDiscs(n,k-1) - numDiscs(n-1, k-n)
```

Listing 4.10. Recursive Kendall's Tau

For $n = 9$, $k = 6$, we get:

```
> python kendallr.py
n k=? 9 6
no. discordants = 2298
p = 0.00633
```

The iterative version, kendallit.py, is faster for larger problems.

```
def numDiscs(n,k):
  nds = [[0 for j in range(k+1)] for i in range(n+1)]
  for i in range(1,n+1):
    nds[i][0] = 1
  for j in range(k+1):
    nds[1][j] = 1
  for i in range(2,n+1):
    for j in range(1,k+1):
      if j < i:
        nds[i][j] = nds[i][j-1] + nds[i-1][j]
      else:
        nds[i][j] = nds[i][j-1] + nds[i-1][j] - nds[i-1][j-i]
  return nds[n][k]
```

Listing 4.11. Iterative Kendall's Tau

4.7.2 The Permutation Test and Correlation. Let us look again at Table 4.6, but with a test statistic of our own invention, the *dot product* of two sequences X and Y:

$$T = X \cdot Y = \sum X[i]\, Y[i].$$

The data in X is always sorted into ascending order, so T will be a maximum if the elements of Y are also in ascending order, and decreases as the number of discordants in Y increases.

cancer1.py calculates the dot correlation between the X and Y sequences in Table 4.6, and then repeatedly shuffles the Y sequence and retests it against X.

A count is kept of each instance when a larger correlation is calculated, which is used to estimate the probability that a random correlation could be higher than the one obtained between X and Y.

Typical output:

```
> python cancer1.py
X: [1.25, 1.62, 2.49, 2.57, 3.41, 3.83, 6.41, 8.34, 11.64]
Y: [113.5, 137.5, 147.5, 130.1, 129.9, 162.3, 177.9, 210.3, 207.5]
dot corr = 7440.3660
count = 15; p = 0.00047
```

As with cancer.py the result is highly significant, indicating that the correlation of $Y$with X is very unusual.

4.7.3 The Sample Pearson Correlation Coefficient. One way to define the Pearson correlation coefficient r_{xy} over paired data $(x_1, y_1), ..., (x_n, y_n)$ is

$$r_{xy} = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}}$$

We're assuming that all the summations are between 1 and n. Listing 4.12 (corrCoef.py) implements this equation.

```
def dot(xs, ys):
  return sum([x*y for x, y in zip(xs, ys, strict=True)])

def corrCoef(xs, ys):
  # Assume len(xs) == len(ys)
  n = len(xs)
  xSum = sum(xs)
  ySum = sum(ys)
  xSum2 = sum([x*x for x in xs])
  ySum2 = sum([y*y for y in ys])
  xys = dot(xs, ys)
  numer = n*xys - xSum*ySum
  denom = math.sqrt((n*xSum2 - xSum*xSum) * (n*ySum2 - ySum*ySum))
  if denom == 0:
    return 0
  return numer/denom
```

Listing 4.12. The Pearson correlation coefficient

Listing 4.12 is a near duplication of the correlation() function in Python's statistics module, but that version is more versatile. For instance, if its 'method' argument is set to 'ranked', it will switch to calculating Spearman's rank correlation coefficient.

corrCoef() is utilized in cancer2.py which is identical to cancer1.py except that the two calls to dot() are replaced by calls to corrCoef(). Typical output:

```
> python cancer2.py
X: [1.25, 1.62, 2.49, 2.57, 3.41, 3.83, 6.41, 8.34, 11.64]
Y: [113.5, 137.5, 147.5, 130.1, 129.9, 162.3, 177.9, 210.3, 207.5]
pearson corr = 0.9259
count = 10; p = 0.00031
```

The significance level 0.00031 is almost the same as the dot correlation approach above. The difference is probably due to Pearson's underlying assumption that the data is normally distributed (which is implicit in its use of means and standard deviations in its summations), whereas dot correlation makes no such assumptions.

Exercises

(1) draft.csv holds US draft lottery data for 1970, which determined the order of calls to military service in the Vietnam war that year for men born between January 1, 1944 and December 31, 1950 (`https://en.wikipedia.org/wiki/Draft_lottery_(1969)`). readDraft.py can be used to load it as a list of Python dictionaries.

For the draft, 366 days of the year (including February 29) were written on slips of paper, placed in individual plastic capsules, mixed together in a shoe box(!), then emptied into a glass jar. Capsules were drawn from the jar one at a time and opened. The first number drawn was 258 (for September 14), so all registrants with that birthday were assigned rank 1. The next ranks were assigned to April 24, December 30, February 14, October 18, and so on. The last number drawn was for June 8. In draft.csv, we've replaced the birthday numbers by their selection rank. So September 14 is 001, April 24 is 002, and so on.

All of the men who shared a birthday were called to serve, and altogether 195 birthdays were utilized; the last being September 24. Thus, men with low rankings were almost certainly drafted, while those with high ranks were spared. Our aim is to show that the distribution of the ranks in draft.csv is not random.

For each month, we found the median of the ranks assigned to its days, resulting in Table 4.8.

month	1	2	3	4	5	6	7	8	9	10	11	12
median of ranks	211	210	256	225	226	207.50	188	145	168	201	131.5	100

Table 4.8. Draft ranks per month

a) Assign a rank to each median in Table 4.8, producing a permutation of the numbers 1 to 12.

b) Determine the number of concordant pairs, C. You should get $C = 11$, but the expected value is 33 (why 33?).

c) Test the hypothesis H that the rankings in draft.csv are random. Determine the probability $P(C \leq 11|H)$ and draw conclusions.

It seems very likely that the plastic capsules holding the birth dates weren't shuffled sufficiently in the shoe box. The last ones put in tended to be the first ones pulled out.

(2) The archived website at `https://web.archive.org/web/20050318170452/http://cnx.rice.edu/content/m10142/latest/` by David Lane also examines the fairness of the 1970 US draft lottery. He starts with a list of the selected birthday numbers:

```
258 115 365 45 292 250 300 251 ...  52 126 57 160
```

This data can be found in draftBithdays.txt.

He divided the 366 birth dates into three groups of 122 days each: the first third goes from January 1 to May 1, the second from May 2 to August 31, and the last from September 1 to December 31. He plots these groups as three box plots, noting that if there was no relationship between birth date and draft number, then the three boxes would line up horizontally; they don't.

Write a program to load the data from `draftBirthdays.txt` (using the functions in reader.py, split the data as Lane suggests, and use matplotlib to produce three box plots. A similar charting approach was used by readPresidents.py in Sec. 4.5.

(3) Table 4.9 gives the cholesterol levels of males in two age groups. Test the fol-

X (20-30 years)	135	222	251	260	269	235	386	252	173	156	352
Y (40-50 years)	294	311	286	264	277	336	208	346	173	254	346

Table 4.9. Cholesterol levels. Source: Dixon and Massey, *Introduction to Statistical Analysis*.

lowing hypotheses:

- H : The cholesterol levels of both groups is the same.
- A : The older group has higher cholesterol levels.

a) Find the P-value using TwoSample.py.

b) Find the P-value using permSim.py.

4.8 The Binomial Distribution

Fig. 4.7 shows a spinner where p and q $(1-p)$ are the probabilities of the outcomes 1 (success) and 0 (failure).

With this distribution, any binary word, such as 100110...01 with x ones and $n-x$ zeros, will have probability $p^x q^{n-x}$ of occurring. Since there are altogether $\binom{n}{x}$ such words, the probability of exactly x successes in n spins (trials) is

$$b(x) = \binom{n}{x} p^x q^{n-x}$$

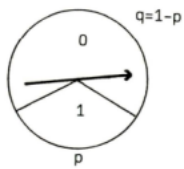


Figure 4.7. A 1 with prob p

If we make explicit that the values of n and p can change, then the equation becomes

$$b(n, p, x) = \binom{n}{x} p^x q^{n-x}, \quad x = 0, 1, 2, ..., n. \tag{4.6}$$

This, with the help of math.comb(), becomes Listing 4.13 (binomial.py).

```
def binomial(n, p, x):
  return math.comb(n, x) * p**x * (1-p)**(n-x)
```

Listing 4.13. Binomial using float

The drawback of this approach is the danger of float overflow or underflow (see Appendix E). For example:

```
>>> from binomial import *
>>> binomial(100, 0.5, 0)
7.888609052210118e-31
>>> binomial(1000, 0.5, 0)
9.332636185032189e-302
>>> binomial(1080, 0.5, 0)
0.0
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
min_10_exp=-307, dig=15, mant_dig=53,
epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

As sys.float_info states, floats can only deal with numbers between around 2×10^{-308} and 2×10^{308}.

One solution is to rewrite binomial() to use Python's decimal type. Let's switch back to the $b(x)$ form, and the transformation is more apparent if we write $b(x)$ and $b(x-1)$ in terms of factorials:

$$\begin{aligned} b(x) &= \frac{n!}{(n-x)!\,x!} p^x q^{n-x} \\ b(x-1) &= \frac{n!}{(n-(x-1))\,(x-1)!} p^{x-1} q^{n-(x-1)} \end{aligned}$$

Defining $b(x)$ in terms of $b(x-1)$:

$$b(0) = q^n, \quad b(x) = b(x-1) * r * \frac{n-x+1}{x}, \quad r = \frac{p}{q}, \quad x = 1, 2, ..., n. \tag{4.7}$$

Equ. (4.7)'s recursion can be readily translated into a loop, and floats are converted to decimals, to produce Listing 4.14 (binomial.py).

```
def binomialIterD(n, p, x):
  dx = D(x); dp = D(p)
  dq = D(1)-dp
```

```
  bin = dq**n
  for i in range(1, x):
    bin *= dp/dq * (n - dx + 1)/dx
  return bin
```

Listing 4.14. Binomial using decimals

Underflow and overflow are avoided:

```
>>> binomialIterD(100, 0.5, 0)
Decimal('7.888609052210118054117285653E-31')
>>> binomialIterD(1000, 0.5, 0)
Decimal('9.33263618503218878990089544 7E-302')
>>> binomialIterD(1080, 0.5, 0)
Decimal('7.719775716269477252758887389E-326')
```

4.8.0.1 A Confidence Interval for p. A random process is controlled by the spinner in Fig. 4.8.

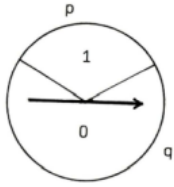


Figure 4.8. Obtain a 1 with prob p

If we carry out n spins and get X successes, then $\hat{p} = X/n$ is a *point* estimate for p. However, it's more useful usually to state a level of confidence that p lies within some range of probabilities.

Suppose we get $x = 80$ successes in $n = 200$ trials, then $80/200 = 0.4$ is the point estimate for p. Of course we could get fewer than 80 successes even if p was larger than 0.4, but as p grows this becomes less likely. Let's find an upper limit $\hat{p}_2$ for p where $P(X \leq 80) = 2.5\%$ exactly. The equation we must solve is

$$B(p) = P(X \leq 80) = \sum_{x=0}^{80} \binom{200}{x} p^x (1-p) p^{200-x} = 0.025.$$

We do this by testing different p values with Listing 4.15 (binomialSum.py).

```
import math
from binomial import *

n,p,c,d = map(float, input("n p c
    d=? ").split())
n = int(n); c = int(c); d = int(d)
    ; sum = 0
for x in range(c, d+1):
  sum += binomial(n, p, x)
print(f"{sum:.8f}")
```

Listing 4.15. Summing binomials

For example:

```
> python binomialSum.py
n p c d=? 200 0.45 0 80
0.08807268

> python binomialSum.py
n p c d=? 200 0.47 0 80
0.02751749
```

The inputs $n = 200$, $c = 0$, $d = 80$ and different p values eventually lead us to $\hat{p}_2 = 0.47147$, as shown in Table 4.10.

p	0.45	0.47	0.471	0.472	0.4714	0.47147
B(p)	0.0881	0.0275	0.0258	0.0241	0.0251	0.0250

Table 4.10. Outputs from Listing 4.15

So if $p > 0.47147$ then the probability of getting only 80 successes in 200 trials is 2.5% or less. We can also phrase this as a 97.5% confidence that $p \leq 0.47147$. So if we repeatedly carry out experiments with this confidence level, we shall be successful, in the long run, at least 97.5% of the time.

It may seem that we could have expressed the same thing more simply by saying that the probability that $p \leq 0.47147$ is at least 97.5%. However, strictly speaking, this type of statement is appropriate only when p is a random variable which can take various values. But our p has a single well defined value, we just don't know what it is. Our calculated probability doesn't refer to the value of p, but to the correctness of the statement we are making about it. That's why we use the word "confidence".

If p was much smaller than 0.4, it's unlikely that we would have got 80 or more successes. We can quantify that with the equation:

$$B(p) = P(X > 80) = \sum_{x=80}^{200} \binom{200}{x} p^x (1-p) p^{200-x} = 0.025.$$

A suitable lower limit for p can be found using Listing 4.15 but with the summation range $c = 80$ and $d = 200$. We obtain:

```
> python binomialSum.py
n p c d=? 200 0.33155 80 200
0.02500667
```

Let $\hat{p}_1 \approx 0.33155$, which means that if $p < \hat{p}_1$ then the probability of obtaining 80 or more successes is $< 2.5\%$. In other words, $\hat{p}_1 < p$ is also a 97.5% interval for p. Therefore the intersection

$$0.33155 \leq p \leq 0.47147$$

of the two intervals is a 95% confidence interval.

4.8.1 Plotting the Binomial Distribution. plotBinomial.py generates a bar plot of binomial(n, p, x) values for $0 \leq x \leq n$, which illustrates how the distribution begins to approximate a Normal distribution when n is large enough and p is close to 0.5. The result for $n = 50$ and $p = 0.5$ is shown in Fig. 4.9.

It's fairly clear that the mean of the Normal distribution is 25 (i.e np) and one standard deviation is 5 (i.e. npq).

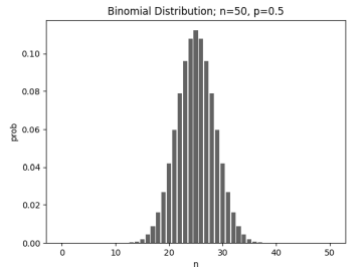
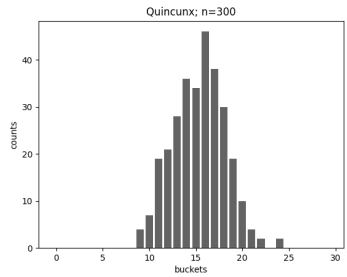


Figure 4.9. Plot of a Binomial distribution



(a) A quincunx



(b) Plot of a quincunx

Figure 4.10. Binomials

Another way of linking the Binomial and Normal distributions is by implementing a Galton board (see Fig. 4.10a), also known as a *quincunx*. Francis Galton used it to demonstrate the central limit theorem: with a sufficiently large sample size the Binomial distribution approximates a Normal distribution.

quincunx.py implements the Galton board, not by calling our binomial() function but by tallying the results of multiple random yes/no choices. This approach hopefully emphasizes that the binomial is the discrete probability distribution of the number of successes in n independent experiments, each asking a yes/no question. When $n = 200$, the resulting plot is Fig. 4.10b.

Exercises

(1) For the example in the text with $X = 80$, $n = 200$ find a confidence interval for p of a) 98%, b) 99%.

(2) For $X = 50$ and $n = 100$ find a confidence interval for p of a) 95%, b) 98%.

(3) An unknown proportion p of a population are smokers, but in a random sample of 1000 people 600 admitted to the habit. Find a confidence interval for p of a) 95%, b) 98%.

(4) We want to estimate the proportion p of left-handers among students. In a random sample of $n = 100$ there were $X = 10$ left-handers. Find 95% and 98% confidence intervals for p.

(5) 100 coin tosses results in $X = 60$ successes. To see how likely this is if the coin is fair, find the probability of the event $|X - 50| \geq 10$ for such a coin.

(6) 900 coin tosses results in $X = 486$ successes. Test the likelihood of this occurring by computing the probability of getting $X \geq 486$ when tossing a fair coin.

(7) 600 rolls of a die results in $X = 120$ sixes. How likely is this if the die is fair? What is the probability of $|X - 100| \geq 20$?

(8) **The Taxi Problem**. City taxis are usually numbered from 1 to T. We want to find a 95% confidence interval (CI) for this unknown T by observing n taxi numbers, which we've ordered according to size: $X_1 \leq X_2 \leq ... \leq X_n = M$ = the biggest observed taxi number. This equation uses $\leq$ since the same taxi may show up repeatedly.

By definition, M $\leq$ T, so we take M as the lower end of the CI. We find the upper end by using $P(\text{all } X \leq M) = (M/T)^n$. We want to exclude T values where this probability is $< 5\%$ so we set the upper end to be $T = M\sqrt[n]{20}$. This comes from:

$$\left(\frac{M}{T}\right)^n > \frac{1}{20}$$
$$\left(\frac{T}{M}\right)^n > 20$$
$$T < M\sqrt[n]{20}$$

Combining both ends of the CI, we want $M \leq T \leq M\sqrt[n]{20}$ to be wrong only for 5% of taxi-cabbed cities.

Write a program which chooses n taxi numbers at random using some T, calculates the CI, and checks if it covers T. This should be repeated 1000 times. Does the CI cover T in about 95% of the cases? How does the CI depend on n?

It's easy to calculate an almost "best point estimate" $\hat{T}$ for T. We consider the n observed gaps between the taxis: $X_1 - 1, X_2 - X_1, X_3 - X_2, ..., X_n - X_{n-1}$. The lengths of all of these gaps, including the first and the last, have the same probability distribution, so we estimate the last unknown gap, $T - M$, as the

mean of the observed ones, which is $(M-1)/n$. So rearranging

$$\begin{aligned} T - M &= \tfrac{M-1}{n} \\ T &= M + \tfrac{M}{n} - \tfrac{1}{n} \\ \hat{T} &= M(1 + \tfrac{1}{n}) - \tfrac{1}{n} \end{aligned}$$

$\hat{T}$ is being used as the *gap estimate* for T. So your code should calculate $\hat{T}$ and T, and compare them.

4.9 Hypergeometric Distribution

The hypergeometric distribution describes the probability of k successes in n draws *without* replacement, from a finite population of size N that contains exactly K objects. The binomial distribution differs in that it describes the probability of k successes in n draws *with* replacement.

The hypergeometric is identical to the one-tailed version of Fisher's exact test, named after Ronald Fisher. It's *exact* in the sense that its P-value can be calculated exactly.

One large group of experiments where the hypergeometric distribution is appropriate is the investigation of new medical treatments (e.g. new drugs) where the experiment must be **controlled**, **randomized**, and **double blind**. This means:

(1) The effects of a treatment can be judged only by comparison. We need an *experimental group E* which receives the treatment, and a control group C, which does not receive the treatment (a *controlled experiment*).

(2) The only way to exclude extraneous factors is to assign subjects to groups E and C by a random device (*randomization*).

(3) A subject mustn't know if he/she belongs to group E or C. Hence each person in C receives a placebo, which has no effect, but can't be distinguished from the new treatment. This ensures that any effects are due to the medicine and not psychological (a *blind experiment*).

(4) The doctor who measures the effects of the treatment mustn't know if a subject belongs to group C or E. This ensures that the diagnosis isn't influenced by the doctor's subconscious behavior (a *double-blind experiment*).

Does vitamin C help to prevent the common cold? This question has been studied repeatedly with contradictory results. We'll use one of the more extensive and reliable data sets (Table 4.11) from the *Canadian Medical Association Journal*, Sept. 1972, 503-508 (Toronto Study). It was a controlled, randomized, and double-blind experiment.

	cold	no cold	sum
Group E	302	105	407
Group C	335	76	411
sum	637	181	818

Table 4.11. Vitamin C and colds

The 407 subjects in group E received a vitamin C pill daily, while the 411 subjects in the control group C were given a placebo, which tasted exactly like the vitamin.

To simplify the calculations, we'll concentrate on the smallest of the four numbers in Table 4.11. How many people should we expect to see in the '76' box, which denotes untreated people who stayed healthy?

A person picked at random will fall into the second row with probability $\hat{p_1} = 411/818$ and into the second column with probability $\hat{p_2} = 181/818$. The probability of one person falling into the '76' box is the intersection of those two sets, $\hat{p_1}\hat{p_2}$. For n people, we would expect $n\hat{p_1}\hat{p_2}$ people in that box, i.e.

$$n\hat{p_1}\hat{p_2} = 818 * \frac{411}{818} * \frac{181}{818} = \frac{411 * 181}{818} \approx 91.$$

It looks as though the difference between the expected number and the actual value (91 - 76 = 15) might be a random fluctuation, but appearances are sometimes deceiving. We must test the hypotheses:

- H : Vitamin C does not help.
- A: Vitamin C helps.

If H is true then the numbers indicate that $s = 181$ of 818 people remain uninfected for reasons unrelated to vitamin C. The results are just one possible outcome of a random experiment. Instead of recording 76 in the box, we can generalize to x, and modify the other three boxes to state their results in terms of x (see Fig. 4.11).

We've also included three variables, n, r, and s in Fig. 4.11 to denote the totals for certain groups: n = the total number of people in the experiment, r = the number of people in group C (the placebo group), and s = the total number of people who did not catch a cold. These variables allows us to rewrite the table to be completely general, resulting in Table 4.12.

To test our hypothesis, we observe that if vitamin C were effective, we'd expect fewer individuals in the untreated yet uninfected category (the x box) than we would on a random basis; the uninfected people would tend to be in the treated group (the $s - x$ box). Therefore our P-value will be the probability that

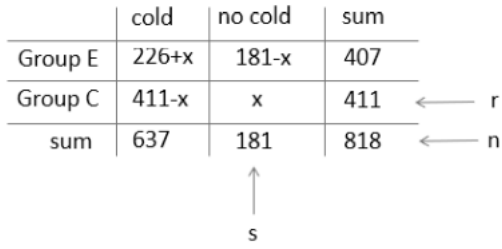


Figure 4.11. Vitamin C and colds with x untreated and healthy

	cold	no cold	sum
Group E	n-s-r+x	s-x	n-r
Group C	r-x	x	r
sum	n-s	s	n

Table 4.12. Vitamin C and colds expressed with variables

x is 76, as it was in the actual experiment, or smaller, by chance alone:

$$\text{P-value } = P(0 \le x \le 76|H) = \sum_{x=0}^{76} h(x).$$

The P-value is defined as the sum of the $h()$ function, which calculates the probability that there are x untreated people out of a total of s people who did not catch a cold. This breaks down into three counts related to people who did not catch a cold, highlighted in Fig. 4.12.

	cold	no cold	sum
Group E	n-s-r+x	s-x	n-r
Group C	r-x	x	r
sum	n-s	s	n

Figure 4.12. Counts involving people who did not catch a cold

The three counts are:

(1) $\binom{n}{s}$: the number of ways of select s (people with no colds) from n (the total number of people in the experiment).

(2) $\binom{r}{x}$: the number of ways of selecting x (the untreated, healthy people) from r (all the untreated people)

(3) $\binom{n-r}{s-x}$: the number of ways of selecting $s-x$ (the treated, healthy people) from $n-r$ (all the treated people)

h(x) combines these three counts to produce a probability for x given n, r, and s:

$$h(x) = \frac{\binom{r}{x}\binom{n-r}{s-x}}{\binom{n}{s}}, \quad h(0) = \frac{\binom{n-r}{s}}{\binom{n}{s}} = \frac{(n-r)\cdots(n-r-s+1)}{n\cdots(n-s+1)}$$

$h(0)$ follows immediately by setting $x = 0$ and considering the factorial results for the two choices in the numerator.

Listing 4.16 (leftTail.py), which employs math.comb(), follows directly from the definition for $h()$, and doesn't need to treat $h(0)$ as a special case.

```
def h(n, r, s, x):
  rs = math.comb(r, x)
  nrsx = math.comb(n-r, s-x)
  ns = math.comb(n, s)
  return (rs * nrsx)/ns

def hSum(n, r, s, x):
  return sum([h(n, r, s, i) for i in range(x+1)])
```

Listing 4.16. Summing $h()$ using floats

The inputs from the original problem $n = 818$, $s = 181$, $r = 411$, $x = 76$ yield

```
>>> from leftTail import *
>>> hSum(818, 411, 181, 76)
0.00742356691775225
```

In other words, the probability of getting 76 or less untreated, healthy people is $< 1\%$, which indicates that vitamin C definitely helps.

One drawback of this approach, which it shares with the binomial code from the last section, is it's use of Python floats, which means that underflow or overflow could be a problem, We'll address this in a similar way as before.

$h(x)$ can be defined recursively as

$$h(x) = h(x-1) * \frac{(r-x+1)(s-x+1)}{x(n-s-r+x)}$$

This is obtained by writing $h(x)$ and $h(x-1)$ in factorial form, and noting how $h(x-1)$ can be converted to $h(x)$ through various multiplications and divisions.

Although this formula is recursive, it can be easily converted to an iterative form, which we implement using Python's decimal type. The resulting functions, hD() and hSumD(), are included in leftTail.py . The inputs $n = 818$, $s = 181$, $r = 411$, and $x = 76$ yield

```
>>> hSumD(818, 411, 181, 76)
Decimal('0.007423566917752249768957316007')
```

Exercises

(1) Until near the end of the 19th century, mortality after surgery was extremely high. Even after surgeons finally switched from dirty to clean bandages, there wasn't much improvement. Finally the surgeon Joseph Lister started to use sterilized bandages. Table 4.13 shows the results of 75 amputations he performed, 35 without and 40 with sterilized bandages.

	steril.	unsteril.
survived	34	19
died	6	16

Table 4.13. Sterilized bandages and survival. From C.E.A. Winslow, *The Conquest of Epidemic Disease*. Princeton 1943, p.303.

For the hypotheses:

- H : Sterilization does not help
- A : Sterilization helps

find the P-value.

It's interesting to note that this statistical evidence did not immediately persuade other doctors to follow Lister's approach. The historical background is discussed in 'Statistics and the British controversy about the effects of Joseph Lister's system of antisepsis for surgery, 1867–1890', Ulrich Tröhler, *J R Soc Med.*, 2015 Jul; 108(7): 280–287.

(2) Modify Listing 4.16 to compute $P = h(x) + h(x+1) + ... + h(n)$, and solve the following problem. To estimate the number of trout in a pond, 1000 trout were caught, marked, and released. A few days later another 1000 trout were caught, and 100 turned out to be marked. Find a 95% confidence interval for the number N of trout in the pond.

Hint: proceed by trial-and-error to find confidence intervals for p in a similar way to in Sec. 4.8.

(3) During 1970-1974, exactly 194 patients with hip fractures were brought to a municipal hospital. Of the 73 patients delivered on Monday to Wednesday, none died. Of the 121 patients delivered Thursday to Sunday, ten died. Table 4.14 summarizes the data.

For the hypotheses:

- H : Death does not depend on the day of the week the patient was admitted

arrival	survived	died	sum
Mon-Wed.	73	0	73
Thu-Sun.	111	10	121
sum	184	10	194

Table 4.14. Hips and survival. Source: D. McNeil: 'Hip Fractures - Influence of Delay in Surgery on Mortality', *Wisconsin Medical Journal*, 74, Dec. 1975, 129-130.

- A : Patients arriving on weekends have a greater mortality rate (probably because their treatment was delayed)

find the P-value.

Why did we count Thursday as a weekend day? Surgery is treated as a last resort, so a patient arriving on a Thursday is observed for one to two days before a decision on progressing with surgery is made.

(4) **Quiet Don**. In 1965 M. Sholokhov received the Nobel Prize for literature for the novel *Quiet Don*, but rumors had been circulating in the USSR and abroad since 1928 that the novel wasn't his work. In an anonymous study by a Soviet critic the novel was attributed to the Cossak writer F. Kryukov who died of typhoid in 1920. The allegations were denounced as malicious slander in the official Soviet newspaper, *Pravda*.

In 1977 Geir Kjetsaa and others published research on the question. 1000 words were chosen at random from each of *Marking Time* (by Kryukov), *The Way and the Road* (by Sholokhov) and *Quiet Don* (?), and the number of lexemes counted. A lexeme is a single dictionary entry, so *write*, *writes*, *wrote*, *written*, and *writing* are treated as being the same. Table 4.15 shows the results.

	Words	Lexemes
Marking Time (Kryukov)	1000	589
The Way and the Road (Sholokhov)	1000	656
The Quiet Don (?)	1000	646

Table 4.15. Lexeme counts. Source: G. Kjetsaa, 'The Battle of the Quiet Don: Another Pilot Study', *Computers and the Humanities*, Vol. 11, pp. 341-346, 1977.

Since there's almost no difference between Sholokhov's writing and *Quiet Don*, Kryukov's writing was also tested against *Quiet Don*, resulting in Table 4.16.

For the hypotheses:

	Lexemes	1000 - Lexemes	Sum
Kryukov	589	411	1000
Quiet Don	646	354	1000
Sum	1235	765	2000

Table 4.16. Kryukov vs. *Quiet Don* lexeme counts

- H : Kryukov could have written *Quiet Don*
- A : Kryukov did not write it

find the P-value.

For a more recent analysis, which decides strongly in Sholokhov's favor, read the 2007 paper by Nils Lid Hjort, 'And quiet does not flow the Don' at `https://www.mn.uio.no/math/english/research/projects/focustat/news-and-events/hjort_consilience.pdf`.

4.10 Fishing Laws

Three countries, Anchuria, Celophania, and Sikinia, lie on the shores of a lake. The weights of the fish in the lake are uniformly distributed between 0 and 1, which means that the weight of the next fish caught can be conveniently determined by a call to random.random(). The inhabitants of these countries are passionate anglers, so strict laws are enforced to conserve the fish population. Fishing is allowed once a week, and each nation imposes its own rule A, C, and S, defined below, which determines when an angler must stop fishing.

Let $W_1, W_2, W_3, ...$ be the weights of the successive fish caught by an angler. The stopping rules state:

- A: An angler must stop when $W_{n-1} < W_n$.
- C: An angler must stop when $W_1 + W_2 + ... + W_n > 1$.
- S: An angler must stop when $W_n > W_1$.

The number X of fish caught by an angler is a random variable in the range $2, 3, 4, ...$

The probabilities that an angler's catch consists of *exactly n or more* fish, are defined as:

$$p_n = P(X = n),$$
$$q_n = P(X > n).$$

$E = E(X)$ is the expected number of fish in a catch.

We want to investigate each country's p_n and q_n fishing distributions, and their $E(X)$ catch expectation. We'll outline how to do this for Anchuria through

simulation (using the 'A' stopping rule), and leave the investigation of Celophania and Sikinia to you.

```
catchFreq = [0]*(MAX_FISH+1)
totFish = 0
for i in range(NUM_ANGLERS):
  prec = random.random()
  next = random.random()
  numFish = 2  # catch at least two
  while prec >= next:
    prec = next
    next = random.random()
    numFish += 1
  catchFreq[numFish] += 1
  totFish += numFish
print("  n      p         q     ")
q = 1
for i in range(2,MAX_FISH+1):
  if catchFreq[i] > 0:
    p = catchFreq[i]/NUM_ANGLERS
    q -= p
    print(f"{i:3} {p:9.5f} {q:9.5f}")
print(f"E = {totFish/NUM_ANGLERS:.5f}")
```

Listing 4.17. Simulating Anchuria's fishing

Typical output:

```
> python Anchuria.py
  n      p         q
  2   0.50229   0.49771
  3   0.33217   0.16554
  4   0.12436   0.04118
  5   0.03354   0.00764
  6   0.00652   0.00112
  7   0.00097   0.00015
  8   0.00013   0.00002
  9   0.00002  -0.00000
E = 2.71336
```

Listing 4.17 (Anchuria.py) utilizes `prec` and `next` to stand for the preceding and next fish caught, and `catchFreq` holds the frequencies of different size catches.

Based on this output, we guess that $q_n = \frac{1}{n!}$, while E seems very close to $e = 2.71828182$.

Exercises

(1) Write the corresponding program for rule C.

(2) Write the corresponding program for rule S.

The output from your 'C' program should be quite similar to A's results. However, for the data produced by rule S, it's fairly easy to guess q_n and p_n, but not the expectation. For instance, when we ran our 'S' program ten times, the reported E values were 10.84, 46.56, 90.92, 10.60, 214.85, 132.89, 22.06, 14.44, 47.59, and 26.27.

4.10.0.1 Math to the Rescue. It's time to prove our observations.

For $n \geq 2$, $p_n = P(X = n)$ and $q_n = P(X > n)$. This implies that $q_n = 1 - \sum_{i=2}^{n} p_i$, and that $p_n = q_{n-1} - q_n$.

For rule A, $q_n = X > n$ is the situation when $W_1 > W_2 > ... > W_n$. The probability of such a decreasing sequence of length n is $\frac{1}{n!}$. Thus

$$q_n = \frac{1}{n!}, \quad p_n = q_{n-1} - q_n = \frac{n-1}{n!}, \quad np_n = \frac{1}{(n-2)!},$$
$$E(X) = \sum_{n\geq 2} \frac{1}{(n-2)!} = 1 + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + ... = e.$$

Now let's tackle rule S. Clearly $X > n$ when $W_1 = \max(W_1, ..., W_n)$, and the probability of this is $q_n = 1/n$. Thus

$$p_n = q_{n-1} - q_n = \frac{1}{n(n-1)}, \quad np_n = \frac{1}{n-1}, \quad n = 2, 3, 4, ...$$

and

$$E(X) = \sum_{n\geq 2} np_n = 1 + \frac{1}{2} + \frac{1}{3} + ...,$$

which is the harmonic series, and so infinite. This explains why the reported expectations for our 'S' code were so erratic – we were trying to approximate infinity.

Finally, rule C. The probability $q_n = W_1 + ... + W_n \leq 1$ can be treated as the volume of an n-dimensional tetrahedron. The following reasoning shows that the volume V_n of this shape is $\frac{1}{n!}$.

Cavalieri's Principle (`https://en.wikipedia.org/wiki/Cavalieri%27s_principle`) implies that the volume of an n-dimensional pyramid is C_n(base)(height), where the constant C_n depends only on n. We determine C_n by decomposing a cube with unit length edges into $2n$ pyramids with the $2n$ faces as their bases and the center of the cube as the apex vertex of all the pyramids. This means that $2nC_n\frac{1}{2}1 = 1$ and hence that $C_n = \frac{1}{n}$.

This is probably easiest to visualize in 3D (i.e. when $n = 3$). Fig. 4.13 illustrates how six ($2n$) square pyramids fit together to form a cube, with their bases as the cube's six sides.

Fig. 4.13 comes from the website 'The Volume of a Hyperpyramid' (`http://physicsinsights.org/pyramids-1.html`) which does a nice job of explaining how the volume of an n-dimensional square pyramid is calculated.

The points satisfying $W_1 + ... + W_n \leq 1$ can be treated as an n-dimensional hyperpyramid with the W_n vertex as its apex and the other $(n-1)$-points forming its base. The base coordinates also satisfy $W_1 + ... + W_{n-1} \leq 1$ which means that they define a $(n-1)$-dimensional hyperpyramid. This is easiest to understand by thinking of a 3D tetrahedron; its base is a triangle, a 2D hyperpyramid.

This recursive nature for hyperpyramids leads to a recursive definition for their volume: $V_n = \frac{1}{n}V_{n-1}$. This and $V_1 = 1$ imply that $V_n = \frac{1}{n!}$.

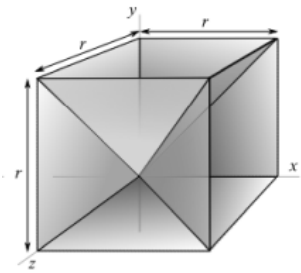


Figure 4.13. Six square pyramids forming a cube

Exercise

Antithetic variables. If U_1 is uniformly distributed in (0,1), then $U_2 = 1 - U_1$ has the same distribution. Suppose the count X in rule C is based on U_1 and Y is based on U_2. If X is below its expected value then Y will be above. The average $(X + Y)/2$ is also an estimate of e with a lower spread (or *variance*). Write a program which uses this idea to estimate e; it helps to realize that min(X,Y) always equals 2.

4.11 Probability of Not Being Lonesome

Let $M = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ be a set of ten people. Each person randomly chooses two of the others as friends. A person chosen by nobody will be deemed "lonesome" (even if other people are his friends). Find the expected number of lonesome people, and the probability that nobody is lonesome.

lonesome.py performs friend generation multiple times, and tallies the results.

Every person is given two friends, who are recorded in the friends list. After everyone has been 'friended', the list is examined for anyone who is still not a friend of someone else, and the `numLone` counter is incremented. Alternatively, if there are no lonely people in the list then `numNotLone` is incremented.

Typical output:

```
> python.exe lonesome.py
Not Lonesome = 2689 ; Lonesome = 10546
Prob Not Lonesome = 0.2689
Expected No. Lonesome = 1.0546
```

We can easily derive the expected number of lonesome people, but formulating the probability of not being lonesome is messy.

Consider any person, say #0. Any other person does not choose #0 with probability $(8/9)(7/8) = 7/9$. Therefore none of the nine others chooses #0 with probability $(7/9)^9$. The expected number E of lonesome people is np with $n = 10$ and $p = (7/9)^9$, so $E = 10 * (7/9) \approx 1.041597$.

The number of choices which leave nobody lonesome can be expressed by means of the "inclusion-exclusion principle" [**Niv65**]. We find P(nobody is lonesome) = 0.274293753....

4.12 Runs

The binary sequence

$$W = 10101101001011000101100101011000101010011001010100$$

consists of 36 runs (i.e. blocks of successive equal digits), made up of 18 runs of ones and 18 runs of zeros. It was concocted by a student to represent 50 coin tosses without bothering to use a coin, and appears to be realistic – it is quite irregular, utilizing 23 ones and 27 zeros. Yet a simple test reveals the student to be a miserable "forger", because his sequence has too many runs.

How many runs should we expect in an n-sequence?

Let $E(k)$ be the expected number of runs in a series of k coin tosses. $E(k+1)$ will only depend on the value of the previous coin toss k and and the current one, $k+1$. If the k-th and $k+1$-th tosses are different, then $E(k+1)$ will increase by 1. We translate this behavior into an equation:

$$E(k+1) = p(pE(k) + (1-p)(1+E(k)) + (1-p)((1-p)E(k) + p(1+E(k)))$$

where p is the probability of a head (or 1 or 'H'). The expression represents the four possibilities for the k-th and $k+1$-th tosses (i.e. HH, HT, TT, TH).

Expanding, and rearranging, gives

$$E(k+1) = 2p(1-p) + E(k)$$

The initial coin toss always starts a run, so

$$E(1) = 1$$

This allows us to rewrite the recursive definition as

$$E(k+1) = 2p(1-p)k + 1$$

or

$$E(n) = 1 + 2(n-1)p(1-p)$$

If our coin is fair, then $p = 1/2$, so

$$E(n) = 1 + (n-1)/2 = (n+1)/2.$$

For $n = 50$ we have $E(n) = 25.5$, but the W sequence has $R = 36$ runs. What's the probability of R $\geq$ 36 with $n = 50$ tosses? Listing 4.18 (runtest.py) finds an answer by simulation.

```
NUM_TRIALS = 10000

n, runs = map(int, input("n runs=? ").
    split())
count = 0
for i in range(NUM_TRIALS):
  numRuns = 1
  prevFlip = random.choice(['H', 'T'])
  for j in range(n-1):
    flip = random.choice(['H', 'T'])
    if flip != prevFlip:
      numRuns += 1
    prevFlip = flip
  if numRuns >= runs:
    count += 1
print(f"Prob of Runs >= {runs} = {(
    count/NUM_TRIALS):.5f}")
```

Listing 4.18. Probability for run numbers

A few different outputs for $n = 50$ and runs $= 36$:

```
> python.exe runtest.py
n runs=? 50 36
Prob of Runs >= 36 = 0.00200

> python.exe runtest.py
n runs=? 50 36
Prob of Runs >= 36 = 0.00230

> python.exe runtest.py
n runs=? 50 36
Prob of Runs >= 36 = 0.00120
```

In other words, the 36 runs in the student's W sequence is very unlikely.

The code repeats the experiment multiple times, each time generating n bits and totaling the coin changes. If the total is greater than our expected run count then a counter is incremented, which is used to generate a probability $P(R \geq$ runs) at the end.

As an alternative to Listing 4.18 we could have reused Listing 4.15. We rephrase the problem as "What's the probability of getting 35 or more successes in 49 tosses of a fair coin?", which becomes a call to binomialSum.py with $n = 49, p = 0.5, c = 0, d = 14$:

```
> python.exe binomialSum.py
n p c d=? 49 0.5 0 14
0.00190083
```

A similar result.

4.12.1 The Expected Length of the Longest Run. This time we'll generate n random bits one by one and keep track of the length of the longest run.

Let's consider each run of heads in a sequence as the number of consecutive heads beginning with the first toss or immediately after a tail occurs. In total, there will be approximately nq head runs, the expected number of tails.

Around nqp of these head runs will contain at least one head, nqp^2 will be at least two heads long, and so on. As long as nqp^x is greater than or equal to one, at least one run of length x or more can be expected. Eventually for larger values of x, nqp^x will fall below 1, which means that getting a run as long as x is unlikely. Hence we should find ER_n, the expected length of the longest head run, when $nqp^{ER} \approx 1$. Rearranging, we get $ER_n \approx \log_{1/p}(nq)$. For our fair coin,

$p = 1/2, ER_n \approx \log_2 n - 1$. So for $n = 50$ (the length of W), we would expect ER_n to be near 4.6. A more accurate analysis produces:

$$ER_n = \log_{1/p}(nq) + \gamma/\ln(1/p) - 1/2$$
$$\text{Var } R_n = (\pi^2/6)\ln^2(1/p) + 1/12$$

where $\gamma = 0.5777...$ is Euler's constant, and Var is the variance. With a fair coin, these become

$$ER_n = \log_2(n/2) + \gamma/\ln 2 - 1/2 \approx \log_2 n - 2/3$$
$$\text{Var } R_n = (\pi^2/6)\ln^2 2 + 1/12 \approx 3.5070$$

The variance is remarkable since it's independent of n, so the standard deviation of even a very long run is still only approximately 1.873.

For more details on this analysis, an excellent resource is 'The Longest Run of Heads' by Mark F. Schilling, *The College Mathematics Journal*, Vol. 21, No.3, 1990, pp.196-207 (available at `https://maa.org/sites/default/files/pdf/upload_library/22/Polya/07468342.di020742.02p0021g.pdf`).

Using this improved (but still approximate) equation, when $n = 1024$ we have $ER_n \approx 9.333$, which we test with Listing 4.19 (`maxruntest.py`).

```
n = int(input("n=? "))
maxRuns = []
for j in range(NUM_TRIALS):
  maxRun = 0; count = 0
  while count < n:
    i = 0
    while (random.choice(['H','T']) ==
          'H') and (count < n):
      i += 1; count += 1
    if i > maxRun:
      maxRun = i
  print(f"{maxRun:2}", end=' ')
  if (j+1)%20 == 1:
    print()
  maxRuns.append(maxRun)
print()
print(f"Mean maxrun = {statistics.mean
    (maxRuns):.2f}")
print(f"Stdev = {statistics.stdev(
    maxRuns):.2f}")
```

Listing 4.19. Length of the longest run

A typical run:

```
> python.exe maxruntest.py
n=? 1024
10
10  8 10 14  9  8  9  8 12 14
 8  8 10 12 15  8 12  8 10  9
  : # more lines, not shown
 8 10 10 11  9 16  9 13 13 12
11  9 13 10 13  9 12  8  8
Mean maxrun = 10.35
Stdev = 1.91
```

4.12.2 Time on the Winning Side. Contrary to popular notions, it's quite likely that in a long coin-tossing game one of the players remains on the winning

side practically the whole time. By winning we mean that the player's total number of heads (or tails) is almost always greater than their opponent's number of tails (or heads).

Most people would say that the time spent winning would be shared roughly equally. This seems to fit with the law of large numbers which states that as the number of trials increases, the proportion of heads vs. tails will converge on the expected value of 0.5.

largeFlips.py confirms this latter point (see the left hand chart in Fig. 4.14). However, it also plots the absolute difference between the numbers of heads and tails which does *not* remain close to 0.

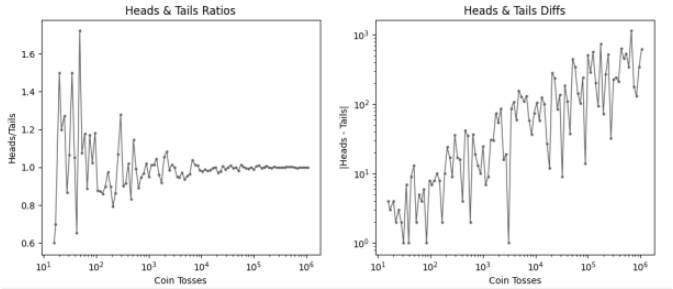


Figure 4.14. Plotting head/tail ratio and head/tail difference

The printed output reinforces the graphical results:

```
> python largeFlips.py
Final abs diff: 1018
Final ratio: 1.001943567802451
```

The 'Heads and Tails Diffs' graph in Fig. 4.14 uses logarithmic scales to highlight the linear relationship between the axes, which means there's a power relationship in the original data. One way to understand this is to think about coin tosses as a 1-dimensional random walk, starting from the origin. A 'H' moves the position one step to the right, and 'T' moves one step to the left. Since forward and backward steps are equally likely, the expected average finishing position must be back at the origin, but the interesting question is how far away from the origin, on average, we might expect to reach, regardless of direction.

We can ignore the direction issue by computing the expected value of the *square* of the distances from the origin, then take its square root, the "root mean square" or rms distance. In general, this distance from the origin, after a random walk of n steps, will be $\sqrt{n}$, and is the power law displayed in the 'Heads and Tails Diff' chart. As the graph shows, after 1 million coin tosses, the absolute difference between the number of heads and tails is on average about 1000.

This difference between the number of heads and tails grows as the number of coin tosses increases, which means that a player who gains a slight advantage over their opponent in terms of their number of heads (or tails) is likely to see that advantage grow.

headsDiffs.py visualizes what this growing advantage can mean by generating data for the number of heads - number of tails (i.e. the head's advantage) over a long series of coin flips, as the graph in Fig. 4.15.

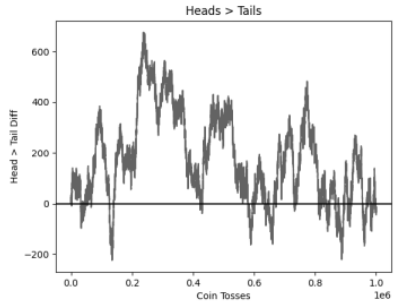


Figure 4.15. Plotting heads > tails

The program also calculates the percentage time (in terms of the fraction of the total number of coin tosses) that heads is in the lead. The following shows a few of the outputs for different runs:

```
> python headsDiffs.py
Final heads > tails diff: -44
% Times that Heads > Tails:
Pos: 84.02%; Neg: 15.98%

> python headsDiffs.py
Final heads > tails diff: 258
Pos: 82.13%; Neg: 17.87%

> python headsDiffs.py
Final heads > tails diff: 632
Pos: 59.53%; Neg: 40.47%

> python headsDiffs.py
Final heads > tails diff: 746
Pos: 19.29%; Neg: 80.71%

> python headsDiffs.py
Final heads > tails diff: 548
Pos: 25.30%; Neg: 74.70%
```

Only one of the five runs reports anything near an equal balance between the numbers of heads and tails during the game, while two favored heads, and two favored tails.

headsLeads.py automates the repeated calls to headsDiffs.py (10000 times) and stores the heads lead over the tails (if there is one) for each trial. After all the runs, the number of occurrences of each lead value is tallied and displayed in a bar chart. Fig. 4.16 displays the distribution of the heads lead.

The graph shows that the player collecting heads will most probably find himself in one of two situations – miserably far behind the player collecting tails (this corresponds to the left-hand side of Fig. 4.16) or happily way in front of his opponent (that's the right-hand side). Most surprisingly, the least likely situation is when the two players have about the same score. Fig. 4.16 displays data for 80 flips/trial, so the 'balanced' case is when head has a 'lead' of 40, which means that the tails player has a similar lead.

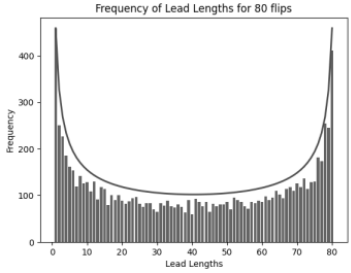


Figure 4.16. The heads lead distribution

The same kind of graph will be produced if headsLeads.py is modified to record the tails lead rather than the heads (simply change the if-test that checks the random choice from 'H' to 'T'). This reinforces the fact that the coin is fair, since both players have the same lead distribution.

In conclusion, it's quite likely that in a long coin-tossing game one of the players remains on the winning side almost all the time.

The curve drawn over the bar chart in Fig. 4.16 is for the function:

$$f(x) = \frac{1}{\pi\sqrt{x(1-x)}}$$

which fits the lead distribution curve remarkably well. It can be readily integrated to obtain an approximation to the cumulative distribution:

$$P(\frac{L_n}{n} \leq x) = \frac{2}{\pi} \arcsin\sqrt{x}$$

where L_n is the lead after n coin tosses. This is known as the arcsine law or 'Law of Long Leads'. For more information, the best source is Chapter 3 of Feller [**Fel68**]. A great online read is 'What is the Arcsine Law?' by Ethan Ackelsberg at `https://math.osu.edu/sites/math.osu.edu/files/What_is_2018_Arcsine_Law.pdf`.

4.12.3 How Long Before we Get HHHH?. A fair coin with faces 'H' and 'T' is flipped until HHHH occurs in the run. Let X be the number of flips we have to make before this situation arises.

We can represent the sequence of coin tosses using a directed graph (Fig. 4.17) like the ones in Appendix A. If the sequence produces a 'T' then the graph traversal returns to 'Start'.

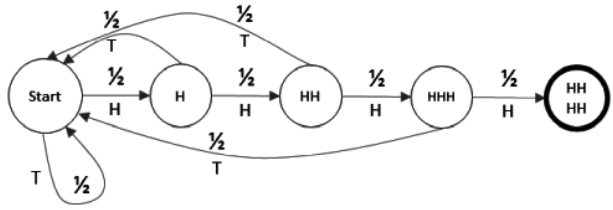


Figure 4.17. Transition probabilities for getting to HHHH

We want to find the probability P_n that exactly n flips will be required to reach HHHH:

$$(i) \quad P_n = P(X = n), \quad (ii) \quad E(X) = \sum_{n \geq 1} n * p_n.$$

Fig. 4.17 shows that

$$p_i = 0 \text{ for } i = 1, 2, 3, \quad p_4 = \frac{1}{16}, \quad p_5 = \frac{1}{32},$$

If we assume p_n is the success state (HHHH), reached after n tosses, then the preceding states HHH, HH, and H are p_{n-1}, p_{n-2}, and p_{n-3} respectively. Therefore, the graph structure implies that

$$p_n = \frac{1}{2}p_{n-1} + \frac{1}{4}p_{n-2} + \frac{1}{8}p_{n-3} + \frac{1}{16}p_{n-4}, \quad n \geq 5.$$

Creating a new equation by replacing n by $n-1$, and subtracting the two, we get

$$p_n = p_{n-1} - \frac{p_{n-5}}{32}, \quad n \geq 6.$$

This recurrence is also valid for $n = 5$ if we define $p_0 = 1$. Then

$$p_n = p_{n-1} - \frac{p_{n-5}}{32}, \quad n \geq 5.$$

`expectHHHH.py` is an iterative program that uses this equation to find E(X). For $n = 1000$ we get the expected number of flips = 30.0.

4.12.4 Which is More Likely: HH or TH?. When the tossing of two coins is considered as a single event, the probability of getting HH or TH is of course the same, $\frac{1}{4}$. However, if we ask this question in the context of coin toss *runs*, the answer is very different. We now ask:

> Suppose you were to keep flipping a coin until it landed either HH or TH on two consecutive flips. Which of these two sequences would you predict would occur first?

It turns out that TH is three times more likely to appear first. Listing 4.20 (`countPairs.py`) shows how this can be simulated.

```
thCount = 0; hhCount = 0; cnts = []
for j in range(NUM_TRIES):
  prevFlip = random.choice(['H', 'T'])
  while True:
    flip = random.choice(['H', 'T'])
    if prevFlip == 'T' and flip == 'H':
      thCount += 1; break
    elif prevFlip == 'H' and flip == 'H':
      hhCount += 1; break
    else:
      prevFlip = flip
print(f"TH%:{(thCount/NUM_TRIES):.2%};HH%:{(hhCount/NUM_TRIES):.2%}")
```

Listing 4.20. HH vs. TH

The output:

```
> python countPairs.py
TH%: 74.94%; HH%: 25.06%
```

This strange behavior is best understood by considering Fig. 4.18.

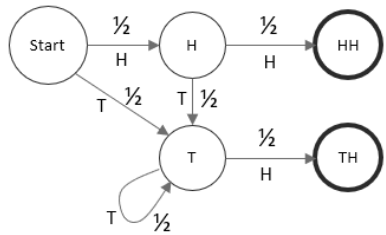


Figure 4.18. Transition probabilities for getting to HH and TH

It's clear that the probability of reaching HH is $\frac{1}{4}$. The probability for reaching TH can be obtained in two stages. The first is to realize that once the player enters the 'T' state, then there's no other possibility but to (eventually) reach TH, and so we can assign a probability of 1 to that state. There are two paths to the 'T' state – in two steps via the 'H' state or directly from 'Start'. Therefore the probability of reaching TH is $\frac{1}{4} + \frac{1}{2} = \frac{3}{4}$.

More complicated (and entertaining) examples of this kind can be found in 'Penney Ante: Counterintuitive Probabilities in Coin Tossing', by R.S. Nickerson, *The UMAP Journal*, 28.4, 2007, pp.503-532.

Exercises

(1) Write a program with inputs n, r which computes the mean waiting time for a sequence with r ones in a row. Devise a formula for the time.

(2) A fair coin with faces 0 and 1 is flipped until the word 1001 occurs for the first time. Let X be the number of flips until that arises. Draw a graph similar to Fig. 4.17 and find

$$p_n = P(X = n) \quad \text{and} \quad E(X) = \sum_{n \geq 1} n * p_n.$$

4.13 Waiting for a Complete Set

Spin the spinner in Fig. 4.19 until all the numbers from 1 to n have occurred.

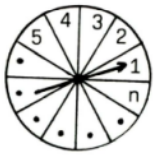


Figure 4.19. n possible values

Let S be the number of spins to complete the set, which is also known as the *coupon collector's problem*. We want to estimate the expected number $E(S)$ by simulation. Listing 4.21 (coupons.py) is the program for this equiprobable case.

```
def collect(n):
  nTries = 0; collected = set()
  while len(collected) < n:
    coupon = random.randint(1, n)
    nTries += 1
    collected.add(coupon)
  return nTries

if __name__ == "__main__":
  n = int(input("n=? "))
  tot = 0
  for i in range(10):
    spins = collect(n)
    print(spins, end = ' ')
    tot += spins
  print()
  print("Avg:", tot/10)
```

Typical output for $n = 6$ and $n = 10$:

```
> python coupons.py
n=? 6
15 13 11 16 15 8 19 15 9 10
Avg: 13.1

> python coupons.py
n=? 10
43 27 27 26 19 17 30 22 36 25
Avg: 27.2
```

Listing 4.21. How long to collect everything?

coupons.py always carries out 10 trials for a given n, reporting the total number of spins needed to complete the collection for each trial, and the average.

This type of coupon collection (where each item has equal probability of being selected) is easy to analyze. Problem 2 of Appendix A utilizes a directed graph to show that

$$E(S) = n(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}) \approx n \log n. \tag{4.8}$$

For $n = 6$ and $n = 10$ we get $E(S) = 14.7$ and 29.29, respectively.

Another approach uses the fact that the problem has a geometric distribution since the sequence of independent trials (i.e. spins of the pointer), involve only two outcomes (failure or success). If the probability of success on each trial is p, then the probability that the k-th trial is the first success is

$$Pr(X = k) = (1 - p)^{k-1} p$$

We'll use the fact that a geometric random variable X has the following properties:

$$\begin{aligned} E(X) &= 1/p \\ \text{Var}(X) &= (1 - p)/p^2 \end{aligned}$$

Let s_i be the spin that collects the i-th coupon after $i - 1$ coupons have been obtained already. Then $S = s_1 + \cdots + s_n$, and the probability of collecting a new coupon is

$$p_i = \frac{n - (i - 1)}{n}.$$

Therefore, s_i has a geometric distribution with expectation

$$\frac{1}{p_i} = \frac{n}{n - i + 1}.$$

By the linearity of expectations we have:

$$\begin{aligned} E(S) &= E(s_1 + s_2 + \cdots + s_n) \\ &= E(s_1) + E(s_2) + \cdots + E(s_n) \\ &= \frac{1}{p_1} + \frac{1}{p_2} + \cdots + \frac{1}{p_n} \\ &= \frac{n}{n} + \frac{n}{n-1} + \cdots + \frac{n}{1} \\ &= n \cdot \left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{1}\right) \\ &= n \cdot H_n. \end{aligned}$$

H_n is the n-th harmonic number, such that:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}.$$

Using H_n's asymptotics:

$$E(S) = n \cdot H_n \approx n \log n + \gamma n + \frac{1}{2} \tag{4.9}$$

where $\gamma = 0.5772...$ is the Euler constant. This confirms that the expected number of trials needed to complete a set of size n is $O(n \log(n))$ as we saw in Equ. (4.8).

The variance is determined in a similar way:

$$\begin{aligned}
\mathrm{Var}(S) &= \mathrm{Var}(s_1 + s_2 + \cdots + s_n) \\
&= \mathrm{Var}(s_1) + \mathrm{Var}(s_2) + \cdots + \mathrm{Var}(s_n) \\
&= \frac{1-p_1}{p_1^2} + \frac{1-p_2}{p_2^2} + \cdots + \frac{1-p_n}{p_n^2} \\
&< \left(\frac{n^2}{n^2} + \frac{n^2}{(n-1)^2} + \cdots + \frac{n^2}{1^2}\right) \\
&= n^2 \cdot \left(\frac{1}{n^2} + \frac{1}{(n-1)^2} + \cdots + \frac{1}{1^2}\right) \\
&< \frac{\pi^2}{6} n^2
\end{aligned}$$

since

$$\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \cdots + \frac{1}{n^2} + \cdots,$$

the solution to the *Basel Problem* (see Sec. 6.7.7).

`couponsHarm.py` tests whether $E(S) = n \cdot H_n$ by running each coupon collection trial between $n = 1$ and $n = 100$ a hundred times, and plotting the average against the curve for $n \cdot H_n$.

Fig. 4.20 confirms that the expected values do closely follow the $n \cdot H_n$ curve.

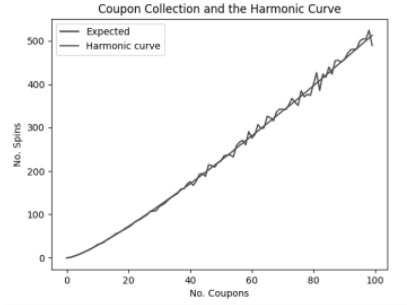


Figure 4.20. Coupon collection is harmonious

4.13.1 Unequal Probabilities. The case of unequal probabilities for obtaining different coupons is more complicated. It's still possible to give a closed formula similar to the one in Equ. (4.8), but the formula size grows proportionally with $n!$. To show you what this means we've calculated the exact formulas for the spinners in Fig. 4.21, where $n = 3$ and $n = 4$.

The relations $z + y + ... = 1$ could be used to eliminate one of the variables but the formulas are simpler if we don't do that.

For Fig. 4.21a, with $n = 3$, we get for $E(S)$

$$1 + \frac{x}{1+x}(1 + \frac{y}{z} + \frac{z}{y}) + \frac{y}{1-y}(1 + \frac{x}{z} + \frac{z}{x}) + \frac{z}{1-z}(1 + \frac{x}{y} + \frac{y}{x}),$$

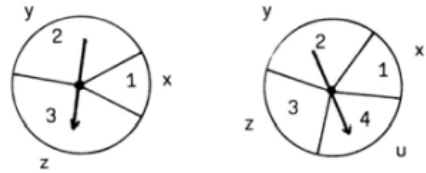


Figure 4.21. More complex coupon collection

For Fig. 4.21b, with four probabilities x, y, z and u, we get for $E(S)$

$$1+\frac{x}{1-x}\left[1+\frac{y}{z+u}(1+\frac{u}{z}+\frac{z}{u})+\frac{z}{y+u}(1+\frac{y}{u}+\frac{u}{y})+\frac{u}{y+z}(1+\frac{y}{z}+\frac{z}{y})\right]$$
$$+\frac{y}{1-y}\left[1+\frac{z}{u+x}(1+\frac{x}{u}+\frac{u}{x})+\frac{u}{z+x}(1+\frac{z}{x}+\frac{x}{z})+\frac{x}{z+u}(1+\frac{z}{u}+\frac{u}{z})\right]$$
$$+\frac{z}{1-z}\left[1+\frac{u}{x+y}(1+\frac{y}{x}+\frac{x}{y})+\frac{x}{u+y}(1+\frac{u}{y}+\frac{y}{u})+\frac{y}{u+x}(1+\frac{u}{x}+\frac{x}{u})\right]$$
$$+\frac{u}{1-u}\left[1+\frac{x}{y+z}(1+\frac{z}{y}+\frac{y}{z})+\frac{y}{x+z}(1+\frac{x}{z}+\frac{z}{x})+\frac{z}{x+y}(1+\frac{x}{y}+\frac{y}{x})\right]$$

For $n = 10$ the exact formula would fill hundreds of volumes, and would be of no practical use. Simulation is the only feasible approach for obtaining numerical results for these problems.

Exercises

(1) *Collecting two complete sets.* A spinner can land on $1, 2, ..., n$ with equal probability. What is the expected number of spins until all the numbers from 1 to n have occurred at least twice? Run a simulation for $n = 6$ and $n = 10$.

(2) Extend Listing 4.21 so that it generates 100 complete sets and calculates the mean number of spins. Run the program for $n = 6$ and $n = 10$ and compare the results with the exact values 14.7 and 29.29.

(3) Write a program for the complete set problem where you can vary both the outcome i and its probability p_i. Run it for $n = 10$ where $p_i = \frac{1}{i} - \frac{1}{i+1}$ for $i = 1, 2, ..., 9$, but $p_{10} = \frac{1}{10}$.

(4) Rooks (castles) are placed on the squares of an $n \times n$ chessboard one at a time and with the choice of any square the same $1/n^2$. This continues until all the squares on the board are threatened by the rooks. This version allows the repeated choice of a square, so it's possible that a square might end up containing several rooks.

Find the expected number of rooks when the program is run with $n = 8$. Theory predicts that $E_M \approx 17.12865$.

(5) (Continuation.) Suppose now that each rook placement is only permitted in an unoccupied square. Calculate the expected number of rooks needed to threaten all the squares. Theory predicts $E_M \approx 15.0029$.

(6) *A total tour of a regular n-gon.* A beetle starts at a vertex of a regular n-gon and goes for a random walk along its edges. Find by simulation the expected time for the bug to complete a total tour of all the edges. (The beetle completes one edge in one time unit.) *Hint*: consider the $n = 3, 4, 5, ...$ cases in turn, and then guess a formula.

(7) Start with n coins. Toss all the coins at once, and eliminate those that land showing a head. Find by simulation the expected number E(n) of tosses until all the coins have been eliminated. You should get the following results: $E(3) = 22/7$, $E(4) = 368/105$, $E(15) = 5.287$. Also have a look at Ex. 2 of Sec. 5.4.

4.14 The Birthday Problem (and Factoring)

The birthday problem asks for the probability that in a set of n randomly chosen people at least two will share a birthday. The birthday paradox refers to the counterintuitive fact that only 23 people are needed for that probability to exceed 50%.

Given n people, the probability P_n, that there are *no* common birthdays among them is

$$P_n = \left(1 - \frac{1}{365}\right)\left(1 - \frac{2}{365}\right)\cdots\left(1 - \frac{n-1}{365}\right)$$

The first factor is the probability that two given people do not have the same birthday. The second factor is the probability that a third person does not have a birthday in common with either of the first two. This continues until the last factor is the probability that the n-th person does not have a birthday in common with any of the other $n - 1$ people.

We want $P_n < 1/2$, which can be determined by examining the output from Listing 4.22 (birthday.py).

```
NUM_PEOPLE = 30

prob = 1
for i in range(1, NUM_PEOPLE+1):
  prob *= (1-((i-1)/365))
  print(f"p({i:2})={prob:.4f} ", end = ' ')
  if i%4 == 0:
    print()
```

Listing 4.22. Calculating P_n up to $n = 30$

The output:

```
> python birthday.py
p( 1)=1.0000  p( 2)=0.9973  p( 3)=0.9918  p( 4)=0.9836
p( 5)=0.9729  p( 6)=0.9595  p( 7)=0.9438  p( 8)=0.9257
p( 9)=0.9054  p(10)=0.8831  p(11)=0.8589  p(12)=0.8330
p(13)=0.8056  p(14)=0.7769  p(15)=0.7471  p(16)=0.7164
p(17)=0.6850  p(18)=0.6531  p(19)=0.6209  p(20)=0.5886
p(21)=0.5563  p(22)=0.5243  p(23)=0.4927  p(24)=0.4617
p(25)=0.4313  p(26)=0.4018  p(27)=0.3731  p(28)=0.3455
p(29)=0.3190  p(30)=0.2937
```

There must be at least 23 people in a room in order for the odds to favor at least two of them having the same birthday.

This answer is much smaller than most people expect. Even for $n = 30$, where the odds of a common birthday increase to 70.6%, most people still find it hard to believe.

One reason may be that they are asking themselves a different question, namely, "How many people need to be present for there to be a 50% chance that someone else has *my* birthday?" The answer to that question *is much* larger than 23. The probability that no one out of n people has a birthday on a given day is $(1 - 1/365)^n$. If we want a value for n where the probability exceeds 0.5 that at least one other person has your birthday:

$$\begin{aligned}(1 - \tfrac{1}{365})^n &\geq \tfrac{1}{2} \\ n \cdot \log(\tfrac{364}{365}) &\geq \log(\tfrac{1}{2}) \\ n &\geq 252.65\end{aligned}$$

We can generalize the birthday problem to be about finding n people who share a birthday over N different days:

$$P_n = \left(1 - \frac{1}{N}\right)\left(1 - \frac{2}{N}\right)\cdots\left(1 - \frac{n-1}{N}\right)$$

It will be useful later to have a simple approximation which links n to N. Given a large number of people, n, there are $\binom{n}{2} = \frac{n(n-1)}{2} \approx n^2/2$ pairs. The probability that a given pair has the same birthday is $1/N$, so the probability that they do not have the same birthday is $1 - (1/N)$. Therefore, the probability that no pair has a common birthday is

$$P_n = \left(1 - \frac{1}{N}\right)^{n^2/2} \approx \left(1 - \frac{1}{N}\right)^{N(n^2/2N)} \approx e^{-n^2/(2N)} \tag{4.10}$$

The last step is utilizing Euler's product limit formula for e^x:

$$e^x = \lim_{n\to\infty}(1+\frac{x}{n})^n$$

When $P_n = 1/2$, we can rearrange Equ. (4.10) to give

$$n \approx \sqrt{2\ln 2}\sqrt{N}$$

4.14.1 The Expected Waiting Time. Let E(N) be the expected waiting time until we get a double birthday. Let $q(N,n)$ be the probability that all n people chosen at random have different birthdays in a 'year' of N days. Therefore:

$$\begin{aligned} q(N,n) &= (1-\tfrac{1}{N})(1-\tfrac{2}{N})\cdots(1-\tfrac{n-1}{N}) \\ &= \tfrac{N}{N}\tfrac{N-1}{N}\tfrac{N-2}{N}\cdots\tfrac{N-n+1}{N} \\ &= \tfrac{1}{N^n}(N(N-1)(N-2)\cdots(N-n+1)) \\ &= \tfrac{1}{N^n}\tfrac{N!}{(N-n)!} \end{aligned}$$

The expected waiting time is the sum of these probabilities for all possibly values of n:

$$E(N) = \sum_{n=1}^{N} q(N,n).$$

This can also be phrased as the average number of people required before we find a pair with the same birthday. In other words, we are calculating the mean of the distribution, as opposed to the earlier median. This averaging problem is relevant to several hashing algorithms explained in Section 6.4, Vol. 3 of Knuth [**Knu97**].

The right hand side of the equation is very close to Srinivasa Ramanujan's Q formula:

$$Q(M) = \sum_{k=1}^{M} \frac{M!}{(M-k)!M^k}$$

Ramanujan found its asymptotic expansion to be:

$$Q(M) \sim \sqrt{\frac{\pi M}{2}} - \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2M}} - \frac{4}{135M} + \cdots$$

A derivation of this asymptote using only elementary calculus is given in Section 1.2.11.3, Vol. 1 of Knuth [**Knu97**]. It can be plugged into E(N), and simplified:

$$E(N) \sim \sqrt{\frac{\pi N}{2}} - \frac{1}{3}.$$

Listing 4.23 (birthexp.py) calculates $E(N)$ in two ways – by summing $q(N,n)$ probabilities from n=1 to N, and by calculating Ramanujan's asymptote.

```
N = int(input("days=? "))
prob = 1
tot = 0
for i in range(1, N+1):
  prob *= (1-((i-1)/N))
  tot += prob
print(f"Mean: {tot:.6f}, Q: {math.sqrt(math.pi*N/2)-1/3:.6f}")
```

Listing 4.23. Computing E(N) and $\sqrt{\pi N/2}$ - 1/3

The output when the number of days is 365.

```
> python birthexp.py
days=? 365
Mean: 23.616586, Q: 23.611200
```

The average number of people required to find a pair with the same birthday is close to the earlier calculation of the number giving a 50% chance for a pair.

4.14.2 An O($n^{1/4}$) Method of Factoring. Aside from the birthday problem's use in hashing, it's also employed in John Pollard's rho algorithm for fast integer factorization. The code's expected running time is proportional to the square root of the smallest prime factor of the composite being factorized. Finding this prime can be implemented in O($n^{1/2}$) running time, where n is the composite, so it is sometimes claimed that the algorithm is O($n^{1/4}$), although there's no rigorous proof of that speed.

As we'll see, the procedure uses a recurrence formula to generate random numbers, but its randomness is far from certain, so neither its running time nor its success are guaranteed. Nevertheless the procedure is highly effective in practice, and also requires a very small amount of computer memory.

We've already looked at factoring using trial division (see Listing 2.13.8). factor.py is a version of that code which returns its values using `yield` rather than by constructing a list. This means that factorsTime() can report the times taken to find each factor, which allow factorsY() to be more fairly compared to our rho() function (which only returns one factor).

Example runs:

```
> python factor.py
n=? 5166523324060302275610181
141325637  in 12.43 secs
36557580306963713  in 3.81 secs
  Total: 16.24 secs

> python factor.py
n=? 280846283204599997
265371653  in 21.86 secs
```

```
1058313049  in 0.00 secs
  Total: 21.86 secs
```

One way to double-check results for larger factorization is via the calculator at `https://www.numberempire.com/numberfactorizer.php`.

4.14.2.1 Applying the Birthday Problem to Factoring. If we are using trial division to search a number $N = p * q$ for its two prime factors, p and q, then the code may have to examine $\sqrt{N}$ numbers before finding one of them. But instead of examining one number at a time, we can pick k: $x_1, x_2, ...x_k$, and check whether any of the pair differences, $x_i - x_j$, divide N. This change parallels the difference between picking one person and asking if their birthday falls on a specific day and picking k people and asking if any two of them share a birthday.

We also know that for a k size of around $\sqrt{N}$, we have even odds of finding a prime. Very roughly speaking, we improve the chances from $1/\sqrt{N}$ with trial division to $1/\sqrt[4]{x}$ with the birthday technique. But there's a downside: if we select k people, then our code will have to perform roughly k^2 pairwise comparisons which is costly in time.

4.14.2.2 GCD to the Rescue. The solution to the pairwise problem is to compare the numbers in a way that increases the chance of a match. Instead of asking if $x_i - x_j$ divides N, we can ask if $\gcd(x_i - x_j, N) > 1$. In other words, we ask if $x_i - x_j$ and N have a non-trivial factor in common. This at once increases the number of chances for successes.

If we ask how many numbers divide N, there are only two: p and q. If we ask how many numbers have $\gcd(x, N) > 1$, we have quite a few more: $p, 2p, 3p.4p, 5p, ...(q-1)p$ and $q, 2q, 3q, ...(p-1)q$. A total of $p + q + 2$ numbers.

So the rho() function will do the following:

- Pick k numbers $x_1, ..., x_k$ at random from between 2 and $N - 1$.
- Test if $\gcd(x_i - x_j, N) > 1$. If it is greater, then $\gcd(x_i - x_j, N)$ is a factor of N (a multiple of p or q).

4.14.2.3 Space for k Numbers. This approach requires the generation of k random numbers all at once. For us to have a 50% chance of success the size of k should be around $N^{1/4}$, requiring a lot of storage. For example, if we're factoring $N \sim 10^{32}$, then we need to examine a range of numbers up to 10^{16} ($N^{1/2}$) which means generating $k = 10^8$ numbers ($N^{1/4}$).

This problem is addressed by *not* generating all k of them at once. The next best thing is to generate them randomly one by one and check consecutive numbers. We keep doing this for a while and *hopefully* we'll get lucky. (This is where the heuristic nature of the rho() function appears.)

We use a function $f()$ that we know generates pseudo-random numbers, such as $f(x) = x^2 + a \bmod N$. We start with a random seed, such as $x_1 = 2$, and keep calling $f(x)$ recursively to generate more x's: $x_2 = f(x_1)$, $x_3 = f(x_2)$, ... The general rule is $x_{n+1} = f(x_n)$.

This works well most of the time, but what about the output from the following $f()$:

```
2, 10, 16, 23, 29, 13, 16, 23, 29, 13, ...
```

In this case, the numbers quickly start cycling and will never find a factor. Indeed, cycling will be an issue with every PRNG eventually, and our rho() must deal with it.

One approach is to revisit Floyd's cycle detection algorithm from Sec. 3.14. The idea is to utilize two pointers A and B, but with pointer B moving twice as fast as A. They start off at the same place and when B overtakes A, we know that B has cycled around at least once.

4.14.2.4 Implementing rho(). Listing 4.24 (pollardRho.py) implements the features we've described: the birthday algorithm, the use of gcd() to increase the likelihood of success, the utilization of a PRNG to produce multiple random numbers, and Floyd's cycle detection.

```
def rho(n):
  if (n <= 1):
    return n
  if (n % 2 == 0):
    return 2
  x = random.randint(2, n-1)  # randoms in [2,n)
  y = x
  c = random.randint(1, n)
  d = 1
  # loop until the prime factor isn't obtained.
  # If n is prime, return n
  while (d == 1):
    x = (modPow(x,2,n) + c+n)%n  # slow move
    y = (modPow(y,2,n) + c+n)%n  # fast move
    y = (modPow(y,2,n) + c+n)%n
    d = math.gcd(abs(x-y), n)
    if (d == n):   # detected a cycle
      print("Retrying")
      return rho(n)
  return d  # 1 < d < n, so found a prime
```

Listing 4.24. Pollard's Rho function

The d variable returned by gcd() is used in three different ways. If $d = 1$ then the number tested against n is co-prime, so not divisible, which means that the search should continue. If $d = n$ then it means that $x - y$ is a multiple of n. This

implies that y (the fast pointer) is at the same position as x (the slow pointer) but after going round a cycle (of length n). We have entered a cycle, which means that we need to restart the function with a new x and c.

millerRabin.py contains a standard implementation of the function $(x^e) \mod m$.

If we test rho() on the composites passed to factor.py in the examples above, the speed-up is quite significant:

```
> python pollardRho.py
n=? 516652332406030227561018 1
One divisor for 516652332406030227561018 1 = 141325637
in 0.10 secs

> python pollardRho.py
n=? 280846283204599997
One divisor for 280846283204599997 = 265371653
in 0.12 secs
```

The original reference for this algorithm is Pollard, J. M. (1975), 'A Monte Carlo method for factorization', *BIT Numerical Mathematics*, 15 (3), pp.331-334. Improvements were later suggested in Brent, Richard P. (1980), 'An Improved Monte Carlo Factorization Algorithm', *BIT*, 20. pp.176-184.

A good textbook reference is Cormen *et al.* [**CLRS22**]. They discuss integer factorization and Pollard's rho algorithm in section 31.9, which sadly has been removed in the current 4th edition, but it can still be found on their website at `http://mitpress.mit.edu/algorithms`.

Exercise

Collect birthdays at random, until you obtain three birthdays for the same day. Repeat this experiment 10000 times and estimate the expected waiting time.

Additional Exercises for Chapters 3 to 4

(1) Write a program which performs n games of Craps and counts the number of wins. Here are the rules:

Roll two dice and total the points showing. If the total is 7 or 11 you win. If it is 2, 3, or 12 you lose. For any other total, call it the "point" and continue rolling the dice until either the total 7 comes up (a loss) or the point comes up again (a win).

Estimate the probability of a win by repeating the experiment 10000 times. Simple theory shows that the winning probability is $w = 244/495 = 0.492492....$ How many experiments do you need to perform to be 95% sure that $w < 0.5$?

(2) A fair coin is flipped until either the sequence 0011 or 1111 comes up. In the first case Abby wins and in the second case Carl wins. Write a simulation which estimates Abby's winning probability.

(3) *Symmetric Random Walk on a Line.* A particle starts at the origin and jumps one step to the left or to the right each second, with a probability of 0.5. Let D_n be the particle's distance from the origin after n steps. Find $E(D_n^2)$.

(4) A particle starts at the origin O and performs a random walk over the plane. At each step it goes from its current position (x, y) to one of its four neighbors $(x+1, y), (x-1, y), (x, y+1), (x, y-1)$ with probability 1/4. Let D_n be its distance from O after n steps. Find $E(D_n^2)$.

(5) A particle starts at O and takes unit steps. The direction of each step is uniformly distributed between 0 and 2π. Let D_n be its distance from O after n steps. Find $E(D_n^2)$ by simulation.

(6) *Neighbors in Lotto.* Six numbers are drawn at random without replacement from 1..49. Find by simulation the probability that the sample drawn contains at least two consecutive numbers, such as 18 and 19.

(7) *Random Walk in Space.* A particle starts at the origin O and takes unit steps from its current position (x, y, z) to one of its six neighbors $(x \pm 1, y, z)$, $(x, y \pm 1, z)$, $(x, y, z \pm 1)$, each with probability 1/6. Let D_n be its distance from O after n steps. Estimate $E(D_n^2)$ by simulation.

(8) *Random Walk in Space in Random Directions.* A particle in space starts at the origin O and takes unit steps. A unit sphere is projected around its current position P, a random point R is chosen on this sphere, and the next step goes from P to R. Let D_n be the particle's distance from O after n steps. Estimate $E(D_n^2)$ by simulation.

(9) *Maximum Test.* What is the probability that among any three successive base ten random digits the middle digit is strictly larger than any of its neighbors? Generate 10000 successive triples of base ten digits and find the proportion. Use:

a) Python's PRNG random.random();

b) The PRNG in Listing 3.22.

(10) *Poker Test.* Generate 1000 words made up of five random decimal digits, find the proportions of the seven types listed in Table 4.17, and compare them with the given probabilities. (More example poker hands are listed at `https://en.wikipedia.org/wiki/List_of_poker_hands`.)

type	example	probability
all distinct	30862	0.3024
one pair	32082	0.5040
two pairs	02772	0.1080
one triple	96066	0.0720
triple and pair	39399	0.0090
quadruple	80888	0.0045
quintuple	55555	0.0001

Table 4.17. Poker probabilities

(11) Re-read the rules of Craps in Ex. 1. Draw a graph for this game and calculate the winning probability exactly (up to 11 digits).

(12) *It all depends on the information.* If you toss a thumbtack, it will land pointing up (U) or down (D) as in Fig. 4.22.

Let $p = P(D)$ so that when we are given the outcome DDDUDUDDDDDU, we test H: $p = \frac{1}{2}$ versus A: $p > \frac{1}{2}$. But to compute the P-value we need more information about X (the number of D's):

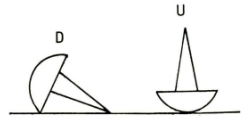


Figure 4.22. Tack positions (D=down; U=up)

a) It was decided in advance to make 12 tosses, so

$$P = P(9 \le X \le 12|H) = 2^{-12}\left(\binom{12}{3} + \binom{12}{2} + \binom{12}{1} + \binom{12}{0}\right) = \frac{299}{4096} \approx 7.3\%$$

b) In this case, the experiment was halted as soon as three U's appeared. Now the results $X = 9, 10, 11, 12, 13, ...$ are as extreme, or more extreme, than what we were given, so we must find

$$P = P(x \ge 9|H) = \sum_{x=9}^{\infty} p(x), \quad p(x) = \binom{x+2}{2} 2^{-x-2}\frac{1}{2} = \binom{x+2}{2} 2^{-x-3}.$$

(i) Find the P-value by simulation.

(ii) Find the exact P-value by avoiding an infinite sum.

(13) The sequence $u_1, ..., u_n$ is *unimodal* if there exists an s with

$$u_1 < u_2 < ... < u_s < u_{s+1} > ... > u_n \quad (s = 1 \text{ or } s = n \text{ are allowed})$$

a) How can we test for unimodality?

b) Let p_n be the probability that a sequence of n randomly chosen real numbers is unimodal. (p_n does not depend on the probability distribution as long as the chance of getting the same number twice is 0.) Find p_1 to p_9 by simulation and guess a formula for p_n.

c) Find the exact value for p_n.

(14) *Palindromes.* We are given a list A of integers, with a length $n < 1000$.

a) Find a maximal subsequence $A[i, ..., j]$ such that $A[i] = A[j], A[i+1] = A[j-1], ...$, and print its length. The algorithm should have complexity $O(n^2)$.

b) Let $A[i]$ = random.randint(0, 9). Find by simulation the expected length of the longest palindrome.

(15) *Palindromes again.* We generate a sequence consisting of digits from the alphabet {0, 1, 2} each occurring with equal probability. Estimate the probability P that the sequence starts with a palindrome of length ≥ 2.

Hints: a) Look at sequences of length n. They will start with a palindrome with probability $p_n < P$, which we can estimate by simulation. For large n, $p_n \approx P$.

b) For small n you can find p_n by brute force. This is handy for testing whether your code is correct.

(16) *A Total Beetle Tour.* A beetle starts at a vertex of a cube and performs a random walk along its edges until it has visited all of the vertices. What is the expected time for such a Total Beetle Tour? (The beetle covers one edge in one time unit.)

(17) Suppose the random walk in Ex. 16 only stops when all the edges have been traversed. Find the expected time by simulation (you should get ≈ 48.5).

Additional Exercises for Chapters 1 to 4

(1) *A Two-Pile Game.* We have two piles containing a and b chips. Every second a pile X is chosen at random and a chip moved from X to the other pile Y. Let T be the waiting time until one pile is empty. Find by simulation a formula for the expected time $E(T) = f(a, b)$.

(2) *A Three-Pile Game.* We have three piles containing a, b, and c chips. Every second a pile X is chosen at random, then another pile Y is chosen at random, and a chip is moved from X to Y. Let T be the waiting time until one pile is empty. Write a simulation for calculating $E(T) = f(a, b, c)$. By choosing small values for a, b, and c, guess the $f(a, b, c)$ formula, which was first revealed in 1992 through computer experimentation.

(3) Write a program which computes $f(m, n) = \sqrt{m\sqrt{(m+1)...\sqrt{n}}}$. Explore $f(2, n)$ for ever larger values of n and study its convergence speed. Also, find $f(n, 2n)$, $f(n, 3n)$, $f(n, \infty)$ for large n.

5

Combinatorial Algorithms

Combinatorial algorithms is a vast field, so vast in fact that even Donald Knuth is unsure how long Vol. 4 on Combinatorial Algorithms in *The Art of Computer Programming* (TAOCP) will be. As we write this in 2024, Volumes 4A, 4B, and 4C (very soon) are available, with more pre-fascicles, fascicles, and sub-volumes projected. A good collection of links to these can be found at the end of TAOCP's Wikipedia entry (`https://en.wikipedia.org/wiki/The_Art_of_Computer_Programming`).

Most beginner texts start with permutations, combinations, sorting, and partitioning, moving onto undirected and directed graphs to focus on algorithms related to search, shortest paths, spanning trees, and optimization problems.

We've decided to follow a similar approach, but rather than be too explicit about graphs, that topic is introduced through a series of recreational search puzzles (Sec. 5.5) and games (Sec. 5.8).

5.1 Sorting

Sorting is a major topic in computer science education since it provides a straightforward way to introduce concepts such as best, worst, and average case analysis, time–space tradeoffs, big-Oh notation, recursion, and data structures such as heaps and binary trees. The textbooks by Cormen *et al.* [**CLRS22**] and Sedgewick and Wayne [**SW11**] cover all of these topics excellently, as does Vol. 3 of Knuth's classic work [**Knu97**].

Sorting is an active research area – the widely used Timsort dates from 2002 [`https://en.wikipedia.org/wiki/Timsort`), and library sort was released

in 2006 (`https://en.wikipedia.org/wiki/Library_sort`). Timsort is derived from merge sort and insertion sort (described below), and was Python's standard sorting algorithm from version 2.3 to version 3.10 (until 2022), when it was dethroned by powersort (`https://www.wild-inter.net/posts/powersort-in-python-3.11`).

Fig. 5.1 summarizes the seven algorithms presented in this section.

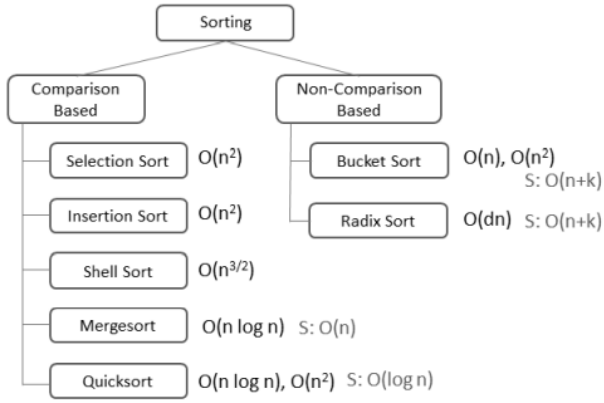


Figure 5.1. Seven sorting algorithms

Comparison sorts compare elements to determine which of them should be placed ahead of the other in the sorted structure. Usually a "less than or equal to" operation is utilized, but the only requirement is that the comparator forms a total preorder over the data, with:

- if $a \leq b$ and $b \leq c$ then $a \leq c$
- for all a and b, $a \leq b$ or $b \leq a$.

It's entirely possible that $a \leq b$ *and* $b \leq a$, and in a *stable* sort, the input order determines the sorted order in this case.

The fastest comparison sorts have an average-case lower bound of $O(n \log n)$ comparisons (n refers to the number of elements in the list). In this sense, mergesort and quicksort are optimal, but then other metrics, such as memory usage, become important.

Non-comparison sorts achieve $O(n)$ performance by using operations other than comparisons, allowing them to sidestep the $n \log n$ lower bound. However, they may lose out in other ways, such as by requiring more memory or by having to make assumptions about the type and range of the data.

Algorithms are usually analyzed in terms of their best, worst, and average running times, i.e. time complexity, but memory usage is also a common measure. Average and worst-case performance are most frequently used in algorithm

analysis, and we'll restrict ourselves to those two in what follows. Calculating an average running time can be quite difficult since we have to assume that the data was produced by a random process, typically one employing a uniform probability distribution.

Up until now, we've been using $O()$ (big-Oh notation) to describe the limiting behavior of functions, but there are several related notations, including $\Omega()$ and $\Theta()$ for other kinds of bounds on asymptotic growth rates. To summarize:

- $f(x) = O(g(x))$ means that the growth rate of the function $f(x)$ is asymptotically *less than or equal to* the growth rate of the function $g(x)$.
- $f(x) = \Omega(g(x))$ means that the growth rate of $f(x)$ is asymptotically *greater than or equal to* the growth rate of $g(x)$.
- $f(x) = \Theta(g(x))$ means that the growth rate of $f(x)$ is asymptotically *equal to* the growth rate of $g(x)$.

In short, O(n) represents an upper bound, $\Omega(n)$ a lower bound, and $\Theta(n)$ a tight (close) bound. We're going to keep on using $O()$ for simplicity's sake.

In Fig. 5.1, the big-Oh values preceded by 'S:' relate to space usage, and the algorithms without space big-Oh values should be understood to utilize constant amounts of memory (i.e. O(1)).

When there are two big-Oh values next to a sort name, the first is its average running time, the second its worst case time. The sorts with a single big-Oh have the same average and worst case running times.

An excellent way to gain an intuition about the seven algorithms in this section is to watch animations of their sorts at `https://m.youtube.com/watch?v=kPRA0W1kECg`. Altogether, fifteen sorting methods are covered by the video.

5.1.1 Selection Sort.

We proceed as follows: find the smallest element in the elems[0..n-1] list and exchange it with elems[0]. Then find the smallest element in elems[1..n-1] and exchange it with elems[1], progressing in this manner until the whole list is sorted. The code appears in Listing 5.1 (selectionsort.py).

```
def selectionSort(elems):
  n = len(elems)
  for idx in range(n):
    minIdx = idx
    for j in range(idx+1, n):
      # Find index of smallest number
      if elems[j] < elems[minIdx]:
        minIdx = j
    # swap elements
    elems[idx], elems[minIdx] = elems[minIdx], elems[idx]
```

Listing 5.1. Selection sort

The running time is $O(n^2)$ in both the worst and average cases. This can be confirmed by looking at the nested loops – the outer one iterates n times, while the inner one executes $n-1$ times on its first loop, $n-2$ times on its second, and so on. The total number of iteration is $(n-1)+(n-2)+...+1$, which is $\frac{(n-1)n}{2}$ or $O(n^2)$.

5.1.2 Insertion Sort. Insertion sort is almost as simple as selection sort, and may be familiar to card players as the way they sort their hands: look at the cards one at a time, inserting each in its proper place among those already sorted. The code appears in Listing 5.2 (insertionsort.py).

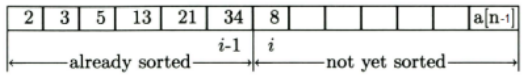


Figure 5.2. Partially sorted

```
def insertionSort(elems):
  n = len(elems)
  if n > 1:
    for i in range(1, n):
      val = elems[i]
      j = i-1
      while j >= 0 and val < elems[j]: # Move elems > val
        elems[j+1] = elems[j]  # Shift to right
        j -= 1
      elems[j+1] = val
```

Listing 5.2. Insertion sort

In common with selection sort, insertion sort's worst and average case running times are both $O(n^2)$. The n elements of the list can be viewed as $n(n-1)/2$ pairs. Each pair is out of order with a probability of 0.5, so the expected number of out of order pairs is $n(n-1)/4$. To bring them into order requires $n(n-1)/4$ exchanges, or $O(n^2)$ running time.

5.1.3 Shell Sort. Since insertion sort's running time is $O(n^2)$, it will take less time to sort h lists of n/h elements than to sort a single list of n elements. Shell sort takes advantage of this behavior.

For a given value of h it separately sorts sublists consisting of elements which are multiples of h apart. For large values of h, this will move elements long distances and, in general, rearrange the list to be much closer to its natural order. The algorithm incrementally switches to smaller h-values until finally $h = 1$, which is just insertion sort.

Some sequences of h-values seem to sort faster than others. Knuth recommends ..., 1093, 364, 121, 40, 13, 4, 1 [**Knu97**], with each h obtained from the next by the substitution $h_{i+1} = 3h_i + 1$. Rather foolishly, we've ignored this advice in Listing 5.3 (shellsort.py), and gone with the easier to code halving of h, starting at $n \mathbin{//} 2$. An interesting exercise is to measure how much speed gain comes from the use of $3h + 1$ instead.

```
def shellSort(elems):
  n = len(elems)
  h = n//2
  while h > 0:
    for i in range(h, n):
      val = elems[i]
      j = i
      while j >= h and val < elems[j-h]:
        elems[j] = elems[j-h]
        j -= h
      elems[j] = val
    h = h//2
```

Listing 5.3. Shell sort

Shell sort's worst case and average case time complexity depend very much on the h-sequence, so it's hard to calculate an accurate O() value. Pratt (1979) has shown that the worst case is proportional to $O(n^{3/2})$ for many sequences [**SW11**], but some researchers believe that shell sort may be able to run in an average time of $O(n \log n)$ for h-sequences that grow in proportion to the logarithm of the list size. In Fig. 5.1, we've conservatively assigned shell sort a running time of $O(n^{3/2})$, but it's a lot faster than that in practice, as indicated by the running time graph in Fig. 5.5b.

5.1.4 Mergesort. Mergesort is a divide-and-conquer algorithm invented by John von Neumann in 1945. Assuming there's an n-element list, elems, divide it into two sublists, each of half the size. To do so, compute the midpoint mid, and divide elems[left..right] into sublists elems[left..mid] and elems[(mid+1)..right]. Sort each sublist recursively using mergesort, only stopping when it reaches a sublist with 0 or 1 element.

The sublists are recombined by *merging* as the recursion levels created during the 'conquer' stage return. The merging can assume that adjacent sublists elems[left..mid] and elems[(mid +1)..right] are sorted so it only needs to compare the first elements in each sublist to decide which one is smaller and should be moved to the larger sublist elems[left..right]. It repeats this comparison with the next first elements of the sublists, continuing until one of the lists is empty, which causes the other to be copied over in its entirety. Merging requires the creation of a temporary list so that the sublists aren't overwritten while the merged

larger one is being built. Merging takes time roughly proportional to the length of the full list, so is $O(n)$.

The top-level code is given in Listing 5.4 (the complete program is in mergeSort.py). The recursive division of the list is managed by msort(), and merging the resulting sublists is handled by merge().

```
def mergeSort(elems):
  msort(elems, 0, len(elems)-1)

def msort(elems, left, right):
  if left < right:
    mid = left+(right-left)//2
    # Sort halves
    msort(elems, left, mid)
    msort(elems, mid+1, right)
    merge(elems, left, mid, right)
```

Listing 5.4. Mergesort

5.1.4.1 Mergesort Big-Oh Analysis. Let $T(n)$ be the maximum number of comparisons used while merge-sorting a list of n numbers. To simplify matters, let's assume that n is a power of 2, which ensures that the input can be divided in half at every stage of the recursion. If there's only one number in the list, then no comparisons are required, so $T(1) = 0$. More generally, the $T(n)$ time includes the time sorting the first half (at most $T(n/2)$), sorting the second half (also at most $T(n/2)$), and merging the two sorted halves (at most n). Therefore, the number of comparisons needed to mergesort n items becomes the recurrence:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 2T(n/2) + n \quad \text{for } n \geq 2 \end{aligned}$$

The easiest way to obtain a closed form for $T()$ is to expand the recurrence relation until we (hopefully) see a pattern:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/4) + n/2) + n &&= 4T(n/4) + 2n \\ &= 4(2T(n/8) + n/4) + 2n &&= 8T(n/8) + 3n \\ &= 8(2T(n/16) + n/8) + 3n &&= 16T(n/16) + 4n \end{aligned}$$

A pattern emerges when we notice that $16 = 2^4$. Let's generalize the division on the right-hand side from 16 to 2^k, and also rewrite the $4n$ term as kn:

$$T(n) = 2^k T(n/(2^k)) + kn, \text{ for } k = 1, 2, ...$$

As k increases, the $n/2^k$ argument on the right-hand side keeps halving, reaching 1 when $n = 2^k$, or equivalently when $\log_2 n = k$. Substitute this value of k into

the equation:

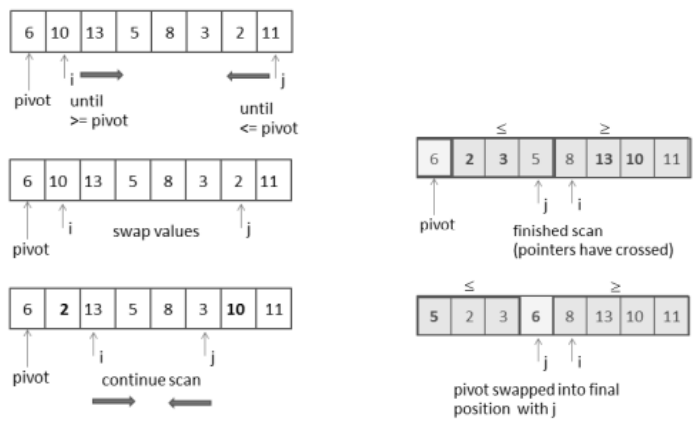
$$\begin{aligned} T(n) &= 2^k T(n/(2^k)) + kn \\ &= nT(n/n) + \log_2 n \cdot n \\ &= nT(1) + n \log_2 n \\ &= n \log_2 n \end{aligned}$$

Thus, $T(n)$ is $O(n \log_2 n)$, but a drawback is that it requires memory of size $O(n)$ for the temporary lists required by merge().

5.1.5 Quicksort. Quicksort, due to C.A.R. Hoare (1960), is a popular, fast, general sorting method. It employs divide-and-conquer like mergesort, but without the extra memory overheads.

The algorithm starts by partitioning a list into two, then sorts them by recursively calling itself on the two parts. The partitioning of list elems[left..right] moves all the elements smaller than a chosen pivot value to the left, and all the bigger elements to the right. The pivot can be any element in the list, so we'll choose the leftmost value for simplicity. However, quicksort is only 'quick' if the pivot divides the list into two roughly equal sized halves.

The movement of the smaller elements to the left and bigger elements to the right is done by scanning the list from the left until an element bigger (or equal) to the pivot is found, *and* from the right until an element smaller (or equal) is found. The two elements are swapped, and scanning and swapping continues until the two traversals meet. These steps are illustrated in Fig. 5.3a.



(a) Part 1 (b) Part 2

Figure 5.3. Partitioning a list

When the i and j pointers in Fig. 5.3a cross, the list can be considered to have three sections (which are shown in various shades of gray in Fig. 5.3b). The pivot is at the start of the list, the values $\leq$ the pivot are stored up to the j position, and the values $\geq$ the pivot complete the list.

The last step swaps the pivot value with the j value, causing it to finish in its correct sorted position. Note that the values on either side of the pivot are *not* yet sorted, but they are $\leq$ and $\geq$ than the pivot.

The partition() code is in quicksort.py.

Quicksort has and average case running time of $O(n \log n)$. Unfortunately, its worst case is $O(n^2)$, which occurs when the pivot divides the list into two very unequal parts, but that occurs rarely in practice.

Python's list comprehensions allows partitioning to be implemented very simply in Listing 5.5 (quicksort.py). However, this version is a little less time efficient than the original since two passes are made over the list to create the sublists. It's also less space efficient since the comprehensions create new sublists rather than carry out in-place repositioning.

```
def quickSortL(elems):
  if len(elems) <= 1:
    return elems
  else:
    pivot = elems[0]  # use left-most
    left = [x for x in elems[1:] if x < pivot]
    right = [x for x in elems[1:] if x >= pivot]
    return quickSortL(left) + [pivot] + quickSortL(right)
```

Listing 5.5. Slower but elegant quicksort

This version also needs to assign its result back to elems in order to change the original:

```
elems = quickSortL(elems)
```

5.1.5.1 Quicksort Average-Case Analysis. We assume that quicksort is sorting a list S of n elements. The partitioning function P, will return an index that partitions the list into S_L (left) and S_R (right). We have $P(n) = O(n)$, because this work can be done in one pass through S. Hence the running time, $T(n)$ is given by:

$$T(n) = T(i) + T(n - i - 1) + n, \text{ for } n > 1, \text{ and } T(0) = T(1) = 1, \tag{5.1}$$

where $i = |S_L|$ is the size of the left sublist.

We'll assume that every possible size of S_L (and consequently S_R) is equally likely (this is a probabilistic argument, which is needed when dealing with average case analyses). Possible sizes are $0, 1, ..., n - 1$, each having a probability $1/n$.

Hence, the average value of $T(n)$ in Equ. (5.1) is given by,

$$T(n) = \frac{1}{n}\left[\sum_{i=0}^{n-1} T(i) + T(n-i-1)\right] + n \tag{5.2}$$

Since $\sum_{i=0}^{n-1} T(i) = \sum_{i=0}^{n-1} T(n-i-1)$, $T(n)$ can be written as,

$$T(n) = \frac{2}{n}\left[\sum_{i=0}^{n-1} T(i)\right] + n \tag{5.3}$$

Multiplying by n, we get

$$nT(n) = 2\left[\sum_{i=0}^{n-1} T(i)\right] + n^2 \tag{5.4}$$

Replacing n by $n-1$:

$$(n-1)T(n-1) = 2\left[\sum_{i=0}^{n-2} T(i)\right] + (n-1)^2 \tag{5.5}$$

Subtracting Equ. (5.5) from Equ. (5.4), we obtain,

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1$$

Rearranging and simplifying, produces

$$nT(n) = (n+1)T(n-1) + 2n$$

Dividing both sides by $n(n+1)$:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1} \tag{5.6}$$

We can now repeatedly expand the $T()$ expression on the right-hand side by applying Equ. (5.6) multiple times:

$$\begin{aligned}
\tfrac{T(n)}{n+1} &= \tfrac{T(n-2)}{n-1} + \tfrac{2}{n} + \tfrac{2}{n+1} \\
&= \tfrac{T(n-3)}{n-2} + \tfrac{2}{n-1} + \tfrac{2}{n} + \tfrac{2}{n+1} \\
&= \cdots \\
&= \tfrac{T(1)}{2} + 2\textstyle\sum_{i=3}^{n+1} \tfrac{1}{i} \\
&\approx \tfrac{1}{2} + 2[\ln(n+1) + \gamma - \tfrac{3}{2}]
\end{aligned}$$

In the last line, we've replaced the harmonic series summation by its approximation using Euler's constant. Therefore,

$$\frac{T(n)}{n+1} = O(\log n)$$

and hence,

$$T(n) = O(n \log n)$$

Exercises

(1) Suppose $n = 2^m$ equal numbers are sorted by quicksort. Show that $0.5n \log_2 n$ interchanges will be made.

(2) Write a function that sorts a list that is known to have at most three different values. (Edsgar Dijkstra called this the Dutch-national-flag problem because the result is three "stripes" of values like the three stripes in the Dutch flag.)

Hint: First partition the list into two parts with all elements having the smallest value in the first part and all other elements in the second part, then partition the second part.

5.1.6 Bucket Sort. If we have a list of data that is known to reside in a certain range; for example, floats between 0 and 1, or integers between 1900 and 2100, then we can employ a bucket sort to obtain close to linear running times for the sorting. This is possible because we don't need to do comparisons on the data (or at least very few comparisons), instead relying on the separation of the data into buckets. Each bucket will represent a subrange of the data, hopefully small enough so that only one data item will be assigned to each bucket. Sorting then becomes a matter of scanning the filled buckets in order.

Listing 5.6 (bucketsort2.py) can sort lists that contain floats between 0 and 1. If the list has n elements, then n buckets are created in the hope that each data element will be assigned to its own bucket. This best case may not occur, so after the data has been distributed, insertion sort is used to sort each bucket. Since there will probably only be a few elements per bucket this step will be very fast, with near linear running time. The final step is to combine the buckets into a single list.

```
def bucketSort(elems):
  n = len(elems)
  buckets = [[] for _ in range(n)]
  # Put elements in different buckets
  for num in elems:
    bi = int(n * num)
    buckets[bi].append(num)
  # Sort individual buckets using insertion sort
  for bucket in buckets:
    insertionSort(bucket)
  # Concatenate all buckets into elems[]
  index = 0
  for bucket in buckets:
    for num in bucket:
      elems[index] = num
      index += 1
```

Listing 5.6. Bucket sort

The only tricky aspect of Listing 5.6 is how a data item is mapped to a particular bucket. This will vary depending on the type of data and the number of buckets. In this case, we're assuming that the data is uniformly distributed, so its sufficient to multiply each value by n then cast it to an integer to get a bucket index. The underlying aim is to avoid placing a large amount of the data in a single bucket.

The worst-case running time occurs when all the elements are placed in a single bucket. The overall performance will then be dominated by the algorithm used to sort each bucket – $O(n^2)$ for insertion sort. The average case however will be as sketched above and so the three loops inside bucketSort() will each be linear, resulting in a running time of $O(n)$.

The speed-up by not using comparison tests is not cost free. In the case of bucket sort, extra memory is needed for the buckets. Typically, if the data has n elements, and there are k buckets, then the memory usage will be $O(n + k)$.

5.1.7 Radix Sort. Radix sort dates from the 1890 US census, developed by Herman Hollerith for his punched card-sorting machines which tallied the results. A sorter could be mechanically 'programmed' to examine a given column on each card in a deck and drop the card into a bin depending on which place had been punched. An operator would gather the cards bin by bin, and order them so that those with the first place punched were on top of the cards with the second place punched, and so on.

Hollerith's original (inefficient) idea was to sort by starting with the most significant digit, then move right. This generates lots of intermediate piles of cards (i.e. extra temporary memory) to keep track of the calculations. A surprisingly simple fix is to sort on the least-significant digit first, and move to the left. The pseudo-code is:

```
def radixSort(elems, digits):
  for i in range(digits):
    stableSort(elems) on digit i
```

The use of a 'stable' sort is necessary so that cards with the same digit in a particular column stay in their original relative order. If this ordering isn't maintained, then a sort of a later digit may position the cards incorrectly.

Fig. 5.4 shows the operation of radix sort on seven 3-digit positive integers. The right-most 1's digits are sorted first, then the 10's digits, and so on. After three iterations (the number of digits in each number), the list is sorted.

radixSort.py uses a variant of the bucket sort from the last section. The main change is to hardwire the use of ten buckets representing the ten possible values

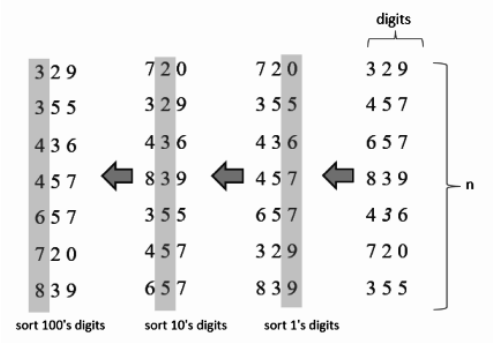


Figure 5.4. Radix sort of seven 3-digit numbers

for a decimal digit. This bucket sort version is stable because it no longer calls insertion sort.

Big-Oh calculations for radix sort typically involve three variables: n (the size of the list), k (the range of values for a single digit), and d (the maximum number of digits used by a value in the data set). In Fig. 5.4, $n = 7$, $k = 10$, and $d = 3$.

Each call to bucket sort will sort n numbers based on looking at a particular column of digits, so its running time will be $O(n)$. This work will be repeated by the outer loop which examines each digit position 0, 1, ... $d - 1$. Therefore its total running time is $O(dn)$. Since d is often a constant, we can simplify this to $O(n)$.

One downside of radix sort, as illustrated in the code, is the need to calculate the number of digits d, which requires a linear scan of the list before sorting can begin. Another drawback is the assumption that the data consists of positive base-10 integers. Radix sort can be used on other types of data, such as floats and binary data, but different coding tricks are needed, based on the range of values.

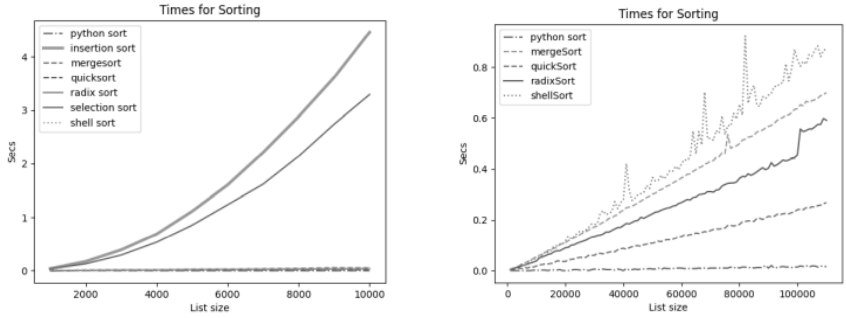
A less obvious weakness, compared to quicksort for example, is poor *locality of reference* which means that the algorithm may need to access values widely spread across the list when comparing digits. If the list is very big, such widely dispersed data may not be cached in RAM, requiring the OS to do a slow access to the hard disk to get it.

5.1.8 Which is Fastest? sortTimings.py imports six of the seven sorting functions previously described, excluding Listing 5.6 since it was designed to work with floats between 0 and 1. However, bucket sort *is* represented here since radix sort uses a version suitable for positive integers. `sortTimings.py` generates random integer lists of various sizes between 1000 and 10000 elements, and calls timer() in sortTimings.py to obtain an average time for each sort function. The timing data for each sort are plotted in Fig. 5.5a.

The incredibly fast 'python sort' is Python's built-in powersort, which is utilized by timer() calling:

```
def pSort(elems):
  elems.sort()
```

There are three groups of timings in Fig. 5.5a – sorts which have quadratic running time (selection sort, insertion sort), those with $n \log n$ running times (shell sort, mergesort, quicksort, Python's sort), and linear running time (radix sort).



(a) All the sorts (b) Faster sorts

Figure 5.5. Sort timings

The calls to the two quadratic sorts were commented out before generating Fig. 5.5b so that the slight differences between the others are easier to see. We also increased the random list sizes to 110000 elements.

Quicksort lives up to its name, beating even radix sort. Also, the timing shown in Fig. 5.5b is based on calling quicksortL() (Listing 5.5), which uses list comprehensions to partition the lists.

It may seem surprising that an $n \log n$ algorithm can 'beat' a linear running time, but this illustrates a problem with big-Oh calculations: constants are elided which may in practice be a significant component of the run time. Recall that big-Oh is only an upper bound, with no guarantee that it's particularly close to the actual running time. For a tighter bound we need $\Theta()$ which usually requires a much deeper analysis.

The erratic behavior of shell sort is a consequence of its susceptibility to h-interval size. Our code simply halves the h-interval on each iteration, and (as mentioned before) it's likely that better speeds could be gained by using Knuth's $3h + 1$ equation.

The final graph (Fig. 5.6) only measures Python's sort, with lists sizes between 1 and 11 million.

The documentation for powersort (https://github.com/python/cpython/blob/main/Objects/listsort.txt) refers to its "supernatural performance" on many kinds of partially ordered arrays (less than $\log(n!)$ comparisons needed, and as few as $n-1$), and this is confirmed by our plots.

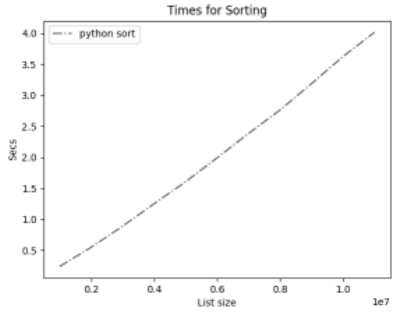


Figure 5.6. Timings for Python's sort()

The C source code for powersort is available at https://github.com/python/cpython/blob/main/Objects/listobject.c, starting with function islt() and proceeding for *many* lines. Sebastian Wild's blog post https://www.wild-inter.net/publications/munro-wild-2018 includes a link to an academic paper by Munro and Wild on the algorithm.

5.2 The kth Smallest Element of a List

We want to find the k-th smallest element of an n-element list. We could utilize selection sort's approach (Sec. 5.1.1) and find the first smallest, second smallest, ..., until we reach the k-th smallest, which is fine if k is small. We could also sort the entire list and then print elems[k].

Generally, we can reach our goal faster if we use quicksort's partition function (Sec. 5.1.5). It rearranges the list's values so that all of those less (or equal) are placed before the pivot, and those that are greater (or equal) are positioned after it. It finishes by returning the pivot's index, which is its final position when the list is fully sorted.

We can use this returned value to guide a search over the list. If partition() returns a pivot index greater than k then the element we are looking for must be in the left half of the list. Conversely, if the pivot index is less than k then we should examine the right half. We stop when the pivot index equals k. This is implemented in Listing 5.7 (kselect.py).

```
def kselectP(elems, k):
  left = 0
  right = len(elems)-1
  while left <= right:
    pivot = partition(elems, left, right)
    if pivot == k:
      return elems[pivot]
    elif pivot < k:
      left = pivot+1   # focus on right half
    else:  # pivot > k
```

```
      right = pivot-1  # focus on left half
  return -1
```

Listing 5.7. k-th selection using partition()

In the worst case, each call to partition() will half the sequence range being searched, so the running time $\approx n + n/2 + n/4 + ...1 \approx 2n$, or $O(n)$.

Currently, partition() simply chooses the left-most element in the subsequence. Listing 5.7 can be made faster if we can make the choice of pivot more relevant to the selection problem. Listing 5.8 (kselect.py) implements this idea by replacing the call to partition() in Listing 5.7 by partition()'s code. The pivot choice (`pivot = elems[left]`) can now be modified.

```
def kselect(elems, k):
  left = 0; right = len(elems)-1
  while left < right:
    pivot = elems[k] # use kth element
    i = left; j = right
    while (i <= k) and (k <= j):
      while elems[i] < pivot:
        i += 1   # move right
      while pivot < elems[j]:
        j -= 1   # move left
      elems[i], elems[j] = elems[j], elems[i]
      i += 1; j -= 1
    if k < i:
      right = j  # focus on left half
    if j < k:
      left = i   # focus on right half
  return elems[k]
```

Listing 5.8. k-th selection with kth element as pivot

The key line change (on line 4) is:

```
pivot = elems[k]
```

If the data is partially sorted then this pivot will probably be closer to the actual k-th smallest element, and so the algorithm will find it sooner, although in the worst case it will still take $O(n)$ time.

Chapter 15 of Bentley [**Ben88**] considers the k-th smallest problem in detail, including some revealing visualizations of how various different algorithms perform.

5.3 Binary Search

Listing 5.9 (binSearch2.py) is a recursive function that employs binary search on a sorted list to find the index position of a specified v data item. If the data cannot be found then the function returns -1.

```
def binSearch(elems, v):
  return binSrch(elems, v, 0, len(
      elems)-1)

def binSrch(elems,v,left,right):
  if left > right:
    return -1
  mid = (left + right)//2
  if v < elems[mid]:
    return binSrch(elems, v, left,
        mid-1)
  elif v == elems[mid]:
    return mid
  else:
    return binSrch(elems, v, mid
        +1, right)
```

Listing 5.9. Recursive binary search

Two examples:

```
>>> from binSearch2 import *
>>> elems = [2, 4, 5, 7, 8, 9, 12,
14, 17, 19, 22, 25, 27, 28, 33, 37]

>>> binSearch(elems, 22)
10
>>> binSearch(elems, 44)
-1
```

Fig. 5.7 shows the four stages in the search for 22 in the list.

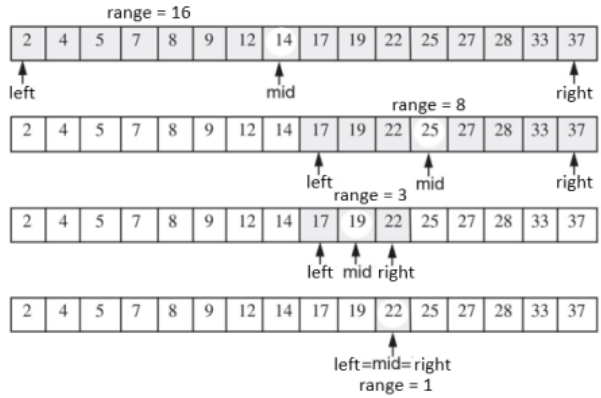


Figure 5.7. Binary search for 22

Binary search is sometimes known as 'binary chop' since it finds an element by *roughly* halving the search interval at each step. It compares the target value to the middle element of the list. If they are not equal, the half in which the target cannot lie is ignored and the search continues with the other half, comparing its middle element to the target, and repeating this until the value is found. If the search ends with the searchable half empty, then the target is not in the list.

Binary search runs in logarithmic time in the worst and average cases, using $O(\log_2 n)$ comparisons, where n is the number of elements. This means that it's faster than linear search (except for small lists), but the list must be sorted.

The worst case running time is fairly easy to determine. If T(n) represent the running time, then we can characterize it as:

$$T(n) \leq \begin{cases} a & \text{if } n < 2 \\ T(n/2) + b & \text{otherwise} \end{cases}$$

where a and b are constants (usually written as $O(1)$).

If we assume that n starts as a power of 2, 2^k for example, then we can expand $T(n)$ to look for a pattern:

$$\begin{aligned} T(2^k) &= b + T(2^{k-1}) \\ &= b + b + T(2^{k-2}) \\ &= b + b + b + T(2^{k-3}) \\ &= \cdots \\ &= kb + T(2^{k-k}) \\ &= bk + a \end{aligned}$$

We assumed that $2^k = n$, so $k = \log_2 n$. This means that:

$$T(n) = b \times \log_2 n + a$$

If we replace constants by Big-Oh notation:

$$\begin{aligned} T(n) &= O(\log_2 n) + O(1) \\ &= O(\log_2 n) \end{aligned}$$

5.3.1 Average-case Analysis for Binary Search. It helps if we adjust our view of the list, thinking of it as a *binary tree*, as in Fig. 5.8. We've also deleted the last node (37) to make the list 15 nodes long (i.e. $2^4 - 1$), so the tree is balanced. In general we'll assume that $n = 2^k - 1$.

The shaded nodes in Fig. 5.8 show the sequence of comparisons that were made by the binary search, matching those in Fig. 5.7.

One comparison was done to find the element that is at the root of the tree. Two comparisons were done to get to level 2, three comparisons for level 3, and four comparisons got us to level 4 and the answer. In general, we will need i comparisons to get to level i.

Another point to note is the number of nodes at each level: 1 node at level 1, 2 nodes at level 2, 4 nodes at level 3, and 8 nodes at level 4. In general there will be 2^{i-1} nodes at level i.

We can express the total number of comparisons made in every possible case by summing the product of the number of nodes on each level and the number

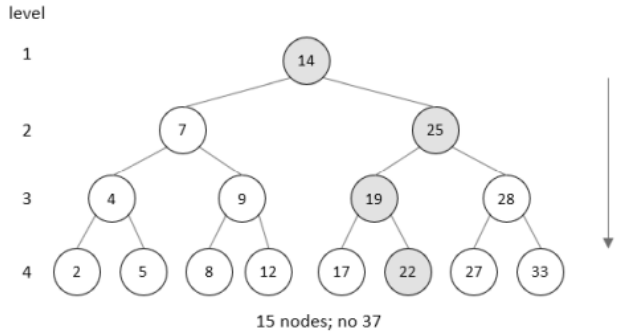


Figure 5.8. Binary search over a binary tree

of comparisons for that level. This gives an average case analysis of

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{i=1}^{k} i2^{i-1} \quad \text{for } n = 2^k - 1 \\ &= \frac{1}{n}\cdot\frac{1}{2}\sum_{i=1}^{k} i2^{i} \end{aligned}$$

The $1/n$ assumes that every possible case is equally likely, and introduces a probabilistic component into the calculation.

For the next step, we'll use a handy summation (which can be proved by induction, or by differentiation):

$$\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$$

Continuing:

$$\begin{aligned} T(n) &= \frac{1}{n}\cdot\frac{1}{2}[(k-1)2^{k+1} + 2] \\ &= \frac{1}{n}[(k-1)2^k + 1] \\ &= \frac{1}{n}[k2^k - 2^k + 1] = \frac{[k2^k - (2^k - 1])}{n} \\ &= \frac{[k2^k - n]}{n} = \frac{k2^k}{n} - 1 \end{aligned}$$

Since $n = 2^k - 1$, $2^k = n + 1$, so:

$$T(n) = \frac{k(n+1)}{n} - 1 = \frac{kn + k}{n} - 1$$

As n gets larger, k/n approaches 0, giving

$$\begin{aligned} T(n) &\approx k-1 \\ &= \log_2(n+1)-1 \\ &= O(\log_2 n) \end{aligned}$$

This is unfortunately not quite enough since we've only consider the n cases where a value is found; we should also include the $n+1$ cases when a value is *not* found. The number is $n+1$ since the target can be smaller than the first element, larger than the first element but smaller than the second, larger than the second but smaller than the third, and so on, through to the possibility that the target is larger than the last n-th element. In each of these cases, it takes i comparisons to decide that the target is not in the list.

We must redo our calculation with $2n+1$ possibilities. This gives

$$T(n) = \frac{1}{2n+1}\left[\left(\sum_{i=1}^{k} i2^{i-1}\right) + (n+1)k\right] \quad \text{for } n = 2^k - 1$$

A similar series of substitutions eventually leads to:

$$\begin{aligned} T(n) &\approx k-\tfrac{1}{2} \\ &= \log_2(n+1)-\tfrac{1}{2} \\ &= O(\log_2 n) \end{aligned}$$

This is a little larger than the average case when the target is in the list, but still logarithmic.

5.3.2 An Iterative Binary Search. The recursion in Listing 5.9 occurs at the end of the algorithm. This makes it fairly easy to translate the function into the iterative version in Listing 5.10 (binSearch2.py).

```
def binSearchIter(elems, v):
  left = 0
  right = len(elems) - 1
  while left <= right:
    mid = (left + right)//2
    if v < elems[mid]:
      right = mid-1
    elif v == elems[mid]:
      return mid
    else:
      left = mid+1
  return -1
```

Listing 5.10. Iterative binary search

Two examples:

```
>>> from binSearch2 import *
>>> elems = [2, 4, 5, 7, 8, 9, 12,
14, 17, 19, 22, 25, 27, 28, 33, 37]
>>> binSearchIter(elems, 22)
10
>>> binSearchIter(elems, 44)
-1
```

Exercises

(1) Modify binSearch() so that it also finds the longest:
a) strictly increasing interval; b) interval of constancy.

If there is more than one such interval it should give the leftmost such interval (and the number of such intervals).

(2) Modify binSearch() so that it starts searching for a given v at index position i = int(t*i), where $t = (\sqrt{5}+1)/2$.

(3) A list is *bitonic* if it consists of an increasing sequence of values followed immediately by a decreasing sequence of values. Given a bitonic list, design a logarithmic algorithm to find the index of a maximum value. An efficient solution uses a modified binary search.

5.4 Binary Questions

Carl thinks of a number x ∈ [1..MAX] which Python will guess with a minimum of questions (see binguess.py) to which Carl need only answer 'yes' or 'no'.

Since MAX = 100, it is hardly surprising that Python will always get to the answer after six questions (MAX $< 2^7$).

Exercises

(1) If MAX $= 2^{10}$ and Carl thinks of the number $x = 777$ then the sequence of questions posed by binguess.py will be answered as 0011110111 in terms of 'no' = 0 and 'yes' = 1. How is this sequence related to the binary expansion of 777? Generally, what is the relationship between a number x that Carl thinks up and the (0, 1)-sequence of answers which leads to Python finding x?

(2) *Random Guessing*. Carl thinks of a number G ∈ 1..n. Abby tries to guess G as follows: she chooses a random subset R ⊂ 1..n and Carl replies 'yes' or 'no' if G ∈ R or G ∉ R, respectively.

Let E(n) be the expected number of guesses to find G. Estimate E(n) by simulation for different values of n. You can check your implementation against the following: $E(2) = 2$, $E(3) = 8/3$, $E(4) = 22/7$, $E(5) = 368/105$, $E(16) = 5.287$. The most surprising thing about this approach is how surprisingly rarely you lose compared to deterministic guessing.

The coding is easier if you play the game with a coin several times first, to get a 'feel' for how it works.

How is this problem related to Ex. 7 of Sec. 4.13?

5.5 Recreational Search

Many recreational math puzzles can be treated as search problems: we start from an initial state, use the puzzle conditions to guide a traversal of the problem space towards a suitable goal, with the resulting path becoming the solution. The search should produce an answer within a reasonable amount of time, and the ever-increasing speed of hardware, combined with the small size of most puzzles, means that the relatively simple breadth-first search (BFS) and depth-first search (DFS) algorithms are viable choices.

BFS and DFS are 'uninformed' algorithms in that they don't utilize any problem-specific information to guide their search. The early years of AI research [**RND10**] saw these techniques augmented with 'heuristics' to improve their results, and we'll briefly look at one of the most popular: the A* (A-Star) algorithm.

One direction to take with these problems is towards graph theory, converting the search spaces into graphs in order to access a considerable collection of algorithms for their manipulation. We've decided *not* to do that, but any good algorithms textbook spends a considerably amount of time on that topic (e.g. Cormen *et al.* [**CLRS22**], Sedgewick and Wayne [**SW11**]).

The puzzles considered here are classics: the water jugs problem, the missionaries and cannibals, maze search, the 8-puzzle (a more tractable version of the 15-puzzle), and eight queens. A good source for details on these, and many others, is the magnificent text by Rouse Ball and Coxeter [**BC87**].

5.5.1 The Water Jugs Problem. Two unmarked jugs can hold m and n liters of water respectively ($0 < m < n$). The permitted operations are: 1) empty a jug, 2) fill it to the top, and 3) pour water from one jug into the other until the jug is empty or the other is full. The goal is to measure out d liters of water ($d < n$) in a minimum number of operations.

The problem can be modeled as a Diophantine equation $mx + ny = d$ which is solvable if and only if $gcd(m, n)$ divides d, which means that the solution x, y can be obtained using the extended GCD algorithm. However, we'll utilize BFS without relying on these attributes. We'll also set $m = 3$, $n = 5$, and $d = 4$ in honor of this problem's appearance in *Die Hard with a Vengeance* (1995). Of course, the puzzle is a bit older, occurring in Bachet's 17th-century textbook, *Problèmes plaisans et délectables qui se font par les nombres*. It's also closely related to the *three* jugs problem devised by the sixteenth-century Italian mathematician, Niccolo Fontana, better known by his nickname, Tartaglia, "the Stutterer".

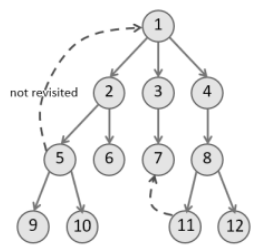


Figure 5.9. Nodes examined in BFS order

BFS treats the search space as a tree (see Fig. 5.9) with the initial state as the root (the node at the *top*). The algorithm explores all the nodes at the current depth before moving onto those at the next level. This means that when BFS finds the goal state that the path from the root will be the shortest.

Most search spaces aren't trees but graphs with cycles (e.g. it's perfectly acceptable to repeatedly fill and empty a jug forever). This is avoided by ignoring nodes that have already been visited, which is why the edges from nodes 5 to 1 and 11 to 7 aren't part of the search.

A simple BFS implementation stores each level in a queue whose size is proportional to b^d, where b is a node's branching factor (its number of 'children'), and d is the tree depth.

The bfs() function in Listing 5.11 is located in searchUtils.py.

```
def bfs(startState):
  queue = [ node(startState, None)]   # no parent
  visited = []
  while queue != []:
    currNode = queue.pop(0)   # remove from start
    visited.append(currNode.state)
    if isGoal(currNode.state):
      return currNode.getPath(), len(visited)
    for s in nextStates(currNode.state):
      if not (s in queue) and not (s in visited):
        queue.append( node(s, currNode))
  return None, len(visited)
```

Listing 5.11. Breadth-first search

bfs() manipulates a `state` whose details will vary from problem to problem. The state is managed by a `node` class (searchUtils.py). Aside from the state, node also maintains a link to its parent, which is utilized to generate the path of states when the search has finished. There's also a cost variable, but that's only used by the A* algorithm described later.

bfs() utilizes two state functions, **isGoal()** and **nextStates()**, which will be redefined for each problem. For this puzzle, they're implemented in Listing 5.12 (jugs.py).

```
def isGoal(state):
  jug1, jug2 = state
  return ((jug1 == GOAL) and (jug2 == 0)) or \
         ((jug1 == 0) and (jug2 == GOAL))

def nextStates(state):
  jug1, jug2 = state
  sts = []
  # fill a jug
  sts.append((MAX_JUG1, jug2)) # Fill jug1
  sts.append((jug1, MAX_JUG2)) # Fill jug2
```

```
  # empty a jug
  sts.append((0, jug2)) # empty jug1
  sts.append((jug1, 0)) # empty jug2
  # pour from jug1 to jug2
  filljug2 = MAX_JUG2 - jug2
  if filljug2 > jug1:
    sts.append((0, jug2+jug1)) # Fill jug2 until jug1 empty
  else:
    sts.append((jug1-filljug2, MAX_JUG2)) # Fill jug2
  # pour from jug2 to jug1
  filljug1 = MAX_JUG1 - jug1
  if filljug1 > jug2:
    sts.append((jug1+jug2, 0)) # Fill jug1 until jug2 empty
  else:
    sts.append((MAX_JUG1, jug2-filljug1)) # Fill jug1
  return sts
```

Listing 5.12. jugs: isGoal() and nextStates()

A look inside isGoal() or nextStates() in Listing 5.12 reveals that the 'jugs' state is a tuple containing the amounts of water currently in the two jugs.

isGoal() returns true if either of the jugs holds the required amount (in this case the GOAL amount = 4) and the other is empty. nextStates() generates all the possible moves from the current jugs state – all combinations of filling and emptying the jugs, and pouring water from one to the other.

The search is initiated in Listing 5.13 (jugs.py).

```
path, numVisited = searchUtils.bfs((0,0))
print("No. of states visited:", numVisited)
if path == None:
  print("No path found");
  sys.exit()
```

Listing 5.13. Die Hard with Jugs

The result:

```
Max of jug1: 5
Max of jug2: 3
Goal: 4
No. of states visited: 16
Path:
(0, 0)
Increase jug1 by 5
(5, 0)
Decrease jug1 by 3
Increase jug2 by 3
(2, 3)
Decrease jug2 by 3
(2, 0)
Decrease jug1 by 2
Increase jug2 by 2
(0, 2)
Increase jug1 by 5
(5, 2)
Decrease jug1 by 1
Increase jug2 by 1
(4, 3)
Decrease jug2 by 3
(4, 0)
```

The number of visited states is printed as a measure of the search's memory usage. Also, a Cartesian grid is plotted (Fig. 5.10) showing the transitions between states, starting in (0,0) and finishing at (4,0).

Grids of this type are sometimes termed "mathematical billiards" due to the way that the trajectories between the states "bounce" off the edges of the space. Incidentally, Tartaglia's three jugs antecedent of this puzzle can be drawn as lines bouncing around a *triangular* pool table [**Oli93**].

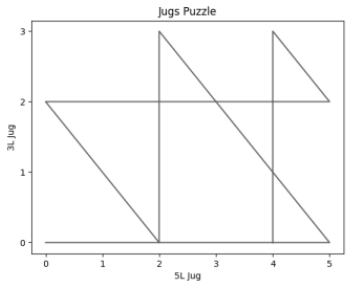


Figure 5.10. Cartesian grid for the jugs problem

bfs() terminates upon finding a solution even though alternative answers are possible. For example: (0,0) - (0,3) - (3,0) - (3,3) - (5,1) - (0,1) - (1,0) - (1,3) - (4,0), which is a longer path, so not reached by bfs().

5.5.2 The Missionaries and Cannibals Problem. Three missionaries and three cannibals are standing on the west bank of a river, along with a boat which can hold up to two people. The group want to reach the east bank but with the proviso that the cannibals should never outnumber the missionaries on either side or the missionaries will be eaten.

This puzzle can be traced back to the medieval text *Propositiones ad Acuendos Juvenes* by Alcuin (as Problem 17). However, his participants were brothers and sisters, and no woman could be in the company of another man unless her brother was present. There are many other variants involving the number of people in the two groups, the size of the boat, and sometimes even an island to act a staging post [**PS89**].

We'll use bfs() again, but with a more complex state tuple: (mWest, cWest, mEast, cEast, boatPos), where mWest = the number of missionaries on the west bank, cWest = the number of cannibals on the west bank, mEast = the number of east bank missionaries, cEast = the number of east bank cannibals, and boatPos = "west" or "east".

The isGoal() and nextStates() are implemented as in Listing 5.14 (missionaries.py).

```
def isGoal(state):
  # are all the missionaries and cannibals on the east bank?
  mWest, cWest, mEast, cEast, boatPos = state
  return (mEast == MAX_NUM) and (cEast == MAX_NUM)

def nextStates(state):
  mWest, cWest, mEast, cEast, boatPos = state
  sts = []
```

```
  if boatPos == "west":  # boat on west bank
    nboat = "east"
    if mWest > 1:
      sts.append((mWest-2, cWest, mEast+2, cEast, nboat))
    if mWest > 0:
      sts.append((mWest-1, cWest, mEast+1, cEast, nboat))
    if cWest > 1:
      sts.append((mWest, cWest-2, mEast, cEast+2, nboat))
    if cWest > 0:
      sts.append((mWest, cWest-1, mEast, cEast+1, nboat))
    if (cWest > 0) and (mWest > 0):
      sts.append((mWest-1, cWest-1, mEast+1, cEast+1, nboat))
  else: # boat on east bank
    nboat = "west"
    if mEast > 1:
      sts.append((mWest+2, cWest, mEast-2, cEast, nboat))
    if mEast > 0:
      sts.append((mWest+1, cWest, mEast-1, cEast, nboat))
    if cEast > 1:
      sts.append((mWest, cWest+2, mEast, cEast-2, nboat))
    if cEast > 0:
      sts.append((mWest, cWest+1, mEast, cEast-1, nboat))
    if (cEast > 0) and (mEast > 0):
      sts.append((mWest+1, cWest+1, mEast-1, cEast-1, nboat))
  return [s for s in sts if isValid(s)]

def isValid(state):
  mWest, cWest, mEast, cEast, boatPos = state
  if (mWest < cWest) and (mWest > 0):
    return False
  if (mEast < cEast) and (mEast > 0):
    return False
  return True
```

Listing 5.14. missionaries: isGoal() and nextStates()

The search is initiated by Listing 5.15.

```
path, numVisited = searchUtils.bfs((MAX_NUM, MAX_NUM, 0, 0, "west"))
print("No. of states visited:", numVisited)
if path == None:
  print("No path found");
  sys.exit()
```

Listing 5.15. Crossing the river safely

The result:

```
No. of states visited: 25
Path length: 12
West: 3m 3c; East: 0m 0c | Boat: west
  West -> East: 0m 2c
West: 3m 1c; East: 0m 2c | Boat: east
  East -> West: 0m 1c
West: 3m 2c; East: 0m 1c | Boat: west
  West -> East: 0m 2c
```

```
West: 3m 0c; East: 0m 3c | Boat: east
  East -> West: 0m 1c
West: 3m 1c; East: 0m 2c | Boat: west
  West -> East: 2m 0c
West: 1m 1c; East: 2m 2c | Boat: east
  East -> West: 1m 1c
West: 2m 2c; East: 1m 1c | Boat: west
  West -> East: 2m 0c
West: 0m 2c; East: 3m 1c | Boat: east
  East -> West: 0m 1c
West: 0m 3c; East: 3m 0c | Boat: west
  West -> East: 0m 2c
West: 0m 1c; East: 3m 2c | Boat: east
  East -> West: 1m 0c
West: 1m 1c; East: 2m 2c | Boat: west
  West -> East: 1m 1c
West: 0m 0c; East: 3m 3c | Boat: east
```

The output is also plotted as a 3D grid shown in Fig. 5.11.

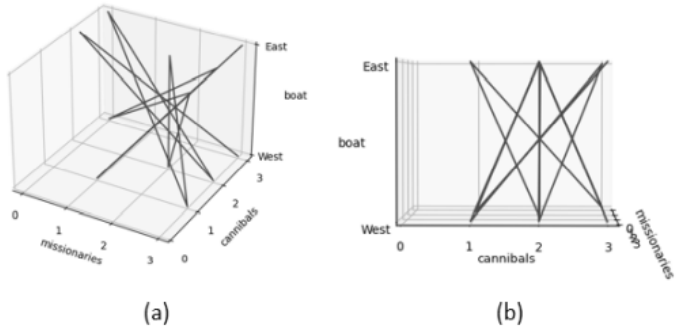


Figure 5.11. 3D plot for the missionaries problem

Matplotlib 3D plots can be rotated, which reveals an interesting symmetry for the movements of the cannibals (Fig. 5.11b) which is absent for the missionaries.

A more graphical representation of the movements of the groups is shown in Fig. 5.12, which also shows other solutions to the problem.

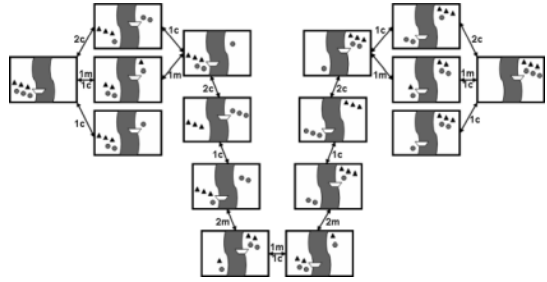


Figure 5.12. Search space for the missionaries (m) and cannibals (c)

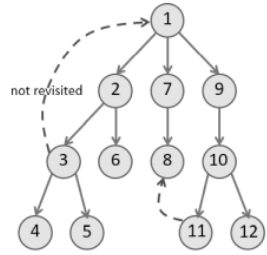


Figure 5.13. Nodes examined in DFS order

5.5.3 Searching a Maze. We'll use maze searching to discuss depth-first search (DFS) which, given the same state space as BFS, focuses on investigating as far along a branch as possible before backtracking to an earlier choice point and trying another branch. It's worth comparing the node numbering in Fig. 5.13 with Fig. 5.9 to see how the two algorithms differ. As with BFS, DFS must maintain a list of visited nodes to prevent it falling into cycles.

DFS generally produces a longer path than BFS, but one of its advantages is obscured by Fig. 5.13. A recursive implementation of DFS only investigates one branch at a time, and so doesn't need to store a large part of the tree in memory. Whereas BFS uses space proportional to $O(b^d)$, DFS is closer to $O(bm)$ where m is the maximum depth of the tree. However, this generally means that Python's limit for recursion depth needs to be increased.

Another reason for using a recursive DFS is that its code can be easily modified to improve its speed. This is illustrated in the dfs() function in Listing 5.16 (searchUtils.py).

```
def dfs(state, max=MAX):
    path = dfsDepth( node(state, None), 1, max)
    numVisited = len(visited)
    return path, numVisited

def dfsDepth(currNode, depth, max):
  if depth > max:  # constrain search depth
    return None
  visited.append(currNode.state)
  if isGoal(currNode.state):
    return currNode.getPath()
  for s in nextStates(currNode.state):
    if not (s in visited):  # no stack to examine
      path = dfsDepth( node(s, currNode), depth+1, max)
      if path != None:
        return path
  return None
```

Listing 5.16. Depth-first search

dfsDepth() shows that DFS can be coded succinctly as a recursive call inside a loop. The recursion implements the search down the tree and the backtracking, while the loop handles the traversal through the options at a choice point. Also, note that the state-specific functions, isGoal() and nextStates(), are still being used.

dfs() adds a depth limit to the basic algorithm which causes the recursion to backtrack upon reaching a specified call depth. This idea is at the heart of *iterative deepening* which repeatedly tries DFS searches with increasing depth limits until an answer is (hopefully) found.

Another important variant of DFS replaces recursion with a stack (see `searchUtils.py`). The coding is very similar to the BFS implementation except that BFS employs a queue. This approach may be preferable if recursion is limited (as in Python) or unavailable (as was the case in older languages such as Fortran 77).

The maze state is encoded as a 2D list. For example:

```
maze = [ [' ', '*', '*', '*', ' ', ' ', ' '],
         [' ', ' ', ' ', ' ', ' ', '*', ' '],
         ['*', ' ', '*', '*', '*', ' ', ' '],
         ['*', ' ', '*', '*', ' ', ' ', ' '],
         ['*', ' ', '*', '*', ' ', '*', ' '],
         ['*', ' ', '*', '*', ' ', '*', ' '],
         ['*', ' ', '*', '*', ' ', '*', ' '],
         ['*', ' ', '*', '*', ' ', '*', ' '],
         [' ', ' ', ' ', ' ', ' ', '*', ' '] ]
```

The start point is the top left corner, the goal is to reach the bottom right, and the '*'s are walls.

maze.py allows the user to choose between using bfs(), dfs() or dfsIter() to traverse the maze, with the following results:

```
BFS
No. of states visited: 40
Path length: 17
-----------------
| . * * * . . . |
| . . . . . * . |
| *   * * *   . |
| *   * *     . |
| *   * *   * . |
| *   * *   * . |
| *   * *   * . |
| *   * *   * . |
|           * . |
-----------------

DFS Recursive
No. of states visited: 25
Path length: 25
-----------------
| . * * *       |
| . .       *   |
| * . * * *     |
| * . * * . . . |
| * . * * . * . |
| * . * * . * . |
| * . * * . * . |
| * . * * . * . |
|   . . . . * . |
-----------------

DFS Iterative
No. of states visited: 25
Path length: 25
-----------------
| . * * *       |
| . .       *   |
| * . * * *     |
| * . * * . . . |
| * . * * . * . |
| * . * * . * . |
| * . * * . * . |
| * . * * . * . |
|   . . . . * . |
-----------------
```

As expected, the two DFS versions produce the same path, which is longer than the one found by BFS. Note though that the number of states was greater for BFS than DFS (40 vs. 25) which reflects the amount of memory used.

In dfs(), the maximum depth (MAX) was set to 30 since the maze grid is 10-by-10, and so even the shortest path will use 19 states (a player can only move horizontally or vertically).

isGoal() and nextStates() deal with the maze state (Listing 5.17; maze.py).

```
def isGoal(state):
  (r, c) = state    # at bottom right corner?
  return (r == len(maze)-1) and (c == len(maze[0])-1)

def nextStates(state):
  (r, c) = state
  sts = []
  for dir in DIRS: # try all dirs
    r1 = r+dir[0]; c1 = c+dir[1]
    if isValid(r1,c1):
      sts.append((r1,c1))
  return sts

def isValid(r, c):
  return ((r >= 0) and (r < len(maze)) and
       (c >= 0) and (c < len(maze[0])) and
       (maze[r][c] == ' '))
```

Listing 5.17. maze: isGoal() and nextStates()

The available directions are implemented as step offsets in a DIRS list:

```
DIRS = [(1,0), (0,1), (-1,0), (0,-1)]  # down, left, up, right
```

5.5.4 The 8-Puzzle. The 8-puzzle problem is a 3-by-3 board holding 8 slideable tiles (numbered 1 to 8) and an empty space (which we'll label as 0). The goal is to rearrange the tiles into the numbered ordered:

```
0 1 2
3 4 5
6 7 8
```

It's possible to move a horizontal or vertical neighbor of the 0 tile into its space, which causes the tile's number and 0 to be swapped.

This is a version of the famous 15-puzzle, which for many years was believed to have been devised by the great American game designer Sam Loyd. Martin Gardner edited two collections of Loyd's puzzles which are well worth a study [**LG59**, **LG60**]; the 15-puzzle appears in Vol. 1.

Extensive game play with the 8-puzzle has established that reaching the solution from some arbitrary starting state takes an average of 22 moves. Also, the number of choices (or branching factor) at each move is about 3. (When the

empty tile is in the middle, four moves are possible; when it's in a corner, two; and when it's at an edge, three.) This implies that an exhaustive tree search to depth 22 will look at about $3^{22} \approx 3.1 \times 10^{10}$ states. Things are nowhere near that bad in practice since only about $9!/2 = 181,440$ distinct states are reachable based on permissible tile operations.

Nevertheless, it's necessary to add heuristics to guide the search to the goal state within a reasonable amount of time. We'll utilize A* (AStar) which applies a cost function $f()$ to each node in the search tree, and then considers the nodes in increasing cost order. The function has two parts: $g()$, the cost of getting from the start state to this node (usually represented by the length of the path), and $h()$ an estimate of the cost of getting from this node to the goal state. This is expressed as:

$$f(n) = g(n) + h(n)$$

where n is the current node. $h(n)$ is a heuristic since we don't have exact knowledge about the cost of reaching the goal.

Probably the most natural $h()$ estimate for sliding puzzles is to sum the 'taxicab' distances between the tiles and their final positions in the goal. This is the smallest number of horizontal and vertical moves between the two positions, and is sometimes called the Manhattan distance since it's similar to the way that we count the number of city blocks between two locations.

For example the Manhattan distance between the start state and goal in Fig. 5.14 is $2+3+1+2+2+2+3+3+2 = 20$ where the tiles are considered in the order 0 to 8.

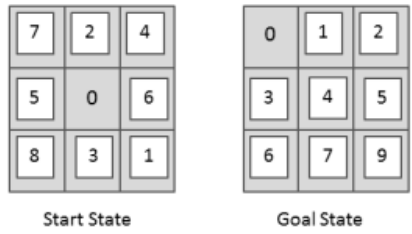


Figure 5.14. 8-puzzle start and goal states

A* is much faster than BFS or DFS but may not return the shortest path to the goal because of the way that the cost function rearranges the nodes.

The implementation shown below (Listing 5.18; searchUtils.py) looks similar to bfs() but the queue data structure is replaced by a priority queue (Python's heapq class). When a node is added to a priority queue it's stored in increasing order based on its cost field. When a node is removed, the one with the smallest cost is selected.

```
def aStar(startState):
  priQueue = []
  startNode = node(startState, None)
  startNode.setCost()
  heapq.heappush(priQueue, startNode)
  visited = []
  while priQueue != []:
    minNode = heapq.heappop(priQueue)
    visited.append(minNode.state)
    if isGoal(minNode.state):
      return minNode.getPath(), len(visited)
    for s in nextStates(minNode.state):
      if not (s in visited):
        # allow duplicate states to be inserted
        # since they are sorted by cost
        nd = node(s, minNode)
        nd.setCost()
        heapq.heappush(priQueue, nd)
  return None, len(visited)
```

Listing 5.18. A* search

The heap combines the access speeds of an array with a tree ordering of its elements, which makes it faster that using a sorted queue. Both the pushing and popping operations have $O(\log n)$ running time, where n is the size of the heap.

Another change is that aStar() uses the setCost() method in the node class:

```
  def setCost(self):
    self.cost = len(self.getPath())-1 + goalDist(self.state)
```

goalDist() is the heuristic function $h()$, and is defined along with isGoal() and nextStates() in puzzle8.py (see Listing 5.19).

```
def goalDist(loc):
  ''' The sum of the Manhattan distances
      (sum of the vertical and horizontal dists)
      from the tiles to their goal positions
  '''
  tot = 0
  for i in range(len(loc)):
    idx =  loc.index(i)   # loc of tile i
    row, col = idx//SIZE, idx%SIZE
    idxG =  GOAL.index(i) # goal loc of tile i
    rowG, colG = idxG//SIZE, idxG%SIZE
    tot += abs(row-rowG) + abs(col-colG)
  return tot
```

Listing 5.19. 8-puzzle: goalDist()

puzzle8.py gives the user the option of employing BFS, A*, or DFS to search for an answer. All of these utilize the same isGoal() and nextStates() functions.

In addition to reporting the path (if there is one) and the number of visited states, the time and tracemalloc Python modules are employed to measure the elapsed time and the peak amount of memory used. The results for four different starting states are shown in Table 5.1.

Start Tile	Algorithm	Path length	No. of states	Duration (secs)	Memory (KB)
1 2 5	BFS	4	11	0.000	9
3 4 0	A*	4	4	0.000	5
6 7 8	DFS	4	3222	0.440	479
1 4 2	BFS	9	194	0.020	102
6 5 8	A*	9	12	0.000	10
7 3 0	DFS	19	1776	0.170	268
1 0 2	BFS	24	1906531	11 mins	451473
7 5 4	A*	24	2420	0.340	1128
8 6 3	DFS	no path	7165	1.810	1006
0 8 7	BFS	aborted			
6 5 4	A*	31	16871	12.152	7402
3 2 1	DFS	no path	7500	2.051	1047

Table 5.1. 8-Puzzle algorithm statistics

A graphical representation of BFS's search with the tile setup in row 1 of Table 5.1 is shown in Fig. 5.15. Fig. 5.15 illustrates why the path length is 4 and the search visits 11 states. Recall that BFS traverses the levels left-to-right in a top-down fashion.

Table 5.1 shows that DFS is the obvious loser, partly due to it using a maximum depth setting of only 20 which means that it couldn't investigate a large enough subtree to find answers for the last two tile configurations.

The last two tests also show the drawbacks of basic BFS – the waiting time grew so long on the last test that we aborted the execution, and the time and memory usage for the previous tile settings in rows 2 and 3 were very poor. A* was the easy winner, and only required a 10-line goalDist() function to achieve it.

5.5.5 The n-Queens Problem. We wish to *peacefully* place 8 queens on an 8x8 chessboard, meaning that there are no queens in the same row, column, or diagonal (otherwise they'd *attack* each other). Our solution will generalize the problem to *n*-queens on a *nxn* chessboard.

A particularly poor search approach for 8-queens would be to consider all $\binom{64}{8}$ placements, which results in around 4.42×10^9 tests. Fortunately, position

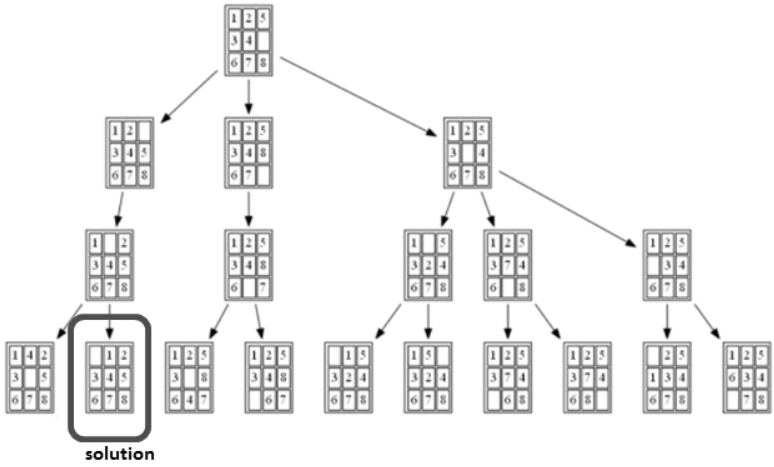


Figure 5.15. BFS search for tile 1

constraints can reduce the search space considerably: the queen in the first column has 8 position choices, the queen in the second column has only 7 choices, the queen in the third column has 6 choices, and so on, resulting in a search space of just $8! = 40,320$. We can also add diagonal constraints to further reduce the size to 2,057.

This approach suggests that we don't need to employ a 2D list for board placements, since there's only one queen per column. A single list will suffice, with an element's index acting as the column number, and the row position stored as the element's value. For instance, the solution in Fig. 5.16 can be written as [3, 5, 7, 1, 6, 0, 2, 4]. Note: we start counting rows and columns from 0, starting from the top-left corner of the board.

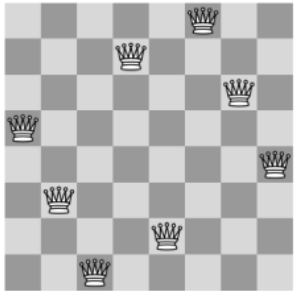


Figure 5.16. One solution for the 8-queens problem

The 8-queens version of the puzzle has 92 distinct solutions, and if symmetric placements based on rotation and reflection aren't counted separately, then there are just 12 unique configurations.

We could again use A* or BFS, but the solution steps described above suggests that we could use recursive DFS based around an examineColumn() function. It places queens column-wise, beginning with the

first column, by setting queens[0] = 0 (we're counting from 0). Then we tentatively move the second queen down the second column until we find a square that isn't attacked (e.g. queens[1] = 2). Then we move the third queen in the third column down until we find a square not attacked by the first two queens, and so on.

If we can't place a queen because each square of its column is attacked, then examineColumn() must backtrack and try to move the preceding queen further down, otherwise the function will have to backtrack further, to the previous column.

row[i] = True	Row i is not attacked
upl[k] = True	Diagonal with $i + j = k$ is not attacked; k takes values from 2 to $2n$.
upr[k] = True	Diagonal with $i - j = k$ is not attacked; k takes values from $1 - n$ to $n - 1$.

Table 5.2. Attack tests for a cell

Unlike the previous examples, we want the search to continue after the first solution. An answer will be printed, a global counter incremented, and then the search will continue as if the solution had never been encountered. In this way, the code will report all solutions in increasing row order, meaning that even the first queen will tried out in the n-th row eventually.

examineColumn() in Listing 5.20 (nqueens.py) utilizes the three boolean lists in Table 5.2, where upl (short for "up-and-left") and upr ("up-and-right") store the results of diagonal tests.

```
def examineColumn(i):
  global count
  for j in range(n):
    if isValid(i,j):
      queens[i] = j
      # col i queen is placed in row j
      # this queen attacks these squares
      uprIdx = (i-j) + (n-1)
      row[j] = False
      upl[i+j] = False; upr[uprIdx] = False
      if i < n-1:
        examineColumn(i+1)
      else:  # found a new solution
        count += 1
      # undo changes for backtracking
      row[j] = True
      upl[i+j] = True; upr[uprIdx] = True
```

Listing 5.20. Solving n-queens

The results when $n = 8$ are:

```
Board 1
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .

Board 2
Q . . . . . . .
. . . . . . Q .
. . . Q . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .

  : many results not shown

Board 92
. . Q . . . . .
. . . . . Q . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . Q . . .
. . . . . . Q .
Q . . . . . .

Found 92 solutions
```

The initial call to the examineColumn(0) is surrounded by code for measuring the elapsed time and memory usage (which we've not listed here). The results are reported in Table 5.3.

n	no. of places	time (secs)	memory (bytes)
8	92	0.015	1152
9	352	0.015	1288
10	724	0.078	1328
11	2680	0.356	1368
12	14280	1.807	1408
13	73712	10.406	1448
14	365596	62.492	1488
15	2279184	404.118	1528
16	14772512	2757.066	1568
17	95815104	19566.369	1672

Table 5.3. n-queens statistics

The time column suggests that we are dealing with exponential running time. To be a little more precise, an upper bound for the running time for examineColumn() on an n-by-n board considers n values for column 0, $n-1$ values for column 1, and so on, resulting in $O(n!)$. Factorial running time is actually worse than exponential.

The memory column is pleasingly small since the only significant data structures are the positions list and the stack of recursive calls, which both increase linearly with the size of n.

History of the problem: 8-queens was first posed in 1848 by Max Bezzel in a chess magazine, but went unnoticed at the time. It was suggested again by Franz Nauck in the popular *Illustrierte Zeitung* (June 1, 1850) and this time caught people's attention, including that of Carl Friedrich Gauss. On 21 September 1850, the blind Dr. Nauck published all 92 solutions in the same journal, beating Gauss who had only discovered 72 answers up to that time. For a much fuller survey of the problem, consult Bell [**BS09**].

Exercises

(1) A knight is placed on any square of an *nxn* chessboard with the task of visiting every square exactly once. Write a backtracking algorithm for this "knight's tour" when $n = 8$. Use a 2D board to store the current move number. Don't forget to set board[x][y] = 0 when you have to backtrack.

(2) Solve the knight's tour problem for the 8 x 8 board using divide-and-conquer. First, tackle the problem for the lower left 4x5 board in Fig. 5.17, extend it to the lower right 4 x 5 board, and finally to the upper 8 x 3 board. One possible solution is shown in Fig. 5.17 where the numbers indicate the sequence of visits.

60	63	50	53	56	43	48	45
51	54	59	62	49	46	57	42
64	61	52	55	58	41	44	47
3	8	15	20	27	22	35	40
16	11	4	9	34	39	28	23
7	2	19	14	21	26	31	36
12	17	10	5	38	33	24	29
1	6	13	18	25	30	37	32

Figure 5.17. One solution to the knight's tour (Ex. 2)

A. J. Schwenk ('Which rectangular chessboards have a knight's tour?', *Math. Magazine* 64, 1991, 325-332, `https://www.mimuw.edu.pl/~rytter/TEACHING/ALCOMB/schwenk.pdf`) generalized this method to construct closed knight tours for all boards where closed tours exist.

(3) *A Hamiltonian circuit* or *path* of a graph visits every node exactly once. The exercises above asked us to find Hamiltonian paths in the graphs whose nodes are squares of a board, with two nodes connected if they are a knight-jump apart.

Finding a Hamiltonian circuit or path is an NP-complete problem. For these problems no method is known which always works *and* is always faster than

simply trying every possibility. However, there are methods which find a Hamiltonian cycle quite quickly for most graphs that have one.

Write a program to find closed knight's tours using the following method developed by Lajos Pósa (details can be found in Schwenk's paper cited above). Construct a path $H_0, H_1, ..., H_l$ by adding nodes until we can not extend it any further. Then H_l must be connected to some H_r with $r \leq l - 2$. Then $H_0, ..., H_r, H_l, H_{l-1}, ..., H_{r+1}$ is also a path. We call the operation of replacing our previous path by this one a *rotation*. Try to extend the new path, and if it can't be done, then rotate again since there are usually choices when making extensions or rotations. When all the nodes have been added to the path, perform more rotations to generate a Hamiltonian cycle. A good way to avoid falling into a loop is to insert a random element into the choices, but the process may still get stuck.

(4) Implement Warnsdorff's rule (1823): the knight should always jumps to a cell from which it can jump to the *fewest* squares not previously visited. In case of several suitable moves choose any one of them. This rule also seems to work for *mxn* boards which contain Hamiltonian paths, but there's no proof that it always works. For instance, sometimes we reach a dead end as a result of having made an unfortunate choice earlier. However, there's no known example where a tour can not be completed without violating Warnsdorff's rule. The rule generates Hamiltonian paths, not cycles.

(5) Show that the 15-puzzle is not solvable if and only if its start state is an odd number of swaps away from the goal state. This property was used by Loyd to make his swapped 14-15 version of the puzzle impossible to solve. For details, see the paper by Keith Conrad at `https://kconrad.math.uconn.edu/blurbs/grouptheory/15puzzle.pdf`.

(6) Another popular heuristic for the 8-puzzle is the *number of misplaced tiles*. In Fig. 5.14 that value would be 9. Implement this heuristic and compare its performance to the Manhattan distance.

(7) Solve the *n*-queen problem using A*, which will require you to devise an *h*() function for the state. One approach is to calculate the number of queens that attack a square, which can utilize code in examineColumns().

(8) Aside from the brothers and sisters version of the Missionaries and Cannibals problem, Alcuin also debuted the wolf, goat, and cabbage puzzle. A farmer with a wolf, a goat, and a cabbage must cross a river by boat. The boat can carry only the farmer and a single item. If left unattended, the wolf will eat the goat, or the goat will eat the cabbage. How can they all cross the river without anything being eaten?

(9) Four missionaries are on one side of the river and four cannibals on the other side. Each group wishes to get to the opposite bank and the only means of transport is a rowing boat big enough for three people. Only one missionary and one cannibal know how to row. The cannibals will eat the missionaries as soon as they outnumber them, whether in the boat or on either bank. Can the crossing be completed successfully?

What if there are five missionaries and five cannibals and the boat still carries only three people? What if there are six missionaries and six cannibals?

(10) An entire category of search puzzle that we left out are *train shunting* problems, which involve rearranging train carriages on a track based on a limited set of permitted moves.

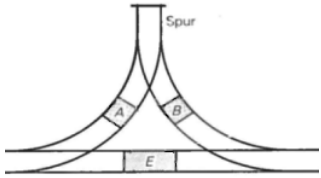


Figure 5.18. Switching trucks A and B

For example, two trucks A and B are in a siding as in Fig. 5.18. Can engine E shunt A into B's position and B into A's position and finish up exactly where it started? Each truck will just fit on the spur (the top part of the two sidings), so it's possible to shunt a truck onto the spur from one siding and pull it off from the other. However, the engine is too big to fit into the spur.

5.6 Permutations

5.6.1 Permutations as Arrangements. An arrangement of three different objects, such as 1, 2, and 3, is called a *permutation*. There are six permutations of 1, 2, and 3: 123, 132, 213, 231, 312, 321. More generally, the product rule tells us that there are $n!$ permutations of n distinct objects.

Listing 5.21 (perms.py) permutes a list by random shuffling its elements.

```
def shuff(elems):
  n = len(elems)
  for i in range(n-1, 1, -1):
    rIdx = random.randint(0, i-1)
    elems[i], elems[rIdx] = elems[rIdx], elems[i]
```

Listing 5.21. Randonly shuffling a list

For example:

```
>>> from perms import *
>>> elems = [1, 2, 3]
>>> shuff(elems); print(elems)
[1, 3, 2]
>>> shuff(elems); print(elems)
[1, 2, 3]
>>> shuff(elems); print(elems)
[3, 2, 1]
```

Python's random module includes a shuffle() function that does much the same thing.

```
>>> import random
>>> random.shuffle(elems); print(elems)
[1, 3, 2]
>>> random.shuffle(elems); print(elems)
[1, 2, 3]
>>> random.shuffle(elems); print(elems)
[3, 1, 2]
```

This approach isn't suitable if you want to iterate through all the possible permutations since each call produces a random result. Listing 5.22 (perms.py) implements a simple backtracking algorithm for printing all the possibilities.

```
def permute1(elems):
  perm(elems, 0, len(elems))

def perm(elems, left, right):
  if left == right:
    print(elems)
  else:
    for i in range(left, right):
      elems[left], elems[i] = elems[i], elems[left]
      perm(elems, left+1, right)
      elems[left], elems[i] = elems[i], elems[left]
```

Listing 5.22. Printing all the permutations

For example:

```
>>> elems = [1, 2, 3]
>>> permute(elems)
[1, 2, 3]      [1, 3, 2]      [2, 1, 3]
[2, 3, 1]      [3, 2, 1]      [3, 1, 2]
```

This function prints all the permutations and then returns, which is not always ideal. Typically, we'd like to obtain each permutation as a separate value or collected together in a list. This requirement is a good reason for using Python's `yield`, as in Listing 5.23 (perms.py).

```
def permuteY(elems):
  if len(elems) <= 1:
    yield elems
    return
  for p in permuteY(elems[1:]):
    for i in range(len(elems)):
      yield p[:i] + elems[0:1] + p[i:]
```

Listing 5.23. Permutations one-by-one

For example:

```
>>> elems = [1, 2, 3]
>>> pg = permuteY(elems)
>>> next(pg)
[1, 2, 3]
>>> next(pg)
[2, 1, 3]
>>> next(pg)
[2, 3, 1]
```

The generator can be induced to store all of the results in a list by surrounding it with a `list()`:

```
>>> elems = [1, 2, 3]
>>> ps = list( permuteY(elems))
>>> print(ps)
[[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]
```

5.6.2 Permutations as Rearrangements. A permutation can also be regarded as a *rearrangement* function that specifies how one ordering is transformed into another. For instance, the permutation 5 4 6 9 7 8 10 2 3 1 can be thought of as the output of the function f_1 that rearranges {1, 2, ..., 10}:

$$f_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 5 & 4 & 6 & 9 & 7 & 8 & 10 & 2 & 3 & 1 \end{pmatrix}.$$

Some other permutations of 1..10:

$$f_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 4 & 5 & 1 & 2 & 9 & 7 & 8 & 10 & 6 \end{pmatrix}$$

$$f_3 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 9 & 2 & 1 & 6 & 5 & 4 & 10 & 7 & 8 \end{pmatrix}$$

$$f_4 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 2 & 4 & 8 & 3 & 10 & 9 & 1 & 6 & 5 & 7 \end{pmatrix}$$

A permutation function f can be represented by a graph. Choose n points and label them from 1 to n. For each $i \in 1..n$ draw an arrow from i to $f(i)$. Fig. 5.19 shows the graphs for f_1 to f_4.

Each graph typically contains several cycles, which can be represented by the cycle notation underneath each one in Fig. 5.19. Note that fixed points, i.e. points where $f(i) = i$, are included. The permutation f_4 is termed *cyclic* because it has just one cycle.

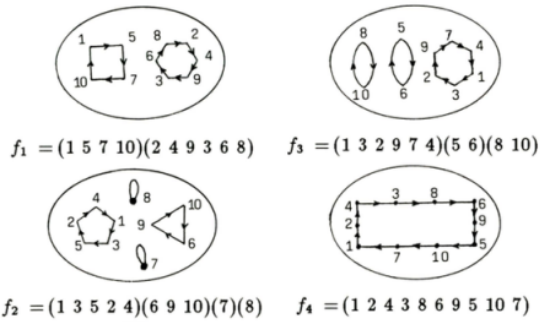


Figure 5.19. Permutation functions as graphs

Listing 5.24 (cycles.py) determines the cycles in the rearrangements stored in the ps permutation of 1..*n*.

```
n = int(input("n=? "))
ps = list(range(1,n+1))  # [1..n]
random.shuffle(ps)
print("Rearrangements:")
for i in range(n):
  print(f"{(i+1):2d} --> {ps[i]:2d}")
print("Cycles:")
for i in range(n):
  first = i
  if ps[i] > 0:  # start a cycle
    print("(", end =' ')
    while True:
      r = first
      print(r+1, end = ' ')
      first = ps[first]-1
      ps[r] = -ps[r]
      if first == i:
        break
    print(")") # end cycle
```

Listing 5.24. Finding cycles

An example:

```
> python cycles.py
n=? 10
Rearrangements:
 1 -->  5
 2 -->  7
 3 -->  3
 4 -->  1
 5 --> 10
 6 -->  9
 7 -->  8
 8 -->  6
 9 -->  4
10 -->  2
Cycles:
( 1 5 10 2 7 8 6 9 4 )
( 3 )
```

The algorithm starts with $i = 0$ and sets first=i. Then it repeatedly updates first = ps[first]-1 until i reoccurs, thereby closing the cycle. When an element joins a cycle it's marked by reversing its sign. When a cycle is completed, the code looks for the next unmarked element to start the next cycle.

5.6.3 The Josephus Permutation. We return to the Josephus Problem (n,k) of Sec. 1.7: n people are arranged in a circle, and every k-th person is eliminated, beginning with number k. Find the number x of the s-th eliminated person.

We now want to print the sequence $J(n,k)$ of eliminations – the *Josephus Permutation* of $1..n$. Only a slight modification of Listing 1.28 is needed, resulting in Listing 5.25 (josPerm.py).

```
n,k = map(int, input("n k=? ").split())
s = 0
while s < n:
  s += 1
  x = k*s
  while x > n:
    x = int((k*(x-n)-1)/(k-1))
  print(x, end =' ')
```

Listing 5.25. The Josephus Permutation

An example, represented by Fig. 5.20:

```
> python josPerm.py
n k=? 10 2
2 4 6 8 10 3 7 1 9 5
```

The last man standing is number 5.

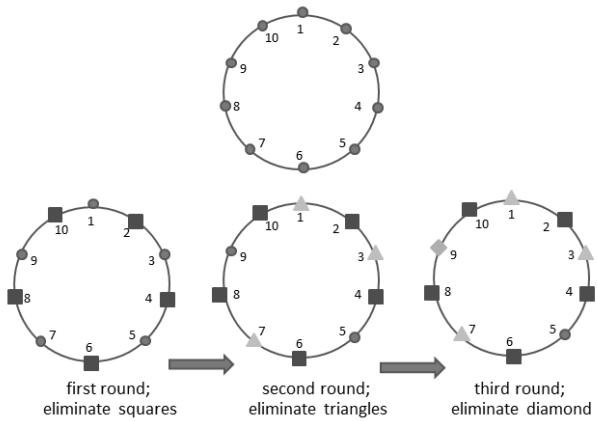


Figure 5.20. The Josephus problem (10,2)

Exercises

(1) Modify Listing 5.24 so that it counts the number of cycles.

(2) Generate multiple permutations of $1..n$, count the number of cycles in each one, and estimate the expected number of cycles in a random n-permutation. It can be shown that it equals the Harmonic series for n terms, $H_n = 1 + 1/2 + 1/3 + ... + 1/n$.

(3) We have n boxes, each with its own unique key. After mixing the keys at random, we drop one into each box which automatically locks shut. Now we smash open k random boxes. What's the probability $p(n, k)$ that we can use the k recovered keys to open the remaining boxes?

This was a problem in the 1971 Kürschák competition in Hungary (with $k = 2$). There are short but difficult ways of deriving the surprisingly simple formula for $p(n, k)$, but it may be easier to find the formula by simulation. First we translate the problem into the language of permutations: choose a random permutation p of $1..n$. What's the probability that the cycles in p containing $1..k$ also cover all the elements from $1..n$?

(4) Consider the following variation of Ex. 3: break open just one box and unlock all the boxes you can. Now break open another box and unlock all the additional boxes. Repeat this until you have opened every box. How many boxes do you have to break open on average?

(5) Show that the following Josephus permutations are cyclic:

a) $J(n, 2)$ for $n = 2, 5, 6, 9, 14, 18$;

b) $J(n, 3)$ for $n = 3, 5, 27, 89, 1139, 1219, 1921, 2155$;

c) $J(n, 4)$ for $n = 5, 10, 369, 609$;

d) $J(n, 7)$ for $n = 11, 21, 35, 85, 103, 161, 231$.

(6) *The golden permutation.* Let $t = (\sqrt{5}-1)/2$. The n pairs of numbers $(i, \text{frac}(i \times t))$ for $i = 1..n$ are sorted so their second values are in increasing order. (frac() returns the fractional part of a float.) When the corresponding first values are printed, the result is the so-called "golden permutation" of $1..n$. Because of its properties it's sometimes preferred by statisticians to a random permutation of $1..n$.

a) Write a program which prints the golden permutation of $1..n$ for $n = 100$.

With suitable coding additions check the following surprising properties:

b) Calculate the *absolute difference* between an element of the permutation and the previous one. (The last element can be regarded as preceding the first one.) Only three different numbers will occur in the entire permutation.

c) Elements with a '1' difference have a *distance* of 38, 39, or 62. (A distance is one more than the number of elements between two numbers with the same difference.) If these elements are organized to form the vertices of a regular 100-gon, only two shortest distances will be left over – 38 and 39.

d) Suppose a random sample of ten elements is to be selected from a sequence of consecutive integers, e.g. from 38..77. We start anywhere in the permutation, for example at 48, and choose successively those elements that lie in our

chosen interval 38..77. For instance, we might obtain 57, 70, 49, 62, 41, 75, 54, 67, 46, and 59, which is unusually uniformly distributed in the 38..77 range. Note also that if we order the ten numbers according to magnitude, only the differences 3 and 5 occur. Also, if we list numbers in their order of occurrence, then only the differences 13, 21, and 34 appear.

e) Consider the segment 38..77 on the number line. If the points 57, 70, 49,... are marked successively, then the current point falls in one of the currently largest free intervals, or in an end-interval.

f) Replace t in the code by t^2. Which of the b) to e) properties remain valid?

g) Replace t in the code by a different irrational number, such as $\sqrt{2}$, π , or e. Which properties remain valid?

h) Investigate b) to g) for values of n other than 100.

i) Sort the pairs of numbers $(i, 55 \times i \bmod 89)$ for $i = 0..88$, so that their second values are in increasing order. Check properties b) to e) for this permutation. Note that 55 and 89 are successive Fibonacci numbers. Repeat this question but with two relatively prime integers that are not Fibonacci numbers.

5.7 Coupled Difference Equations

How many words consisting of n digits from the alphabet A= {0, 1, 2, 3, 4} can be formed with the restriction that neighboring digits inside a word differ by exactly 1?

Table 5.4 outlines what this means for the words that can be built.

Words starting with ...	Digit Sequences
0	0 1 ...
1	1 0 ...
	1 2 ...
2	2 1 ...
	2 3 ...
3	3 2 ...
	3 4 ...
4	4 3 ...

Table 5.4. Possible word forms

Let x_n be the total number of words, and y_n, z_n, u_n the number of n-words starting with 0 (or 4), 1 (or 3), and 2. Then we can write four difference equations

$$\begin{aligned} x_n &= 2y_n + 2z_n + u_n \\ y_n &= z_{n-1} \\ z_n &= y_{n-1} + u_{n-1} \\ u_n &= 2z_{n-1} \end{aligned} \tag{5.7}$$

The boundary cases for these are when a word consists of just 1 digit, which means that $y_1 = 1, z_1 = u_1 = 1, x_1 = 5$.

Each of the equations can be mapped to a function that returns a count of words based on a supplied n argument, as in Listing 5.26 (coupledRecurrences.py).

```
def x(n):
  return 2*y(n) + 2*z(n) + u(n)

def y(n):
  if n == 1: return 1
  else:
    return z(n-1)

def z(n):
  if n == 1: return 1
  else:
    return y(n-1) + u(n-1)

def u(n):
  if n == 1: return 1
  else:
    return 2*z(n-1)
```

Listing 5.26. Implementing the difference equations

Table 5.5 shows the output when $n = 17$.

A convenient way to visualize the grammar is using a graph with four states for the alphabet's digits (see Fig. 5.21). If there is a two-way connection between two states, they're linked with a single line without an arrow.

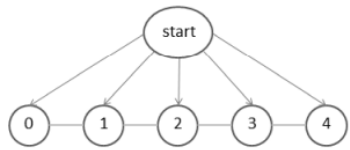


Figure 5.21. Graph of the word grammar

n	x_n	y_n	z_n	u_n
1	5	1	1	1
2	8	1	2	2
3	14	2	3	4
4	24	3	6	6
5	42	6	9	12
6	72	9	18	18
7	126	18	27	36
8	216	27	54	54
9	378	54	81	108
10	648	81	162	162
11	1134	162	243	324
12	1944	243	486	486
13	3402	486	729	972
14	5832	729	1458	1458
15	10206	1458	2187	2916
16	17496	2187	4374	4374
17	30618	4374	6561	8748

Table 5.5. Output from Listing 5.26 for $n = 17$.

Exercises

(1) Study Table 5.5 and find closed formulas for x_n, y_n, z_n, u_n. *Hint*: consider even and odd n values separately.

(2) How many n-words from the alphabet {0, 1, 2, 3} do *not* contain the two words 01 or 10? Write a program based on the difference equations for this problem.

(3) How many n-words from the alphabet {0, 1, 2} have neighboring digits differing by at most 1?

5.8 Games

In this section, we'll look at turn-based games for two players, A and B. Player A always moves first but the rules are otherwise the same for both participants.

We're given a starting state and the set M of legal moves. A player loses if he finds himself in a position from which no legal move can be made.

We can think of each game position as a vertex of a graph, and each move as a directed edge. We'll restrict our games to only having finitely many vertices (positions) without circuits (i.e. a position can only be used once). This ensures that one of the players will eventually lose.

It can be shown by induction on the number of moves that the set P of all positions can be split in two: a set L of losing positions, and a set W of winning positions: $P = L \cup W, L \cap W = 0$ (see Fig. 5.22).

A player finding himself in an L position will lose provided his opponent plays optimally, while a player in a W position can force a win whatever his opponent may choose to do. In order to win, a player must always move so as to force his opponent to choose a move to an L position. From each L position, every move may end in a W position, but for every W position, a move to an L position is possible. L must contain at least one final position f from which there is no escape. The player who leaves his opponent facing such a position has won the game. These criteria are illustrated in Fig. 5.22.

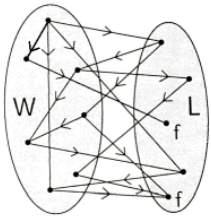


Figure 5.22. Winning (W) and losing (L) game positions

Our problem is to identify the set L of losing positions for a particular game.

5.8.1 Nim. Nim has two players take turns removing chips from several piles. On each turn, a player must remove at least one chip, and may remove any number provided they all come from the same pile. The player removing the last chip (or chips) wins.

A complete mathematical analysis of Nim was worked out by C.L. Bouton in 1902, and it's been a popular topic ever since, discussed by Rouse Ball and Coxeter [**BC87**], Kasner and Newman [**KN01**], and Beasley [**Bea06**].

Nim Game 1. Two players take turns to reduce a *single* pile of n chips. A move consists in removing m chips, where m is a number in the set $\{1, 3, 8\}$. The winner is the one who removes the last chip. Listing 5.27 (nim.py) prints the losing positions for this version.

```
n = int(input("n=? "))
wins = [False]*(2*n)
legalMoves = [1, 3, 8]
for i in range(n):
  if not wins[i]:
    for m in legalMoves:
      wins[i+m] = True
print("Losing Positions:")
for i in range(n-1):
  if not wins[i]:
    print(f"{i:2d} ", end='')
  else:  # a win
    print(" - ", end ='')
  if (i+1)%11 == 0:
    print()
```

Listing 5.27. First Nim game

The output is for a game starting with 100 chips.

```
> python nim.py
n=? 100
Losing Positions:
 0  -  2  -  4  -  6  -  -  -  -
11  - 13  - 15  - 17  -  -  -  -
22  - 24  - 26  - 28  -  -  -  -
33  - 35  - 37  - 39  -  -  -  -
44  - 46  - 48  - 50  -  -  -  -
55  - 57  - 59  - 61  -  -  -  -
66  - 68  - 70  - 72  -  -  -  -
77  - 79  - 81  - 83  -  -  -  -
88  - 90  - 92  - 94  -  -  -  -
```

Position '0' is the initial move (by Player A) which has the potential of being a losing position. The last position is for the 99th move, which is printed as '-' indicating that it is a winning position, since the player can remove the last chip.

The output strongly suggest a pattern with a period of 11. The L set consists of all the non-negative integers of the form $11k, 11k+2, 11k+4, 11k+6$.

This variant of Nim is sometimes called the *Subtraction Game* and its analysis is particularly simple. Player A will lose if and only if $n \pmod{m+1} = 0$, where n is the size of the pile and m is one of the legal moves. In this example, $n = 100$, and the modulo produces 0 when $m = 1$ or $m = 3$. If Player B knows what they are doing, Player A has no chance of winning.

Nim Game 2. In this version, the legal moves are the set of squares {1, 4, 9, 16, 25, 36, ...}. Listing 5.28 (squareNim.py) prints the losing positions.

```
n = int(input("n=? "))
wins = [False]*(n*n)
legalMoves = [(i*i)
  for i in range(1, int( math
      .sqrt(n))+1) ]
print(legalMoves)
for i in range(n):
  if not wins[i]:
    for m in legalMoves:
      wins[i+m] = True
print("Losing Positions:")
for i in range(n-1):
  if not wins[i]:
    print(f"{i:3d} ", end='')
  else:  # a win
    print(" -- ", end ='')
  if (i+1)%10 == 0:
    print()
print()
```

Listing 5.28. Second Nim game

The output is for a game starting with 130 chips.

```
> python squareNim.py
n=? 130
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
Losing Positions:
  0  --   2  --  --   5  --   7  --  --
 10  --  12  --  --  15  --  17  --  --
 20  --  22  --  --  --  --  --  --  --
 --  --  --  --  34  --  --  --  --  39
 --  --  --  --  44  --  --  --  --  --
 --  --  52  --  --  --  --  57  --  --
 --  --  62  --  --  65  --  67  --  --
 --  --  72  --  --  --  --  --  --  --
 --  --  --  --  --  85  --  --  --  --
 --  --  --  --  --  95  --  --  --  --
 --  --  --  --  --  --  --  --  -- 109
 --  --  --  --  --  --  --  --  -- 119
 --  --  --  -- 124  --  -- 127  --
```

There's no apparent periodicity, but there are suggestive patterns. Losing positions tend to end in 0, 2, 4, 5, 7, or 9, and the other digits seem very uncommon. More details on this "Subtract a square" version of Nim can be found at `https://en.wikipedia.org/wiki/Subtract_a_square`.

5.8.2 Wythoff's Nim. There's an old Chinese game called *tsyan-shidzi*, meaning "choosing stones", with a beautiful mathematical theory. The game was reinvented by the Dutch mathematician W. A. Wythoff, who published an analysis of it in 1907, and so it's now commonly known as "Wythoff's Nim".

Here are the rules: on a table there are two piles of chips, and two players who take turns. A move consists of taking any number of chips from one pile *or* the same number of chips from *both* piles (which isn't allowed in standard Nim). The winner is the one who takes the last chip.

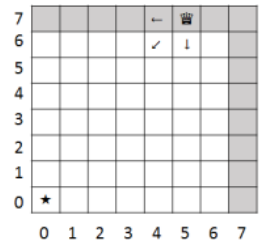


Figure 5.23. The "Corner the Lady" game

We can translate the game into the board game shown in Fig. 5.23, invented in the early 1960s by Rufus P. Isaacs, a mathematician at Johns Hopkins University. Martin Gardner named it: "Corner the Lady" in an informative article about Wythoff's Nim in *Scientific American* [**Gar97**].

Player A puts a queen on any cell in the top row or in the column farthest to the right; the relevant cells appear in gray in Fig. 5.23. The queen moves in the usual way but only west, south, or southwest. Player B moves first, then the players alternate. The player who gets the queen to the starred cell at the lower left corner is the winner.

The isomorphism between the Queen-Cornering game and Wythoff's Nim is fairly apparent. In Fig. 5.24a, we number the columns and rows along the x and y axes, so that each cell can be assigned an (x, y) pair. These numbers correspond to the number of chips in piles x and y in Wythoff's Nim. When the queen moves west, pile x is diminished. When the queen moves south, pile y is reduced. When it moves diagonally southwest, both piles shrink by the same amount. Moving the queen to cell (0,0) is equivalent to reducing both piles to 0.

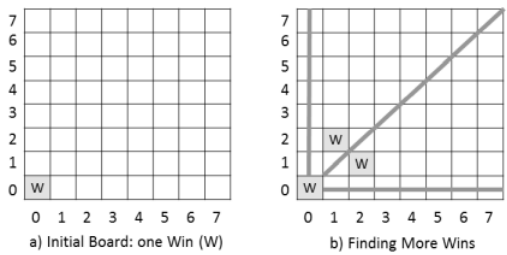


Figure 5.24. Winning the "Corner the Lady" game

In the following analysis, one of the authors will assume the role of player A, and (0,0) is a winning position (W) for me if on my turn I can move to that square.

My possible starting positions necessary for this winning move are marked with lines in Fig. 5.24b. I want Player B to land in one of these marked squares, so how do I ensure that? I must end my previous move in one of the two W squares

in Fig. 5.24b, so that Player B starts from one of them. He can only move to a marked square, from which I can win. Note that these W squares are (1, 2) and (2, 1).

In Fig. 5.25, I extend this reasoning two more stages backwards.

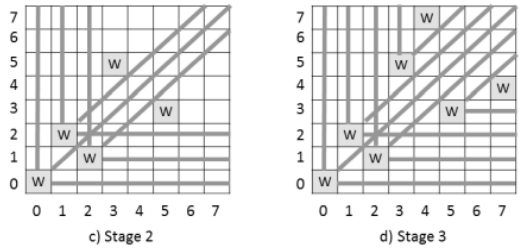


Figure 5.25. Winning "Corner the Lady" (stages 2 and 3)

Fig. 5.25c uses lines to mark all the ways I can move to the winning spots at (1, 2) or (2,1). Therefore on my previous move I want Player B to start from one of the two new W squares ((3,5) and (5,3)) because then he can only move to a marked square, from where I can win.

Fig. 5.25d continues this backwards reasoning to find two more winning spots for me – (4,7) and (7,4). I should finish on one of these squares so that Player B must move to a marked square from where I can win.

I've identified seven W positions: (0,0), (1,2), (2,1), (3,5), (5,3), (4,7) and (7,4). There's an obvious symmetry about the main diagonal, so I'll focus on (0,0), (1,2), (3,5), and (4,7). Wythoff carried out a similar analysis of his game, and came up with Table 5.6.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
xs(n)	0	1	3	4	6	8	9	11	12	14	16	17	19	21	22	24	25	27	29
ys(n)	0	2	5	7	10	13	15	18	20	23	26	28	31	34	36	39	41	44	47

Table 5.6. Losing positions for player B in Wythoff's Nim

Note that Wythoff classified the Table 5.6 positions as *losing ones for Player B*, which is equivalent to my looking for winning moves for Player A.

The table suggests the following algorithm for constructing the set L of losing positions for Player B: suppose the losing positions (xs(i), ys(i)) for $i < n$ are known already. Then xs(n) is the smallest positive integer not yet part of a pair, and ys(n) = xs(n) + n. This is a lot easier to implement in code than 'marking' horizontal, vertical , and diagonal lines.

Also note that every positive integer occurs exactly once as a member of a pair.

The algorithm is translated into Listing 5.29 (wythoff.py) which lets us easily generate Table 5.6, and also plot the pairs.

```
n = int(input("n=? "))
avails = [True]*2*(n+1)
  # ints for use in pairs
xs = [0]*(n+1)
ys = [0]*(n+1)
i = 0
for j in range(1,n+1):
  if avails[j]:
    i += 1
    xs[i] = j
      # smallest not yet used
    ys[i] = j+i
      # sum of xs[i] and i
    avails[i] = False  # i, j+1  used
    avails[j+i] = False
print(" n   xs  ys")
for j in range(i+1):
  print(f"{j:2d}:  {xs[j]:2d}  {ys[j]:2d}")
```

Example output:

```
> python wythoff.py
n=? 30
 n   xs  ys
 0:   0   0
 1:   1   2
 2:   3   5
 3:   4   7
 : # more lines
16:  25  41
17:  27  44
18:  29  47
19:  30  49
```

Listing 5.29. Generating Wythoff losing moves for player B

The plot of these (x, y) pairs is shown in Fig. 5.26.

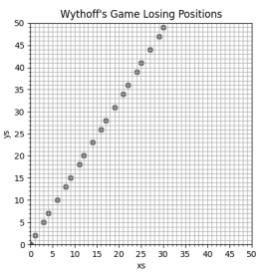


Figure 5.26. Plot of Wythoff losing moves for player B

We should also verify that L is indeed the set of losing positions for Player B.

Let W be the set of pairs not in L. It's easy to check that every move from L leads to a position in W. It remains to show that from any position in W we can move into L.

Let $(p,q) \in W, p \le q$. If $p = q$, I (player A) can move to $(0,0) \in$ L and win in one move. If $p \ne q$, let (p, p') or (p', p) be the element of L with a p value in the pair. Note that there are two cases since the p number of chips may be in either pile.

If $p' < q$, I reduce q to p'. If $q < p'$, then we know that $p < q < p'$ which is equivalent to $0 < q - p < p' - p$ (by subtracting p from all terms). To win I must reduce both piles by equal amounts, which will leave a number of chips equal to the difference $q - p$, which is in L.

5.8.2.1 The Hidden Golden Ratio. Wythoff noticed a hidden feature of Table 5.6 – the numbers in the xs sequence are multiples of the golden ratio rounded down to integers, and the ys sequence are multiples of the squared golden ratio rounded down to integers. (He later wrote that he pulled this discovery "out of a hat".)

Listing 5.30 (phis.py) can be used to confirm this.

```
phi = (math.sqrt(5)+1)/2
phi2 = phi * phi
print("n   xs  ys")
for i in range(19):
  print(f"{i:2d}  {int(i*phi):2d}  {int(i*phi2):2d}")
```

Listing 5.30. The hidden golden ratio

The hidden ϕ^2 in the ys sequence isn't hard to find since $\phi^2 = \phi + 1$, and so

$$\lfloor \phi^2 n \rfloor = \lfloor \phi n + n \rfloor = \lfloor \phi n \rfloor + n$$

which defines the ys sequence in terms of xs ($\lfloor \phi n \rfloor$).

The fact that every positive integer appears just once among the pairs can now be rephrased as all the integers that lie between successive multiples of ϕ and successive multiples of ϕ^2.

Two sequences of increasing positive integers that together contain every positive integer only once are called *complementary*.

ϕ isn't the only irrational number that generates such sequences, although it's the only one that matches the pairs of Wythoff's Nim. In 1926 Sam Beatty, a Canadian mathematician, reported that *any* positive irrational number (e.g. $\sqrt{2}, \pi, e$) can generate complementary sequences. His theorem states that if $\alpha > 1$ is irrational then the complementary sequence of $\lfloor \alpha n \rfloor$ is the sequence $\lfloor \beta n \rfloor$, such that $1/\alpha + 1/\beta = 1$.

For example, let $r = \sqrt{2} \approx 1.414$, and $s = 2 + \sqrt{2} \approx 3.414$. Note that $s = r/(r-1)$ and so $1/s + 1/r = 1$.

The sequence $\lfloor rn \rfloor$ is 1, 2, 4, 5, 7, 8, 9, 11, 12, 14, 15, 16, 18, 19, 21, 22, 24, ...

The sequence $\lfloor sn \rfloor$ is 3, 6, 10, 13, 17, 20, 23, 27, 30, 34, 37, 40, 44, 47, 51, 54, 58, ...

A comparison of these show that they are indeed complementary.

This theorem applies to the golden ratio sequences since $\phi^2 = \phi + 1$, so $1/\phi + 1/\phi^2 = 1$, and hence $\lfloor \phi n \rfloor$ and $\lfloor \phi^2 n \rfloor$ form complementary sequences.

Exercises

(1) Run Listing 5.28 for large values of n and plot the strange fluctuations of the L elements.

(2) For Wythoff's Nim:

a) Write a program which transforms a position in W into a position in L.

b) Write a program which plays against a human opponent.

c) Write a program which plays against a human opponent, based on the golden ratio formulas.

(3) In a game with a pile of n chips, the legal moves are the set $M = \{1, 2, 3, 5, ...\}$ of Fibonacci numbers. Write a program which finds the losing positions.

(4) *Bachet's game.* Initially there are n chips on the table. The set of legal moves is $M = \{1, 2, ..., k\}$. The winner is the person who removes the last chip. Find the losing positions.

(5) In Ex. 4 let $M = \{1, 2, 4, 8, ...\}$ (any power of 2). Find the set L.

(6) In Ex. 4 let $M = \{1, 2, 3, 5, 7, 11, ...\}$ (1 and all the primes). Find L.

(7) In Ex. 4 let $M = \{1, 3, 5, 8, 13\}$. Find the set L.

(8) Write a program that reads in a finite sequence M of positive integers which will act as the game's legal moves. Make predictions about the set L of losing positions.

(9) Write a recursive version of Listing 5.27, which recognizes if an input n belongs to L.

(10) Let xs(n) = $\lfloor \phi n \rfloor$, and consider the sequence as(n) = xs(n+1) - xs(n) for $n = 0$, 1, 2, ... Its first few terms are shown in Table 5.7.

1	2	1	2	2	1	2	1	2	2	1	2	2	1	2
12	122	12	122	122	12	122	12	122	122	12	122	122	12	122

Table 5.7. Terms of the as(n) sequence

The second line arises from the first by inflation: $1 \to 12, 2 \to 122$. This seems to transform the sequence into itself, which would make it self-similar.

a) Test this self-similarity as far out as possible.

b) What's the proportion of 2's among all the digits? Are you surprised? (See the solution of Ex. 3 in Sec. 3.14.]

(11) We start with 0 and repeatedly perform the substitutions $0 \to 1, 1 \to 10$ to generate 0, 1, 10, 101, 10110, 10110101, ... In this way we produce an infinite binary sequence, which is more complicated than the Morse Thue sequence.

Investigate the sequence, especially its self-similarity, proportion of 1's, etc. Compare it with Ex. 24 in Sec. 2.1, and Ex. 3 in Sec. 3.14 where the more difficult substitution $0 \to 001, 1 \to 0$ were examined.

(12) In a Nim-style game, there are 10,000 chips on the table. If it's your turn then you may remove any p^n amount of chips, where p is a prime and $n = 0, 1, 2, 3, \ldots$. The winner is the player who takes the last checker. Is that player A or B, if he plays optimally?

(13) In this question's modification of Wythoff's game, you're allowed to remove an *even* number of chips from a single pile, or equal numbers from both piles. By modifying Listing 5.29 find the set L and closed formulas for its elements. *Hint*: there will be two formulas, depending on the parity of the initial number of chips. In case of an odd initial number the final position will be (0,1).

(14) In this game start with $n = 2$ chips. The players A and B move alternately by adding to the current pile of size n a proper divisor of n. The goal is a number ≥ 1990. Who wins, A or B?

(15) In this version of Wythoff's game, you may remove any number of chips from one pile, or m chips from one pile and n chips from the other pile where $|m - n| < a$. For $a = 3$ find the losing positions and guess closed formulas for them. You've met these formulas before in Ex. 3 in Sec. 3.14.

5.9 The Subset Sum Problem (Part 2)

We're given a set of n positive reals and the task of selecting a subset which sums to a value as close as possible to a goal number, G, without exceeding it. We'll focus on a special case of this famous and notorious problem, due to R.W. Floyd, which appears as Ex. 1.4 in Hopcroft [**HUA83**]:

> Abby and Carl inherit n gold coins with weights $\sqrt{1}, \sqrt{2}, \ldots, \sqrt{n}$ kgs. Divide them into two piles whose total weights differ from each other by as little as possible.

This version of the problem is treating G as:

$$G = \frac{\sqrt{1} + \sqrt{2} + \cdots + \sqrt{n}}{2}$$

We can obtain a good, but not necessarily optimal, solution by employing a FFD (First-Fit-Decreasing) heuristic: arrange the coins into decreasing weight order, and assign the heaviest to Abby if the new total weight doesn't exceed G, otherwise give it to Carl. Repeat this until all the coins have been allocated.

Listing 5.31 (FFD.py) implements this approach, and indicates the final distribution of weights to Abby as a binary word. Its k-th digit is 1 if Abby has been assigned the coin of weight $\sqrt{k}$.

```
n = int(input("n=? "))
wghts = []
for i in range(n):
  wghts.append(math.sqrt(i+1))
goal = sum(wghts)/2
print("goal:", goal)
print("Abby's weights: ",end='')
abby = 0; carl = 0
for i in range(n-1,-1,-1):
  # large to small
  if (abby + wghts[i]) < goal:
    # assign w_i to Abby
    abby += wghts[i]
    print(1, end='')
  else:   # assign to Carl
    carl += wghts[i]
    print(0, end ='')
print()
print("Total Weights")
print(f"abby: {abby:.9f}")
print(f"carl: {carl:.9f}")
```

Listing 5.31. Implementing FFD

The result when $n = 20$:

```
> python FFD.py
n=? 20
goal: 30.832988905709904
Abby's weights:
   11111110000000001000
Total Weights
abby: 30.811421944
carl: 30.854555867
```

The FFD algorithm yields a good result (accurate to 1 d.p.) so it might even be optimal. To check, we should estimate how close we can get to an even division of the weights.

The sum of the seven largest weights is so much smaller than $G/2$ that we can add a light coin and still stay under $G/2$. Hence the share of weights for each player should consist of at least 8 coins and, by symmetry, at most 12.

There are

$$\binom{20}{8} + \binom{20}{9} + \cdots + \binom{20}{12} = 772,616$$

subsets with 8 to 12 coins, with their corresponding weights lying between 16 and 46 kgs. If the weights were uniformly distributed across this interval, each subinterval would be $\approx 30/770000 \approx 3.9 * 10^{-5}$. That's a lot smaller than the current difference between Abby's and Carl's weights, so it seems likely that our solution can be improved.

How do we find the optimal solution? We could try testing every possible subset, stepping through all 2^{20} and choosing the one with a summed weight as close to G as possible. Ideally, in each step of this search, exactly one weight should be added or removed, so the subset sum is easy to update. In terms of the binary word representation of Abby's weights, this approach corresponds to changing one bit in each step. Is this even possible, when all of the 2^{20} subsets

must be visited? The answer is yes, and is relatively easy to implement if we use *Gray codes* [**Gar20**].

5.9.1 Gray Codes. A Gray code represents a sequence of numbers using a positional notation where any adjacent pair will differ in their digits at one position only, with an absolute difference of 1. For instance, 183 and 193 could be adjacent numbers in a decimal Gray code (the middle digits differ by 1), but not 173 and 193, nor 134 and 143.

Binary Gray codes are the simplest since two successive values in a sequence only differ by one bit. They've proved so useful that they've become a common tool for error detection and recovery in digital communication and data storage.

If we limit the code to one bit, the only possible numbers are 0 and 1, which we can graph as a straight line (see Fig. 5.27a). A Gray code is obtained by moving along the line in either direction, and we've chosen to define the code as {0, 1}

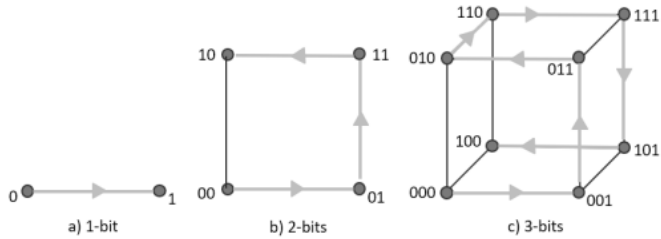


Figure 5.27. Gray codes

A Gray code with two bits is { 00, 01, 11, 10 }, which labels the corners of the square in Fig. 5.27b. Note that the binary numbers at any pair of adjacent corners differ in only one place. The path starting at 00 could be made cyclic by having it return to 00 from 10.

A 3-bit Gray code is used to label the corners of the cube in Fig. 5.27c. The path shown by the lines marks out the Gray code { 000, 001, 011, 010, 110, 111, 101, 100 }. Once again it could be made cyclic by linking from 100 back to 000.

These are called Hamiltonian paths after the Irish mathematician William Rowan Hamilton, and n-bit Gray codes can always be mapped to Hamiltonian paths on cubes of n dimensions. The 4-bit Gray code is shown mapped to a hypercube in Fig. 5.28.

For practical purposes a Gray code should have simple conversion rules for translating standard binary coded decimals to their Gray code equivalents and vice versa. The simplest is called a reflected Gray code.

Start with a 1-bit Gray code {0, 1}, duplicate it, and reverse the duplicate. Add 0s to the left of the original sequence, and 1s to the reversed duplicate, and

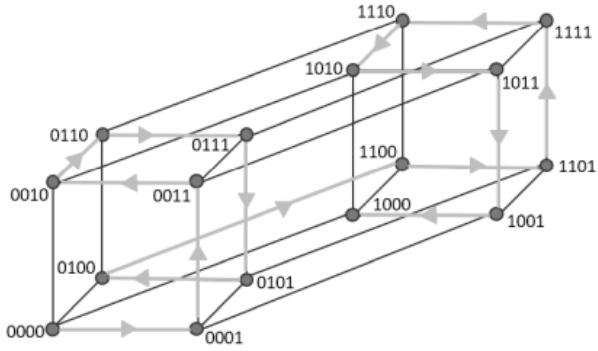


Figure 5.28. A 4-bit Gray code on a hypercube

the combination of the two sequences is a 2-bit Gray code. The single-bit change property is upheld by the reflection property, and also makes the sequence potentially cyclic. This process can be repeated, as illustrated in Fig. 5.29, to generate Gray codes with more bits.

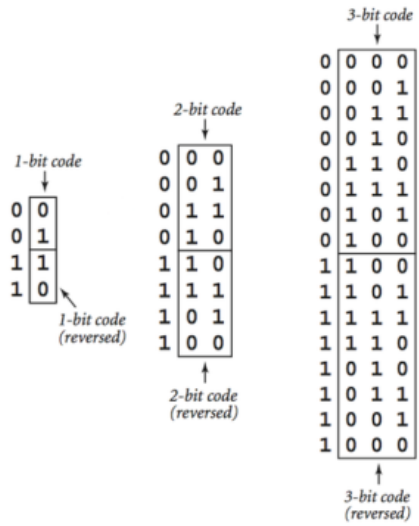


Figure 5.29. Reflected Gray codes

Listing 5.32 (grays.py) implements this approach.

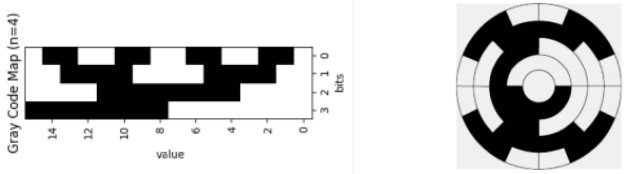
```
def grayCodes(n):
  if n <= 0:
    return ['0']
  elif n == 1:
    return ['0', '1']
  half1 = grayCodes(n-1)
  revhalf2 = reversed(half1.copy())
  return ['0'+bs for bs in half1] + \
         ['1'+bs for bs in revhalf2]
```

Listing 5.32. Generating reflected Gray codes

Examples:

```
>>> from grays import *
>>> grayCodes(3)
['000', '001', '011', '010',
 '110', '111', '101', '100']
>>> grayCodes(4)
['0000', '0001', '0011', '0010',
 '0110', '0111', '0101', '0100',
 '1100', '1101', '1111', '1110',
 '1010', '1011', '1001', '1000']
```

A useful visualization of reflected gray codes is a grid of their bit values, with black squares for 1s and white squares for 0s (see Fig. 5.30a produced by gcMap.py). This pattern (suitably rotated as in Fig. 5.30b) is utilized in some rotary position encoders as conductive strips around concentric rings.



(a) As a grid (b) As a ring

Figure 5.30. 4-bit Gray code patterns

Listing 5.33 (grays.py) converts between binary and reflected Gray codes.

```
def grayEncode(n):
  return n ^ (n >> 1) # xor

def grayDecode(n):
  mask = n >> 1
  while mask:
    n ^= mask; mask >>= 1
  return n
```

Listing 5.33. To/from a Gray code

Convert 1000_2 to the Gray code value 1100, and back again:

```
>>> from grays import *
>>> bin(grayEncode(0b1000))
'0b1100'
>>> bin(grayDecode(0b1100))
'0b1000'
```

bin() is a Python built-in for converting an integer to a binary string.

5.9.1.1 Gray Code Transition Sequences. A Gray code can be represented by its *transition sequence* – a list of the bits position changes as we move from one

Gray code value to the next. We assume that the positions are numbered from right to left, starting at 1.

The transition sequences for the 1-bit, 2-bit, and 3-bit Gray Codes in Fig. 5.27 are:

$$T_1 = 1, \quad T_2 = 1, 2, 1, \quad T_3 = 1, 2, 1, 3, 1, 2, 1.$$

The T_4 transition sequence for Fig. 5.28 is:

$$1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1$$

The underlying reflection mechanism used to create the Gray codes is apparent in their bit changing indices. The general equation for the transition sequence is

$$T_{n+1} = T_n, (n + 1), T_n \text{ reversed}.$$

We can easily prove by induction that for all n

$$T_n = T_n \text{ reversed}$$

and so the recursive definition simplifies to

$$T_1 = 1, \quad T_{n+1} = T_n, (n + 1), T_n.$$

We can generate the T_n sequence efficiently by using a stack. It initially contains the values $n, n - 1, ..., 1$ (as in Fig. 5.31).

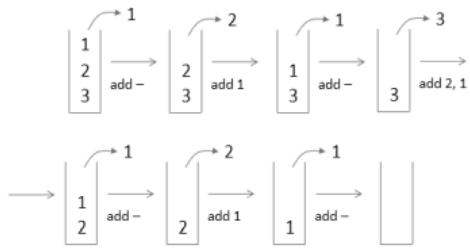


Figure 5.31. 3-bit Gray code transitions using a stack

The algorithm (see Listing 5.34; grayTransStack.py) removes the top number i and adds it to the sequence. Then the numbers $i - 1, i - 2, ..., 1$ are pushed onto the stack. This approach is from Reingold, Nievergelt and Deo [**RND77**], p. 178.

```
n = int(input("n=? "))
stk = [i for i in range(n, 0, -1)]
while stk != []:
  t = stk.pop()
  print(t, end = ' ')
  for i in range(t-1, 0, -1):
    stk.append(i)
print()
```

Listing 5.34. A stack for transitions

Examples:

```
> python grayTransStack.py
n=? 3
1 2 1 3 1 2 1
> python grayTransStack.py
n=? 4
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

T_n is also known as the *ruler* function because when its values are plotted as bars, they look very similar to the markings on a ruler (see Fig. 5.32).

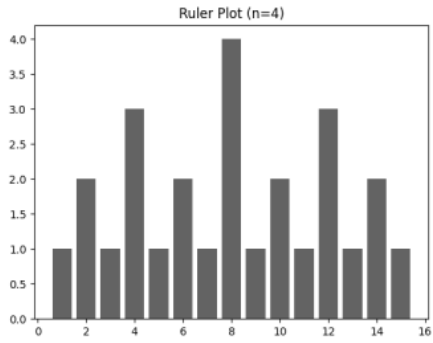


Figure 5.32. 4-bit Gray code transitions as a ruler

The code for generating Fig. 5.32 can be found in `ruler.py`.

5.9.1.2 Transition Sequences for Search. A transition sequence can be used to efficiently iterate through every possible subset of an n-element set. Consider the 3-bit Gray code:

$$000, 001, 011, 010, 110, 111, 101, 100$$

If we have a three element set, say {a,b,c}, then each Gray code number can represent a subset of that set, where 0 in the i-th position means do not include the value, and 1 means inclusion. Applying the Gray code above in this way produces the subsets:

$$\{\}, \{c\}, \{b, c\}, \{b\}, \{a, b\}, \{a, b, c\}, \{a, c\}, \{a\}$$

A key benefit of this particular ordering of the {a,b,c} powerset is that there's only one change between adjacent subsets. The change is made explicit by the corresponding transition sequence for the 3-bit Gray code $\{1, 2, 1, 3, 1, 2, 1\}$. Remember

that we're counting indices from right-to-left in transitions, and starting at 1. So the first '1' denotes the powerset change from {} to {c}. The '2' is the transition {c} to {b,c}. The next '1' means a change to index position 1, which will *remove* the *c*, leaving the subset {b}.

There are a surprisingly large number of search problems which can benefit from the use of Gray codes to efficiently traverse the search space. As an example, consider the problem of generating all the multiples of {2, 3, 5, 7}. We'll utilize a generator version of Listing 5.34 to produce a 4-bit Gray code transition sequence one value at a time. These values will guide how the program (see Listing 5.35; multAll.py) iterates through the multiples.

```
nums = [2, 3, 5, 7]
n = len(nums)
grayGen = grayTransY(n)
 # transitions for n-bit Gray codes
prod = 1
for i in range(2**n-1):
  idx = next(grayGen)
     # get a transition
  val = nums[idx-1]
  if val > 0:
    prod *= val
  else: # flag inclusion
    prod = prod//-val
  nums[idx-1] = -nums[idx-1]
  print(idx, ":", prod)
```

Listing 5.35. Generating all multiples

The output shows the transition sequence value and the multiple that it generates:

```
> python multAll.py

1 : 2          1 : 70
2 : 6          2 : 210
1 : 3          1 : 105
3 : 15         3 : 21
1 : 30         1 : 42
2 : 10         2 : 14
1 : 5          1 : 7
4 : 35
```

The only problem with using Gray code transition sequences is the double meaning of a value. We saw this issue in the set example above, where '1' initially meant include *c*, but later meant remove it. This dual meaning is dealt with in Listing 5.35 by making a number negative when it becomes part of the multiple, and positive again upon its removal.

The generator version of the stack-based transition sequence code (Listing 5.34) is in Fig. 5.36.

```
def grayTransY(n):
  # generate gray code transitions from 0 to 2^n -1
  stk = [i for i in range(n, 0, -1)]
  while stk != []:
    t = stk.pop()
    for i in range(t-1, 0, -1):
      stk.append(i)
    yield t
```

Listing 5.36. Transition sequence generator

5.9.2 Subset Sums and Gray Codes. Armed with Gray codes, we return to our collection of 20 golden coins, with weights ranging from $\sqrt{1}, \sqrt{2}, ..., \sqrt{20}$ kgs. The task is to find a subset closest to half the total weight (i.e. $G = 30.832988905709904$).

We'll employ the transition sequence for a 20-bit gray code to efficiently search through all the possible subsets of the coin weights looking for the closest match. Nowadays, 2^{20} subsets isn't a large number, but let's still try to reduce the search space.

We can immediately cut the computation time in half by checking only the first 2^{19} subsets, excluding $\sqrt{20}$. The excluded subsets are the complements of the ones we are checking, so can be ignored.

Another trick cuts the number of subsets by an additional factor of 16. We remove the set S of *integer roots* in the range 1 to 20 = $\{\sqrt{1}, \sqrt{4}, \sqrt{9}, \sqrt{16}\} = \{1, 2, 3, 4\}$. The powerset of S represents all the integers from 0 to 10, which we can more efficiently represent as an integer variable k. Now we look for a subset with weight w such that $|G - w - k|$ is as small as possible. We will eventually choose a k value that makes $|G - w - k|$ the smallest.

Instead of trying 11 different values for k, the best choice can be obtained directly by calculating round(G-w). Now the problem becomes one of finding a w such that $|G - w - \text{round}(G - w)|$ is as small as possible, provided $0 \leq$ round(G - w) ≤ 10.

Listing 5.37 (subsetSum.py) is hardwired to look for the 16 non-integer square roots in the range 2 to 19. They are generated with `math.sqrt(i+round(math.sqrt(i))`, an approach we first used in Ex. 1 of "Additional Exercises for Chapters 1 to 2".

```
N = 16  # non-squares

nis = [0]*N  # non-integer squares
print("nis[] entries")
for j in range(N):
  nis[j] = math.sqrt(j+1 + round( math.sqrt(j+1)))
grayGen = grayTransY(N)

goal = (10+sum(nis))/2
minW = 1; wghts = 0; i = 0
while i < N-1: # reduce comparisons
  i = next(grayGen)
  wghts += nis[i]
  nis[i] = -nis[i] # -ve means used in weights
  rd = round(goal - wghts)
  nWeight = abs((goal - wghts) - rd)
  if nWeight <= minW:
    if (rd >= 0) and (rd <= 10): # 0 to 10
```

```
    for j in range(N):
      print(int(nis[j] < 0),end ='')
      # 0 (not used) or 1 (used)
    minW = nWeight
    print(f"; min == {minW:.11f}")
```

Listing 5.37. Summing subsets

`minW` will shrink as it is assigned smaller values of $|G - w - \text{round}(G - w)|$. Its current value is printed as the program progresses, along with a binary word indicating which of the 16 weights have been used.

```
> python subsetSum.py
0111111110000000; min == 0.00180212361
0111110011010000; min == 0.00143429476
0001000111011000; min == 0.00113251014
0101011101111000; min == 0.00082114602
0010001011001100; min == 0.00075593530
0101110011101100; min == 0.00021332943
0100001010010110; min == 0.00003584597
```

We use the transition sequence generator, grayTransY(), retrieving the next transition in a while-loop. Once again we've also using the trick of negating a value when it becomes part of the subset, so that next time, the program knows to remove it.

The binary word output shown above requires a little deciphering. The weights $\{\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{6}, \cdots\}$ are denoted from left-to-right, with no integer square roots included. This means that the final 0100001010010110 result corresponds to:

$$\begin{aligned} 0100001010010110 &= \sqrt{3} + \sqrt{10} + \sqrt{12} + \sqrt{15} + \sqrt{18} + \sqrt{19} \\ &\approx 1.732 + 3.162 + 3.464 + 3.873 + 4.243 + 4.359 \\ &= 20.833 \end{aligned}$$

We're aiming for a number close to G, which is approximately 30.832988905709904. Our 20.833 result is not an error, but a consequence of the exclusion of the four integer square roots $(\sqrt{1}, \sqrt{4}, \sqrt{9}, \sqrt{16})$. We manually decide how many of these to add to 20.833 to get a result close to G. The best combination is to add all of them (i.e. 10), to get a final weight for Abby's coins of approximately 30.833. To be more precise, our final result is 30.8329530597.

The absolute and relative deviations from G are

$$|\text{result} - G| \approx 3.6 * 10^{-5}, \quad \frac{|\text{result} - G|}{G} \approx 10^{-6}$$

You may recall that we estimated that we could get as close as $3.9 * 10^{-5}$ to G, which turns out to be (approximately) the case.

There are actually two versions of our result. Instead of using $\{\sqrt{3}, \sqrt{10}, \sqrt{12}, \sqrt{15}, \sqrt{18}, \sqrt{19}\}$ we could replace $\sqrt{18}$ by $\sqrt{2}$ and $\sqrt{8}$. Listing 5.37

didn't report this other answer because our coding only reports reductions to `minW`; probably the other value was slightly bigger due to rounding errors and so was skipped.

Exercises

(1) Suppose k people inherit n gold pieces weighing $x_1, x_2, ..., x_n$. We want to split the inheritance into k heaps which are as nearly equal as possible.

For this case, the FFD algorithm is defined as follows: first order the items so that $x_1 \geq x_2 \geq ... \geq x_n$. Then proceed to distribute the items in order, starting with x_1, which we place in the first heap H_1. In general, if there are heaps H_j which have room for x_i, i.e. the total size of items in H_j is not more than goal$-x_i$, then we place x_i on the heap with the smallest index j. Otherwise we place it on the smallest heap; if there are several equally small heaps, we pick the one with the smallest index.

Write a program for this method and apply it to $k = 3$, $x_i = \sqrt{i}$, and $n = 20$, 30, and 40.

(2) Prove that Listing 5.34 works correctly. (Not easy.)

(3) Partition $\sqrt{1}, ..., \sqrt{25}$ into two heaps differing as little as possible.

What changes are needed in `subsetSum.py`? In this version, $G = 42.81689013753907$. Our program reports `minW` = 1.735072416E-06, for the binary 01001110111101110100.

Let's try to quantify the roundoff error in this computation. First, consider the error introduced by one rounding operation, i.e. replacing a real number x by the closest binary fraction with a k-bit mantissa.

For any integer j, there are $2^{k-1} + 1$ binary fractions with a k-bit mantissa in the interval $[2^{j-1}, 2^j]$. They divide the range into subintervals of length 2^{j-k}. Hence, if j is the integer such that $2^{j-1} < x \leq 2^j$, then there is a binary fraction with a k-bit mantissa at a distance $\leq \frac{1}{2}2^{j-k} < 2^{-k}|x|$ from it. Thus, rounding x changes it in the worst case by less than $2^{-k}|x|$.

If we take into consideration that $|x|$ will in general not be exactly in the middle of two k-bit binary fractions, and that x could be almost twice 2^{j-1}, we see that $2^{-(k+1)}|x|$ is, on average, a generous estimate of the rounding error.

In this version of Listing 5.37, we modify `minW` by additions and subtractions a total of 2^{19} times. The intention is to get a sum close to G (42.81689013753907) which we'll approximate as 40. Since floats in Python are represented with a mantissa of $k = 52$ binary digits, the average roundoff in each operation is roughly $\epsilon = 40 \times 2^{-53} \approx$ 4.4E-15.

Each roundoff error is added to the sum of the previous roundoff errors, so we can obtain a rough estimate of the probable total from the theory of random

walks. Suppose we take n steps of length ϵ, moving randomly left or right each time. One can show that the expected value of the square of the final distance from the starting point is $\epsilon^2 n$. We don't have such a simple formula for the expected value of the magnitude of the distance itself but $\epsilon\sqrt{n}$, is a reasonable guess for our purposes. If we replace ϵ by the average error in one rounding, we get that the magnitude of the error at the end of our computation is probably around $40 \times 2^{19/2} \times 2^{-53} \approx 3.2\text{E-}12$.

Our program reports `minW` = 1.735072416E-6, for the word 01001110111101110100. If we calculate `minW` ourselves, using the square roots in the binary word, plus the integer square roots, then the actual minimum is 1.7350750346E-6, so the actual roundoff error is 2.6E-12. This suggests that we can be confident, although not absolutely certain, that we have found two heap weights which differ as little as possible.

We could reduce the roundoff errors by using variables of type decimal instead of float, but this would slow down the program. Another change would be to make certain that we don't miss solutions by printing out not only the subsets which reduce `minW` but also those which come close, within some range based on our rounding error calculations.

6

Numerical Algorithms

This chapter starts with a selection of problems related to powers, logarithmic and trigonometric functions, and focuses on the numerous ways to calculate e and π. It then shifts to more of an engineering math emphasis with sections of curve fitting, numerical integration, pursuit curves, and the solution of ODEs (ordinary differential equations).

Many of the topics here are affected by the way that reals are encoded as floating point numbers, with resulting issues concerning numerical precision. For that reason, you may find it helpful to read Appendix E 'Numbers in Python' first.

6.1 Powers with Integer and Real Exponents

If y is a non-negative integer, we can find x^y recursively by using the formulae

$$x^y = \begin{cases} (x^2)^{y/2} & \text{if } y \text{ is even} \\ x\, x^{y-1} & \text{if } y \text{ is odd.} \end{cases}$$

The base of the recursion is when $y = 0$, since $x^0 = 1$.

If x, y are positive floats, we could also compute x^y recursively by means of the relations

$$x^y = \begin{cases} (\sqrt{x})^{2y} & \text{if } y < 1 \\ x\, x^{y-1} & \text{if } y \geq 1. \end{cases}$$

The bases of this recursion are the relations $1^y = 1$ and $x^0 = 1$. The condition $x \approx 1$ (up to the roundoff error) is bound to be attained if we replace x

by its square root enough times; if y happens to be an integer we get $y = 0$ before that. These recursions are implemented in Listings 6.1 (natpow.py) and 6.2 (realpow.py).

```
def pow(x, y):
  if y == 0:
    return 1
  elif (y-1)%2 == 1: # even
    return pow(x*x, y//2)
  else:
    return x*pow(x, y-1)
```

Listing 6.1. An integer power

```
def pow(x, y):
  if x == 1 or y == 0:
    return 1
  elif y < 1:
    return pow( math.sqrt(x), 2*y)
  else:
    return x*pow(x, y-1)
```

Listing 6.2. A float power

We can reformulate these computations as iterative programs. We want to find a^b for real a and integer $b, b > 0$. We introduce the variables x, y, z, and set $z = 1$, $x = a$, and $y = b$. Then

$$1 * a^b = z * x^y. \tag{6.1}$$

The idea is to drive y towards 0 while keeping Equ. (6.1) invariant. This can be accomplished by means of the two transformations

$$x = x * x, \quad y = y/2, \quad z = z \quad \text{if } y \text{ is even;}$$
$$x = x, \quad y = y - 1, \quad z = z * x \quad \text{if } y \text{ is odd.}$$

The first reduces y quickly, but will result in an integer exponent only for even y. The second reduces y slowly, but can be used for odd y. At the end $y = 0$ and $z = a^b$. The iterative function in Listing 6.3 (natpowit.py) is based on this idea.

```
def powIter(x, y):
  z = 1
  while y > 0:
    if (y%2 == 1):
      y -= 1
      z = z*x
    else:
      x = x*x
      y = y//2
  return z
```

Listing 6.3. Iterative integer power

```
def powIter(x, y):
  z = 1
  while x != 0 and y > 0:
    if y >= 1:
      y -= 1
      z = z*x
    else:
      x = math.sqrt(x)
      y = 2*y
  return z
```

Listing 6.4. Iterative float power

Suppose now that a and b are positive floats. We again keep Equ. (1) invariant but this time drive x towards 1 by square root extractions. We use the

transformations

$$x = x, \quad y = y - 1, \quad z = x * z \quad \text{if } y \geq 1;$$
$$x = \text{sqrt}(x), \quad y = 2 * y, \quad z = z \quad \text{if } y < 1.$$

When $x \approx 1$ (within roundoff errors), then $z \approx a^b$. Thus we get the iterative function in Listing 6.4 (realpowit.py).

Exercises

(1) Write a recursive function superpower() defined by

$$m{***}n = m^{m^{\cdot^{\cdot^{\cdot^{m}}}}} = m^{m{***}(n-1)}; \quad m{***}0 = 1.$$

Evaluate superpower(m,n) for some small values of m and n.

(2) a) What are the last three digits of 7^{999999}?

b) What are the last five digits of 1987^{999999}?

Hint: None of the preceding four power functions is applicable here, so design your own for this case. Write two functions, one recursive, the other iterative.

6.2 Design of a Square Root Procedure

For a given $a > 1$, we want to find $z > 0$ such that

$$z * z = a. \tag{6.2}$$

An important technique is to replace the unknown constant z by variables which are pushed towards z. So we replace z by the variables x and y such that

$$x > y \tag{6.3}$$

and

$$x * y = a. \tag{6.4}$$

These relations are easy to establish initially by setting $x = a$ and $y = 1$. Now we move toward goal (6.2) by keeping Equs. (6.3) and (6.4) invariant. That is, we must find a step which changes x and y so that the positive difference decreases, but Equs. (6.3) and (6.4) remain valid. We accomplish this by replacing (x, y) by (x', y') when we set

$$x' = \frac{x + y}{2} \quad \text{and} \quad y' = \frac{2xy}{x + y}.$$

Then

$$a = x * y = \frac{x + y}{2} * \frac{2xy}{x + y} = x' * y'$$

so that Equ. (6.4) holds for (x', y'). Moreover, the arithmetic mean of two distinct positive numbers is larger than their harmonic mean; x' is the arithmetic mean of x and y and

$$y' = \frac{2}{\frac{1}{x} + \frac{1}{y}} = \frac{2xy}{x+y}$$

is their harmonic mean. Hence Equ. (6.3) remains valid:

$$x' - y' = \frac{x+y}{2} - \frac{2xy}{x+y} = \frac{(x-y)^2}{2(x+y)} > 0.$$

From the last expression we also get

$$x' - y' = \frac{x-y}{2}\frac{x-y}{x+y} < \frac{x-y}{2}$$

So one step reduces the positive difference $x - y$ by at least the factor $\frac{1}{2}$ Do we gain just one bit per step? **No!** This is a superfast procedure. Suppose

$$0 < x - y < 2^{-n};$$

then for the next difference we have

$$0 < x' - y' < \frac{2^{-2n}}{2(x+y)} < \frac{2^{-2n}}{4y}.$$

One step about doubles the number of correct digits. Thus we get the recursive root() in Listing 6.5 (root.py).

```
def root(x,y):
  if x == y:
    return x
  else:
    return root((x+y)/2, 2*x*y/(x+y))
```

Listing 6.5. Recursive square root

In the iterative version (Listing 6.6; rootit.py) we replace the assignment $(x, y) = (x', y')$ of the recursion by two consecutive assignments $x = (x+y)/2; y = a/x$.

```
def rootit(x,y):
  if x < 0:
    raise ValueError('neg')
  while abs(x-y) > EPS:
    x = (x+y)/2
    y = a/x
  return x
```

Listing 6.6. Iterative root (two args)

```
def rootit1(a):
  if a < 0:
    raise ValueError('neg')
  y = 1; x = (a+y)/2
  while abs(x-y)/x > EPS:
    print(x)
    y = x; x = (x+a/x)/2
  return x
```

Listing 6.7. Iterative root (one arg)

We could even eliminate the variable y by using $x = 0.5 * (x + a/x)$, but it doesn't have a good stopping rule. So we reintroduce y as in Listing 6.7 (rootit1.py).

rootit1() works for all $a > 0$, not just for $a > 1$. The same is true for root() and rootit1() (see Ex. 12).

Listing 6.8 (root1.py) implements a tricky way of listing approximations to square roots. The output is for $x = 2$ and $y = 1$.

```
def root(x, y):
  print(f"{x:.10f}  {y:.10f}")
  if abs(x-y)/x < EPS:
    return (x+y)/2
  else:
    return root((x+y)/2, 2*x*y/(x+y))
```

Listing 6.8. Approximate square root

```
> python root1.py
x y? 2 1
2.0000000000  1.0000000000
1.5000000000  1.3333333333
1.4166666667  1.4117647059
1.4142156863  1.4142114385
1.4142135624  1.4142135624
1.4142135624
```

We just programmed the standard square-root algorithm used in high school, an instance of Newton's method for finding zeros of functions.

We next compute an exact formula for the relative error. Let $x = \sqrt{a}(1 + \epsilon_0)$ where $\epsilon_0 > 0$ and $x' = (x + a/x)/2 = \sqrt{a}(1 + \epsilon_1)$. By rearrangement we find that

$$\epsilon_1 = \frac{\epsilon_0^2}{2(1+\epsilon_0)} = \frac{\epsilon_0}{2}\frac{\epsilon_0}{1+\epsilon_0}.$$

We see again that when $\epsilon_0 \gg 1$, the error is halved in each step (linear convergence), and for small ϵ_0 the error is squared (quadratic convergence). For small ϵ_0 the number of correct digits approximately doubles at each step.

Does it pay to speed up quadratic convergence? We'll make two attempts.

6.2.0.1 Improvement 1. We observe that for $x \gg 1$ we have

$$\sqrt{\frac{x+1}{x-1}} \approx 1 + \frac{1}{x}.$$

This can be verified by squaring.

We restrict our attention to the range $a > 1$. We utilize this approximate formula by writing a as $(x+1)/(x-1)$, where $x = (a+1)/(a-1)$ so that $\sqrt{a}$ can be written in the form

$$\sqrt{a} = \sqrt{\frac{x+1}{x-1}} = (1+\frac{1}{x})\sqrt{\frac{y+1}{y-1}}.$$

In this case, $y = 2x^2 - 1$, so y is much larger than z. Hence we get the product representation

$$\sqrt{a} = \sqrt{\frac{x+1}{x-1}} = (1+1/q_1)(1+1/q_2)\cdots(1+1/q_n)\sqrt{\frac{q_{n+1}+1}{q_{n+1}-1}}$$

with $q_1 = z, q_{n+1} = 2q_n^2 - 1$, where the error factor

$$\sqrt{\frac{q_{n+1}+1}{q_{n+1}-1}} \approx 1 + \frac{1}{q_{n+1}}$$

converges quickly to 1. With $x = 3$ we get $a = 2$ and

$$\sqrt{2} = (1+1/3)(1+1/17)(1+1/577)(1+1/665857)\cdots,$$

It's easy to show that $\epsilon_{n+1} \approx \epsilon_n^2/2$, which is no improvement over Newton's method.

6.2.0.2 Improvement 2. A seemingly faster approximation is based on writing a as $(x+3)/(x-1)$ and use the identity

$$\sqrt{\frac{x+3}{x-1}} = (1+\frac{2}{x})\sqrt{\frac{y+3}{y-1}},$$

where $y = x^3 + 3x^2 - 3$. With $x = 5$ we get

$$\begin{aligned}
\sqrt{2} &= (1+\tfrac{2}{5})(1+\tfrac{2}{197})(1+\tfrac{2}{7761797})\cdots \\
x_1 &= 1.4 \\
x_2 &= 1.414213\underline{198} \\
x_3 &= 1.41421356237309504880...
\end{aligned}$$

The incorrect digits are underlined. Thus we have 2, 7, 21 correct digits., suggesting that each step triples the accuracy, giving us cubic convergence. It's easy to show that $\epsilon_{n+1} \approx \epsilon_n^3/4$. The curtain that hides the unknown digits is rolled back 50% faster, but unfortunately this is negated by more work at each step, so it doesn't seem to be worth the effort.

Remark. There is a simple exact formula for the error in the n-th approximant in Newton's method. We set

$$x_0 = \sqrt{a}\frac{1+w}{1-w}; \quad \text{then} \quad x_n = \sqrt{a}\frac{1+w^{2^n}}{1-w^{2^n}}.$$

In spite of the fact that the computation is somewhat shorter we prefer to set $x_0 = \sqrt{a}(1+\epsilon_0)$, because the interpretation of w is not as natural as that of ϵ_0. Indeed

$$w = \frac{x_0 - \sqrt{a}}{x_0 + \sqrt{a}}.$$

Exercises

The next ten exercises investigate the quadratic equation by putting it into the form $x = f(x)$ and iterating. This method of solving an equation numerically is discussed in some calculus textbooks, such as Strang [**Str91**], which is freely downloadable from `https://ocw.mit.edu/courses/res-18-001-calculus-fall-2023/`.

(1) We want to solve $x^2 - 4x - 1 = 0$ by iteration. So we "solve" for x and get $x^2 = 4x + 1$ or $x = 4 + 1/x$ and so the recurrence

$$x_1 = 4, \quad x_{n+1} = 4 + \frac{1}{x_n}.$$

Show that x_n converges to a root of $x^2 - 4x - 1 = 0$ and that at each iteration the absolute error becomes about 18 times smaller.

(2) We want to solve $x^2 = 10$ iteratively. Now $x = 10/x$ leads nowhere since the sequence of iterations of $f(x) = 10/x$ has period 2. So we set $x = z - 3$ and get $z = 6 + 1/z$. Show that the sequence $z_1 = 6, z_{n+1} = 6 + 1/z_n$ converges to a root of $z^2 - 6z - 1 = 0$ with linear rate $q \approx 1/38$. That is, we get on average about 1.6 additional decimals per iteration.

(3) Take the quadratic $x^2 - 2x + 1 = 0$ (with just one root), which we transform into $x = 2 - 1/x$ and $x_1 = 2, x_{n+1} = 2 - 1/x_n$. Show by induction that $x_n = 1 + 1/n$, slowly converging to the fixed point $s = 1$.

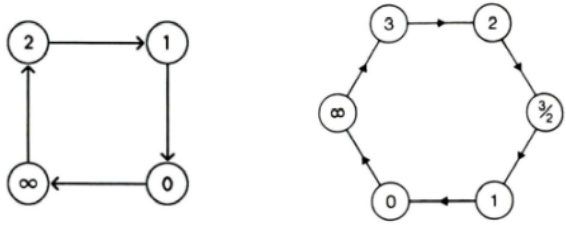
(4) For studying the iterates of rational functions, we adopt the definitions

$$\frac{\text{nonzero}}{0} = \infty, \quad \frac{\text{nonzero}}{\infty} = 0;$$

This eliminates the need for restrictions on the domains of definition of the functions.

Take a quadratic without real solutions: $x^2 - 2x + 2 = 0$, or $x = 2 - 2/x$. The sequence $x_1 = 2, x_{n+1} = 2 - 2/x_n$ has a period of 4 (Fig. 6.1a).

The quadratic $x^2 - 3x + 3 = 0$, or $x = 3 - 3/x$ leads to the sequence $x_1 - 3, x_{n+1} = 3 - 3/x_n$ with a period of 6 (Fig. 6.1b). The equation $x^2 - x + 2 = 0$ or $x = 1 - 2/x$ results in the sequence $x_1 = 1, x_{n+1} = 1 - 2/x_n$. The output suggests that the sequence doesn't converge at all. It seems that the points of the sequence are everywhere dense on the line, i.e. the sequence visits every



(a) Sequence of period 4

(b) Sequence of period 6

Figure 6.1. Quadratic sequences

interval infinitely often. Since computer arithmetic is finite we will ultimately observe an overflow or cycling behavior.

(5) We formulate some questions about the quadratic $x^2 - px + q = 0$ and the associated sequence

$$x_1 = p, \quad x_{n+1} = p - \frac{q}{x_n} \tag{6.5}$$

a) When does the Equ. (6.5) converge?

b) To which root of $x^2 - px + q = 0$ does it converge if it converges at all?

c) How fast is the convergence rate?

d) How can we speed up convergence?

(6) Let r and s be the roots of $x^2 - px + q = 0$ and let x_1, x_2 be the the first two terms of Equ. (6.5). Set $x_1 = r(1 + \epsilon_1), x_2 = r(1 + \epsilon_2)$, where ϵ_1 and ϵ_2 are the relative deviations of x_1 and x_2 from r.

a) Show that

$$\epsilon_1 = \frac{s}{r}, \quad \frac{1}{\epsilon_2} = \frac{r}{s} + \frac{r}{s}\frac{1}{\epsilon_1}, ..., \quad \frac{1}{\epsilon_n} = \frac{r}{s} + \frac{r^2}{s^2} + \cdots + \frac{r^n}{s^n}.$$

b) What do you get for ϵ_n if $r = s$?

c) What do you get for ϵ_n for $r \neq s$?

d) Show that if $|r| > |s|$ we have, for large n, $\epsilon_n \approx (1 - s/r)(s/r)^n$. Thus x_n converges to the root with larger absolute value with constant rate s/r.

e) What happens if $|r| = |s|$, but $r \neq s$?

f) What happens if r/s is an n-th root of unity, i.e. $(r/s)^n = 1$ and $(r/s)^m \neq 1$ for $m < n$?

(7) We aren't satisfied with linear convergence, so transform $x^2 - px + q = 0$ into $(x - p/2)^2 = p^2/4 - q$. Show that Newton's method $x_{n+1} = \frac{1}{2}(x_n + a/x_n)$ leads to

$$x^{n+1} = \frac{x_n^2 - q}{2x_n - p}$$

with quadratic convergence for $p^2 \neq 4q$ and linear convergence for $p^2 = 4q$.

(8) Show that, for $r \neq s$ and any starting approximant x_0,

$$x_0 = \frac{s(x_0 - r) - r(x_0 - s)}{(x_0 - r) - (x_0 - s)}, \quad x_1 = \frac{s(x_0 - r)^2 - r(x_0 - s)^2}{(x_0 - r)^2 - (x_0 - s)^2}$$
$$x_n = \frac{s(x_0 - r)^{2^n} - r(x_0 - s)^{2^n}}{(x_0 - r)^{2^n} - (x_0 - s)^{2^n}} \tag{6.6}$$

(9) Deduce from Equ. (6.6) the following:

Theorem. Let $x^2 - px + q = 0$, where p and q may be complex numbers, and have roots r and s. Let

$$g(x) = \frac{x^2 - q}{2x - p}.$$

We generate the sequence $x_0, x_1 = g(x_0), x_2 = g(x_1), \ldots$.

If $p^2 = 4q$ i.e. $r = s$, then x_n converges linearly to $x = p/2$.

If $p^2 \neq 4q$, i.e. $r \neq s$ and x_0 is closer to r than to s then x_n converges quadratically to r. If the location of x_0 in the complex number plane is on the perpendicular bisector of r and s, then there is no convergence. The sequence lies on the perpendicular bisector. It is periodic only for some initial values x_0.

(10) *Quadratic equations and matrices.* Let $x^2 - px + q = 0$, $x_1 = p$, $x_{n+1} = p - q/x_n$ with linear convergence to the root with largest absolute value. Then

$$x_1 = \frac{u_1}{v_1} = \frac{p}{1}, \cdots, \quad x_{n+1} = \frac{u_{n+1}}{v_{n+1}} = \frac{pu_n - qv_n}{u_n}.$$

Consider the matrix

$$Q = \begin{bmatrix} p & -q \\ 1 & 0 \end{bmatrix}$$

and set

$$\begin{bmatrix} u_0 \\ v_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} u_1 \\ v_1 \end{bmatrix} = \begin{bmatrix} p \\ 1 \end{bmatrix}, \cdots, \quad \begin{bmatrix} u_{n+1} \\ v_{n+1} \end{bmatrix} = \begin{bmatrix} pu_n - qv_n \\ u_n \end{bmatrix}.$$

Then $\begin{bmatrix} u_n \\ v_n \end{bmatrix} = \begin{bmatrix} p & -q \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. So the first column of Q^n is $\begin{bmatrix} u_n \\ v_n \end{bmatrix}$ and the second is $\begin{bmatrix} -qu_{n-1} \\ -qv_{n-1} \end{bmatrix}$.

To get large powers of Q rapidly, we square Q repeatedly. Show that we arrive at Newton's method, i.e. $x_{2n} = (x_n^2 - q)/(2x_n - p)$. Apply the result to $x^2 - 4x - 1 = 0$ with matrix $Q = \begin{bmatrix} 4 & 1 \\ 1 & 0 \end{bmatrix}$.

(11) In the cycle-finding algorithm of Listing 3.24, replace the function f by

```
def f(u):
  return 1-2/u
```

In addition, throw away the global variables a and m. If you start with $x = 1$, then you will trace the nonperiodic sequence $x_1 = 1, x_{n+1} = 1 - 2/x_n$. Run the program until you detect a cycle, which will occur if there is no overflow or underflow.

(12) Why do the programs in Listings 6.5, 6.6 and 6.7 work for all $a > 0$, not just $a > 1$?

(13) (Hendrik Lenstra). Consider the sequence

$$x_0 = 1, \quad x_n = (1 + x_0^2 + x_1^2 + \cdots + x_{n-1}^2)/n \text{ for } n = 1, 2, 3...$$

This sequence is also generated by the formulae

$$x_0 = 1, \quad x_1 = 2, \quad x_n = \frac{x_{n-1}(x_{n-1} + n - 1)}{n} \text{ for } n > 1.$$

We would like to know if all the terms x_n are integers.

a) Find $x_0, x_1, ..., x_9$. They are integers!

b) How do you find out if the 12-digit number x_9 is indeed an integer?

c) We computed $x_0, .., x_{19}$ one by one, and they were integers. x_{19} had several thousand digits, and the number roughly doubles at each step.

In fact the sequence doesn't consist of integers only. Can you think of a way to find a term which isn't an integer?

d) It turns out that x_n is an integer for $n \leq 42$, but not for $n = 43$. How could you prove this?

(14) (Boyd and van der Poorten). Consider the sequence

$$x_0 = 1, \quad x_n = \frac{1}{n}(1 + x_0^3 + x_1^3 + \cdots + x_{n-1}^3), \quad n = 1, 2, 3, ...$$

It seems to be even better than the sequence in Ex. 13; $x_2, x_3, ., x_{88}$ seem to be integers. Investigate as in Ex. 13.

(15) *Superfast reciprocation.* Given $a > 0$; find z such that

$$az = 1. \tag{6.7}$$

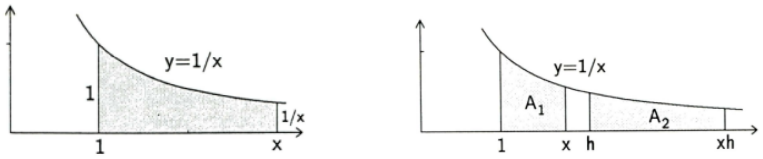
Here we replace the unknown constant z by a variable x. We try to drive the error y to 0 in the equation

$$ax = 1 - y \tag{6.8}$$

We assume that we can find an initial value x such that the y in Equ. (6.8) satisfies $|y| < 1$. If we multiply x by $1 + y$, the error y in Equ. (6.8) is replaced by y^2. Thus we get quadratic convergence. Write code based on this idea. (This trick can also be used for fast reciprocation of power series. In each step the number of correct coefficients doubles. It's akin to Newton's method for power series.)

6.3 The Natural Logarithm

The natural logarithm $\ln x$ is defined for $x > 0$ as the area under the hyperbola $xy = 1$ from 1 to x (Fig. 6.2a).



(a) Unit area (b) Stretched area

Figure 6.2. The area under a hyperbola

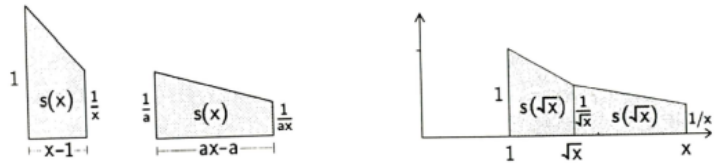
If we stretch the hyperbola $y = 1/x$ horizontally by a factor h and compress it vertically by the same factor, we get back to the curve $xy = 1$ since the product of the two coordinates of any point is unchanged. In Fig. 6.2b we see a region A_1 and the region A_2 resulting from it by stretching horizontally and compressing vertically by the factor h. The two regions have the same area because the compression cancels the effect of the stretching. In terms of our notation this means

$$\ln x = \ln xh - \ln h \quad \text{or} \quad \ln xh = \ln x + \ln h.$$

This property of the hyperbola's area function, which we now regard as the basic property of a logarithm, was actually noticed decades before logarithms were invented.

We want to construct an algorithm which efficiently computes $\ln x$. The trapezoids in Fig. 6.3a have the same area $s(x) = (x - 1/x)/2$. If we apply this to

the trapezoids in Fig. 6.3b we find that each has the area $s(\sqrt{x})$ and their sum is $2s(\sqrt{x})$.



(a) Same area trapezoids (b) Summing areas

Figure 6.3. Trapezoids for computing $\ln x$

Starting with the first approximation $s(x)$ to $\ln x$ we repeatedly replace x by $\sqrt{x}$ and multiply by 2. After n iterations we approximate $\ln x$ by 2^n trapezoids of equal area, which is implemented in Listing 6.9 (cancel.py).

```
def s(x):
  return (x-1/x)

x,n = map(int, input("x n? ").split())
p = 0.5
for i in range(n):
  x = math.sqrt(x)
  p = 2*p
  print(f"{p*s(x):.11f}  ", end = '')
  if (i+1)%4 == 0:
    print()
```

Listing 6.9. Approximating a logarithm

Approximating log(2):

```
> python cancel.py
x n? 2 39
0.70710678119  0.69662139950  0.69401475784  0.69336401383  0.69320138506
0.69316073145  0.69315056827  0.69314802749  0.69314739229  0.69314723349
0.69314719379  0.69314718387  0.69314718139  0.69314718077  0.69314718061
0.69314718058  0.69314718057  0.69314718057  0.69314718063  0.69314718060
0.69314718060  0.69314718107  0.69314718200  0.69314718340  0.69314718246
0.69314717874  0.69314718246  0.69314718246  0.69314718246  0.69314718246
0.69314718246  0.69314670563  0.69314575195  0.69314575195  0.69314575195
0.69314575195  0.69314575195  0.69311523438  0.69311523438
```

Alas, this naive approach leads to catastrophic cancellation. We ran the program for $x = 2$ and $n = 39$ and after 39 steps arrived at 0.69311523438 instead of math.log(2) = 0.69314718056 (to 11 dp). In $x - 1/x$ both terms approach 1 if we repeatedly replace x by $\sqrt{x}$. The roundoff error in the individual terms becomes a

larger and larger fraction of the difference, so the number of correct digits begins to decrease after a while.

In the 17th output above, the error was less than 2 units in the last digit (0.69314718057). At this stage we are approximating the area with 2^{17} trapezoids. We have taken square roots 17 times to compute the area of one trapezoid and then multiplied by 2^{16}. This last step magnifies the roundoff error by 2^{16} but since Python offers 15-digit accuracy for floats, the effect is still quite small. So this method can give reasonable results if used judiciously. However, we can do much better by rewriting our formulae to avoid the subtraction of nearly equal quantities.

We introduce the function $c(x) = (x + 1/x)/2$. Then

$$c(x) = 2(c(\sqrt{x}))^2 - 1, \quad s(x) = 2s(\sqrt{x})\, c(\sqrt{x}).$$

We use the letters c and s to emphasize the similarity to familiar trigonometric formulae, cosine and sine. Solving these formulae for the functions of $\sqrt{x}$ gives

$$c(\sqrt{x}) = \sqrt{\frac{1 + c(x)}{2}}, \quad 2s(\sqrt{x}) = \frac{s(x)}{c(\sqrt{x})}. \tag{6.9}$$

If we use this formula, the transition from $s(x)$ to $2s(\sqrt{x})$ is accomplished by dividing by the factor $c(x)$ which approaches 1, and we stop the computation when $c(x)$ is within some ϵ of 1, as shown in Listing 6.10 (lniter.py). The call logApprox(2) returns 0.69314718057, which is correct except for the last digit, which should be 6.

```
def logApprox(x):
  s = (x-1/x)/2
  c = (x+1/x)/2
  while abs(c-1) > EPS:
    c = math.sqrt((1+c)/2)
    s = s/c
  return s
```

Listing 6.10. A better log approx

6.4 The Inverse Trigonometric Functions

Given the coordinates (c, s) of the point C on the unit circle in Fig. 6.4, we want to compute the angle $a = \sphericalangle BOC$ without using trigonometric functions.

An angle can be measured in degrees by assigning 180° to the half turn, which is somewhat arbitrary and comes from Old Babylonian astronomy. Many computations in mathematics and science are simpler if we use the *radian*, which is the length of the arc intercepted by the angle on the unit circle. Thus, 180° is equal to π radians.

We shall find the radian measure of the angle

$$a = \arcsin s = \arccos c = \arctan \frac{s}{c} = \operatorname{arccot} \frac{c}{s}$$

by computing the length of the arc BC. Into the arc AC we inscribe $1, 2, 4, 8, \ldots$ equal chords. Half of the sum of their lengths will converge to the length of the arc BC. We compute the formulae for one doubling and apply them as many times as necessary to get an accurate answer.

We start with the chord $AC = 2s$ as our first approximation to the arc. On duplication we get two chords, each a distance c' from O and of length $BC = 2u'$. We need to express c' and u' in terms of c and s. The reason we denote MB by u' and not s' is that we wish to use s' to denote the length of the next polygonal approximation to the length of the arc BC, so that s' will denote $2u'$.

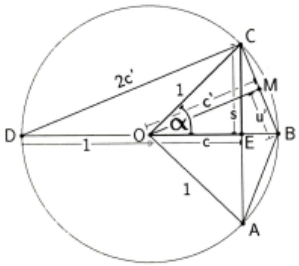


Figure 6.4. Computing α without trigonometry

Since O and M are the midpoints of BD and BC, DC is parallel to OM and twice as long. Thus $CD = 2c'$. $\triangle CDE \sim \triangle BOM$, and so $2c'/(1+c) = 1/c'$ or

$$c' = \sqrt{\frac{1+c}{2}}. \tag{6.10}$$

To find a relation between s and u' we express twice the area of the right triangle BCD in two ways and we get the equation $2c'2u' = 2s$ and hence

$$2u' = s' = \frac{s}{c'}. \tag{6.11}$$

Our formula (6.10) tells us that when we double the number of sides in our polygon, the distance of the sides from the center changes from c to $c' = \sqrt{(1+c)/2}$ and the length of the polygon is multiplied by $1/c'$. So to compute $\arccos c$, we start with c and $s = \sqrt{1-c^2}$. Then we compute c' from Equ. (6.10) and $s' = s/c'$ from Equ. (6.11) repeatedly. The quantity c' approaches 1 and we stop when it is within ϵ of 1.

Listing 6.11 (arcit.py) implements the algorithm; the function returns when c differs from 1 by at most ϵ.
Once the values of $1 - c$ are small, they are multiplied by a factor $\approx 1/4$ as can easily be seen by computing $(1-c')/(1-c)$.

```
def arccosApprox(c):
  s = math.sqrt(1-c*c)
  while abs(1-c) > EPS:
    c = math.sqrt((1+c)/2)
    s = s/c
  return s
```
Listing 6.11. Approximate arccosine

For input $c = 0$ we get $s = \pi/2$. As a check we also print $2s$, which is π for this input.

The idea behind this section is due to Archimedes, the greatest mathematician of antiquity, who used it to estimate π using the circumference of a circle (see

Sec. 6.7.6). The inscribed polygons give lower bounds for n, and he also developed a method for computing circumscribed regular polygons, to furnish upper bounds for n. We adapted his approach to arcs and left out the circumscribed polygons, but see the exercise at the end of this section.

We hope the reader is pleased and surprised by this simple and rapid way of computing the inverse trigonometric functions. What is even more surprising is that the formulae (6.10) and (6.11) for computing $\arccos c$ are the same as the formulae (6.9) in Sec. 6.3 which we used to compute $\ln x$. The only difference is that the starting values of c and s in that computation are calculated in a different way; in particular, there they satisfy the relation $c^2 - s^2 = 1$ whereas in the trigonometric computation we have $c^2 + s^2 = 1$. If we examined further the implications of the fact that the same algorithm yields both the natural logarithm and the inverse cosine, we would arrive at Euler's formula $e^{i\alpha} = \cos\alpha + i\sin\alpha$ which is usually obtained by looking at the Taylor series of these functions.

Exercise

We can add circumscribed polygons to our algorithm without much additional computation. In Fig. 6.4 we draw the tangent at B. Let F be its intersection with OC. Let $t = BF$. Then $(t : 1) = (s : c)$, or $t = s/c$. Write a program which computes both lower and upper bounds for α using inscribed and circumscribed polygons.

6.5 The Function exp()

Let exp() denote the inverse of ln(), i.e. $x = \exp(y) \Leftrightarrow y = \ln(x)$. We rewrite the functional equation $\ln xh = \ln x + \ln h$ of Sec. 6.3 in terms of exp(). Let $u = \ln x$, and $v = \ln h$, then apply exp() to both sides of our equation to get

$$\exp(u + v) = \exp u \cdot \exp v. \tag{6.12}$$

In particular, we get the duplication formula

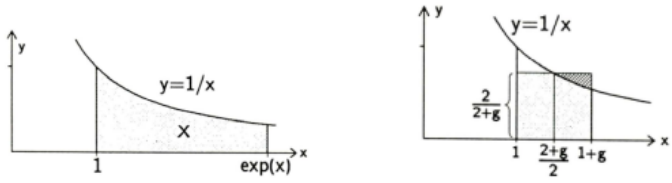
$$\exp 2x = (\exp x)^2. \tag{6.13}$$

If we label the x-coordinates for 1 and $exp(x) = g(x)+1$, then the area below the hyperbolic curve between them is x by definition (see Fig. 6.5a).

We see from Fig. 6.5b that when x and hence $g(x)$ are small, the area under the curve is nicely approximated by the "midpoint rule":

$$\frac{2g(x)}{2 + g(x)} \approx x \quad \Rightarrow \quad g(x) \approx \frac{2x}{2 - x}. \tag{6.14}$$

The relative error in $g(x)$ is as small as we like if x is sufficiently small. We will say more about this error in Sec. 6.11.



(a) exp() and hyperbola area

(b) exp() and the midpoint rule

Figure 6.5. Calculating exp(x) using the hyperbola

We go from small values of x to larger ones by means of the duplication formula for $g(x)$ in Equ. (6.13):

$$g(2x) = g(x)(2 + g(x)). \tag{6.15}$$

Note that if we know $g(x)$ with a certain accuracy and $|g(x)|$ is small, then the relative error in $2 + g(x)$ is much smaller than the relative error in $g(x)$. So if we compute $g(2x)$ using Equ. (6.15), the relative error is only a little greater than the relative error in $g(x)$.

This means that we can divide x by a power of 2 to make the quotient so small that the relative error in Equ. (6.14) is very small. Then we get back to x by means of the duplication formula (6.15) without increasing the relative error too much.

Listing 6.12 (expo.py) implements this idea, n is the number of times we want to divide x by 2 before applying Equ. (6.14). With $n = 16$, $x = 1$ we get $1 + g = 2.71828l8285$. All 11 digits are correct.

```
def expApprox(x,n):
  x = x/(2**n)
  g = 2*x/(2-x)
  for i in range(n):
    g = g*(2+g)
  return g
```

Listing 6.12. Approximate exponential

6.6 The Cosine

Here the situation is similar to that for exp() – if we try to compute cos(x) by means of an obvious algorithm and are punished by severe cancellation errors. We first try a naive approach, then investigate two improvements.

a) If we know $c(x) = \cos x$ for the arc x, then we can easily find $c'(x) = c(2x)$ for the arc $2x$. Indeed, in Fig. 6.6a, $(1 + c')/2c = 2c/2$, so $4c^2 = 2(1 + c') \Rightarrow c' =$

$2c^2 - 1$; thus

$$c(2x) = 2c^2(x) - 1 \quad \text{(duplication formula for } c(x) = \cos x\text{)}. \tag{6.16}$$



(a) $c' = 2c^2 - 1$ (b) $(1 - c) \sim \frac{1}{2}x^2$

Figure 6.6. Calculating cosine ($c(x) = \cos(x)$)

b) For sufficiently small x we know $c(x)$ with any precision we want. Indeed, in Fig. 6.6b, $(1 - c)/AB = AB/2$ or $1 - c = \frac{1}{2}AB^2$. For small x, the percentage difference between the arc x and the line segment AB is very small, hence

$$(1 - c) \sim \frac{1}{2}x^2 \quad \Rightarrow \quad c(x) \approx 1 - \frac{1}{2}x^2. \tag{6.17}$$

To compute $c(x)$ we find the cosine of $a/2^n$ by means of Equ. (6.17) and apply the duplication formula (6.16) n times.

It turns out that errors grow rapidly in this calculation, and it's not difficult to find the reason. Suppose that at some stage the quantity we have computed is $c + \epsilon$ where c is the correct value. One more application of the duplication formula gives

$$\begin{aligned} c' + \epsilon' &= 2(c + \epsilon)^2 - 1 + \text{new roundoff error} \\ &= 2c^2 - 1 + 4c\epsilon + 2\epsilon^2 + \text{new roundoff error} \end{aligned}$$

Since c is close to 1 through out most of the computation, the error is multiplied by about 4 at each step. Nevertheless, the method is usable, because when x is small, the error in Equ. (6.17) is only about $-x^4/24$. (This error estimate follows from Taylor's formula.)

We need to calculate cosines only for acute angles, so $(s - 1)/4$ divisions of our original x by 2 will mean that Equ. (6.17) produces a cosine with an accuracy of about s binary digits. Since we apply the duplication formula $(s - 1)/4$ times, the total loss will be about $(s-1)/2$ bits. We should expect the final result to have about $s/2$ correct bits.

Since $c(x)$ is close to 1 through most of the computation, the difference $f(x) = c(x) - 1$ is much smaller and can therefore be represented with a much smaller error in floating point arithmetic. We get from Equ. (6.16):

$$f(2x) = 2f(x)(2 + f(x)). \tag{6.18}$$

We can see in the same way as before that the error increases by a factor of about 4 each time we apply Equ. (6.18), but this time the relative error does not change much while $f(x)$ is small, so there's no great loss of significant digits.

How many times should we divide x by 2 before we apply Equ. (6.17) and start the duplications? In terms of $f(x)$ the Taylor remainder formula we mentioned above is

$$\frac{f(x)}{\frac{1}{2}x^2} \approx 1 - \frac{x^2}{12} \text{ for small } |x|. \tag{6.19}$$

If we are using a mantissa of s bits, the relative error due to just one rounding can be about 2^{-s}. By Equ. (6.19) the relative error we introduce when we approximate $f(x)$ by $-\frac{1}{2}x^2$ is about $-\frac{1}{12}x^2$. This will be no more than a single roundoff error if $\frac{1}{12}x^2 < 2^{-s}$. So no further gain of accuracy can be expected if we continue halving after we have $|x| < \sqrt{12/2^s}$.

Listing 6.13 (cos.py) implements the algorithm.

```
def cosApprox(x):
  e = math.sqrt(12*EPS)
  p = 0
  while abs(x) > e:
    x = x/2
    p += 1
  f = -x*x/2
  for _ in range(p):
    f = 2*f*(2+f)
  return f
```

Listing 6.13. Approx cosine

6.7 One Million Decimals of e and π

The main programming issue when calculating a real number possessing a large number of digits is how to deal with the limited precision of the real data type. In Python, a real is usually represented as a float, which only supports at most 15 decimals of precision; this is clearly inadequate.

We'll look at two ways to generate e. The first uses a limited form of infinite precision arithmetic by encoding numbers as lists of digits. The second approach is to write an exponential function using Python's decimal type, which has built-in support for infinite precision.

When generating a million digits of π, we'll limit ourselves to decimal implementations, but that still offers many ways of calculating the number.

6.7.1 An Infinite Precision Exponential using Lists. The power series for the exponential function:

$$\exp x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots \tag{6.20}$$

However, the code in this section is more closely related to the nested series:

$$\exp x = 1 + \frac{x}{1}(1 + \frac{x}{2}(1 + \frac{x}{3}(1 + \cdots)))$$

For a particular calculation, this series is truncated after the n-th term:

$$\exp x = 1 + \frac{x}{1}(1 + \frac{x}{2}(1 + \frac{x}{3}(1 + \cdots \frac{x}{n-1}(1 + \frac{x}{n})\cdots)))$$

Iterative code builds the result for exp(1) by starting from the n-th term and working outwards, including the $(n-1)$-th part, then the $(n-2)$-th, and so on. For example, if $n = 450$, then suitable pseudo-code is:

```
n = 450; e = 1
while n > 0:
  e = 1 + e/n
  n -= 1
```

A difficult question is deciding what the n-th term should actually be in order to generate a specific precision, such as 1000 decimals. If we truncate the series for e after the n-th term, we get

$$e_n = 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}$$

with an error

$$\begin{aligned}
\mathrm{err}_n &= e - e_n = \tfrac{1}{(n+1)!} + \tfrac{1}{(n+2)!} + \cdots \\
&= \tfrac{1}{(n+1)!}(1 + \tfrac{1}{n+2} + \tfrac{1}{(n+2)(n+3)} + \cdots) \\
&< \tfrac{1}{(n+1)!}(1 + \tfrac{1}{n+2} + \tfrac{1}{(n+2)^2} + \cdots) \\
&= \tfrac{1}{(n+1)!}\tfrac{1}{1-\frac{1}{n+2}} \\
&= \tfrac{1}{(n+1)!}\tfrac{n+2}{n+1}
\end{aligned}$$

Since $\mathrm{err}_{450} < 452/(451!\,451) \approx 1.3 \times 10^{-1003}$, 450 terms are sufficient for an accuracy of 1000 decimals. In general, we stop at the term which makes the error smaller than the d decimal digits present in the result. In other words,

$$\begin{aligned}
\tfrac{1}{(n+1)!} &\approx 10^{-d} \\
(n+1)! &\approx 10^{d} \\
\log_{10}(n+1)! &\approx d
\end{aligned}$$

This can be calculated in Python:

```
>>> import math
>>> math.log10(math.factorial(450))
1000.2388909583989
>>> math.log10(math.factorial(3249))
10000.807016385219
>>> math.log10(math.factorial(25206))
100000.08124049933
```

However, it's a little tedious trying out different values of n to find a suitable d. It's easier to write a function:

```
def findFact(d):
  f = 1; n = 2
  while math.log10(f) < d:
    f = f*n
    n += 1
  print("log10(fact("+ str(n-1) +
      ")) >=", d)
```

```
>>> from decTrig import *
>>> findFact(1000)
log10(fact(450)) >= 1000
>>> findFact(10000)
log10(fact(3249)) >= 10000
>>> findFact(100000)
log10(fact(25206)) >= 100000
```

Listing 6.14. Factorial with d digits

The first result tells us that to produce 1000 decimal digits of e requires $n \geq 450$.

Another key decision is how to store data with this number of digits. Python's float is inadequate, so we'll use lists – a number with 1000 decimal digits is stored in a list of 1000 digits (actually a little bigger to handle rounding errors):

```
edigs = [0]*(numDigs+2)
edigs[0] = 1
```

Having decided on the data structure, we can flesh out the pseudo-code given above. `edigs` starts by holding 1, which is divided by 450 using the grade school method. 1/450 = 0.00222...2, and these digits are stored back in `edigs`.

The next iteration commences by setting edigs[0] = edigs[0]+1, and then it's divided by 449. The integer part of the quotient q goes to edigs[i], and the remainder r is multiplied by 10 and added to edigs[i+1]. This loop continues until it reaches $n = 0$.

At this point many of the `edigs` elements will contain numbers larger than 9, and so the code makes a second pass over the list to perform 'carry' operations. Starting at the end, u = edigs[i] // 10 is added to edigs[i-1] and edigs[i] is reduced mod 10.

The complete program is available at exp1.py, and consists of three loops – the first implements the looping divisions, the second performs carrying, and the last prints out the result. The output:

```
> python exp1.py
numDigs n? 1000 451
2
71828 18284 59045 23536 02874 71352 66249 77572 47093 69995 95749 66967
62772 40766 30353 54759 45713 82178 52516 64274 27466 39193 20030 59921
81741 35966 29043 57290 03342 95260 59563 07381 32328 62794 34907 63233
82988 07531 95251 01901 15738 34187 93070 21540 89149 93488 41675 09244
76146 06680 82264 80016 84774 11853 74234 54424 37107 53907 77449 92069
55170 27618 38606 26133 13845 83000 75204 49338 26560 29760 67371 13200
70932 87091 27443 74704 72306 96977 20931 01416 92836 81902 55151 08657
46377 21112 52389 78442 50569 53696 77078 54499 69967 94686 44549 05987
```

```
93163 68892 30098 79312 77361 78215 42499 92295 76351 48220 82698 95193
66803 31825 28869 39849 64651 05820 93923 98294 88793 32036 25094 43117
30123 81970 68416 14039 70198 37679 32068 32823 76464 80429 53118 02328
78250 98194 55815 30175 67173 61332 06981 12509 96181 88159 30416 90351
59888 85193 45807 27386 67385 89422 87922 84998 92086 80582 57492 79610
48419 84443 63463 24496 84875 60233 62482 70419 78623 20900 21609 90235
30436 99418 49146 31409 34317 38143 64054 62531 52096 18369 08887 07016
76839 64243 78140 59271 45635 49061 30310 72085 10383 75051 01157 47704
17189 86106 87396 96552 12671 54688 95703 50354
In 0.21 secs
```

Table 6.1 shows various running times for `exp1.py` when tested with increasing numbers of digits and term sizes.

no. of digits	1000	10,000	100,000
no. of terms	451	3250	25207
time (secs)	0.21	14.94	1232.33

Table 6.1. Running times for calculating e

The code runs acceptably fast, but this grade school method uses $O(n^2)$ operations to multiply two n-digit numbers.

6.7.2 A Decimal Exponential Function. This version of the exponential function uses Python's decimal type to implement a fairly conventional algorithm. We begin with the exponential series (Equ. (6.20)), but don't calculate each term in its entirety on each loop (i.e. $t_k = x^k/k!$), which involves both a power and a factorial. It's more efficient to store the previous term (t_{k-1}) and use it as the building block for the current term ($t_k = t_{k-1} * (x/k)$). This is implemented in Listing 6.15 (`expD.py`):

The top-level code sets the required decimal precision and expD() is also bracketed by a further increase of two digits to reduce rounding errors. expD() returns when the value of e from the previous loop can't be distinguished from the value generated during the current loop.

```
def expD(x):
  decimal.getcontext().prec += 2
  term = D(1); eVal = D(1)
  t = 1
  while True:
    term *= x/t; ePrev = eVal
    eVal += term; t += 1
    if eVal == ePrev:
      print("No. steps:", t-1)
      break
  decimal.getcontext().prec -= 2
  return +eVal
```

Listing 6.15. Exponential using decimal

Table 6.2 presents the running times for expD(1) and Python's built-in decimal exp(1) for six different digit precisions (1000, 10000, 100000, 300000, 500000, 1 million). expD() is markedly faster.

Precision	expD(1) secs	exp(1) secs
1000	0.04	0.02
10,000	4.111	4.470
100,000	243	275
300,000	1844 (31 mins)	2300 (38 mins)
500,000	5688 (1.6 hrs)	7580 (2.1 hrs)
1 million	22624 (6.3 hrs)	26641 (7.4 hrs)

Table 6.2. Running times for expD(1) and exp(1)

Fig. 6.7 shows two graphs generated from the timing and iteration information returned by expD(1) for different digit precisions (d). The left-hand graph plots d against running time using log scales.

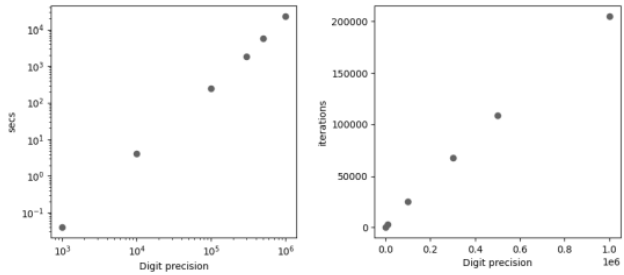


Figure 6.7. expD(1) running times and iterations

The right-hand graph shows that expD() exhibits a linear relationship between the number of operations performed in its while-loop and digit precision. However, the left-hand plot shows that the running time is exponential. The difference is due to the 'hidden' cost of the decimal operations – addition, multiplication, and division – whose running times are exponentially related to the size of the numbers being manipulated.

6.7.3 An Infinite Product for e.

In 1980, inspired by the Wallis product for π (see below), Nicholas Pippenger published a product formula for e:

$$\frac{e}{2} = \left(\frac{2}{1}\right)^{1/2} \left(\frac{2}{3}\frac{4}{3}\right)^{1/4} \left(\frac{4}{5}\frac{6}{5}\frac{6}{7}\frac{8}{7}\right)^{1/8} \cdots$$

For $n \geq 2$, the n-th factor is:

$$\left[\frac{2^{n-1} \cdots 2^{n}}{(2^{n-1}+1) \cdots (2^{v}-1)}\right]^{1/2^n}$$

This poses some interesting coding problems since the n-th term requires the multiplication of 2^{n-1} rationals, and their numerators and denominators are generated using different sequences of integers.

We decided initially to calculate the n-th term using two list comprehensions – one to build the product of all the integers in the numerator, and one for the denominator's values. However, the integer sequences proved too cumbersome to write that way, so we fell back to functions.

The numerator function, numer(), is shown on the right.
This approach proved useful during testing, but generates a very large list which is soon after reduced to a single integer. The amount of memory needed for larger terms may crash the program.

```
def numer(n):
  if n == 2:  return [2]
  start = n//2; elems = [0]*start
  elems[0] = start; elems[-1] = n
  mid = start + 2
  for i in range(1,len(elems)-1, 2):
    elems[i] = mid; elems[i+1] = mid
    mid += 2
  return elems
```

One alternative is to use Python generators, which we first discussed in the 'Lazy Evaluation' section of Ch. 1 while calculating factorials. Their usage here is more complex, but having the numer() function as a starting point makes the coding simpler. Essentially, each line that adds an element to the elems list in numer() is translated into a `yield` call.

The n-th term is constructed by pipTerm() in Listing 6.16 (`pippen.py`), which creates the generator functions for the numerator and denominator, and employs next() to retrieve integers. To be clear, this doesn't reduce the running time since 2^n integers are still being multiplied, but it does remove the memory overhead of storing those numbers in lists.

```
def pipTerm(n):
  numGen = numer(n)
  denGen = denom(n)
  frac = 1
  for i in range(n//2):
    frac = frac * next(numGen)/
        next(denGen)
  return frac**(1/n)
```

Listing 6.16. Pippenger's e product

The exponential running time means that the program very quickly grinds to a halt when asked to generate more than about 24 terms in the series. Unfortunately, the slow series convergence means that this is only enough to produce around 15 correct decimals (math.e = 2.718281828459045).

```
> python pippen.py
No. terms? 24
 est e: 2.718281828459053
Elapsed time: 2.210 secs
```

6.7.4 Google's Problem. In 2004, the advert in Fig. 6.8 appeared on billboards across the USA.

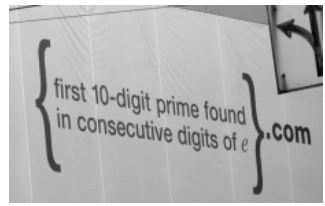


Figure 6.8. Google's billboard problem

With the help of the decimal version of exp(), and a prime number test function from Listing 2.13.1, this problem only takes a few lines to solve in Listing 6.17 (e10Prime.py).

```
decimal.getcontext().prec = 200
eEst = D(1).exp()
printGrouped(eEst)
eList = list(str(eEst).replace('.',''))
for i in range(len(eList)-10):
  num = int("".join(eList[i:i+10]))
  if prime(num):
    print(num, "at index", i)
    break
```

Listing 6.17. First 10-digit prime in e

It reports the first 10-digit prime to be 7427466391, starting at position 99.

Visiting 7427466391.com (when it was still online) led to a more difficult problem – finding the fifth term in the sequence 7182818284, 8182845904, 8747135266, 7427466391. Its solution led to a Google Labs page where the visitor could submit a resume.

6.7.5 How Many Decimals of π Do We Really Need? There's strange, but undeniable, fun in memorizing digits of π, and for writing code that generates untold millions of those digits (actually trillions nowadays), but how many are actually needed in real-life applications?

NASA and JPL addressed that question in an online article (https://www.jpl.nasa.gov/edu/news/2016/3/16/how-many-decimals-of-pi-do-we-really-need/). JPL's highest accuracy calculations, for interplanetary navigation, use 3.141592653589793 (15 decimal places). They justify this with an example involving Voyager 1, the most distant spacecraft from Earth.

Currently, Voyager 1 is about 24.379 billion kilometers away (https://voyager.jpl.nasa.gov/mission/status/). Let's say we want to calculate the circumference of a circle with that radius. Using π to 15 decimal places, that comes out to a little more than 153 billion kilometers which is wrong by about one centimeter compared to using π with 17 decimal places of accuracy (3.14159265358979324):

```
>>> from decimal import Decimal as D
>>> 2*D('3.14159265358979324')*24379071870 -
   (2*D('3.141592653589793')*24379071870)
Decimal('0.0000117019544976')
```

Note that to perform these calculations, we had to switch to decimals to get the necessary accuracy.

Computing π to excessive length has a very long history, which is tabulated and graphed at https://en.wikipedia.org/wiki/Chronology_of_computation_of_%CF%80. Our modest aim of calculating a million digits was first achieved way back in 1973 by Jean Guilloud and Martine Bouyer on a CDC 7600; the machine took 23.3 hours to complete the task.

6.7.6 Archimedes' π Estimate. The first recorded algorithm for calculating π was a geometrical approach using polygons, devised around 250 BC by Archimedes. This became the preferred technique for over 1,000 years.

He computed upper and lower bounds by drawing regular triangles inside and outside a unit circle, and successively doubling the number of their sides until they reached 96. As the polygons' sides double, their perimeters became ever better approximations to the circle's circumference.

Let I_n and C_n be the perimeter of the inscribed and circumscribed n-gons about a circle of radius 1/2. Archimedes derived two recurrence relationships to calculate C_{2n} and I_{2n} from I_n and C_n:

$$\begin{aligned} C_{2n} &= \frac{2C_nI_n}{C_n+I_n} \\ I_{2n} &= \sqrt{C_{2n}I_n} \end{aligned}$$

He obtained the following range:

$$3.1408 \approx \frac{223}{71} < \pi < \frac{22}{7} \approx 3.1429$$

Mathematicians using algorithms like Archimedes' reached 39 digits of accuracy in 1630, a record broken in 1699 when techniques employing infinite series reached 71 digits.

In the following, we'll restrict ourselves to calculating the sides of inscribed regular polygons, and will start the doubling using a square. We'll explain the steps by referring to Fig. 6.9.

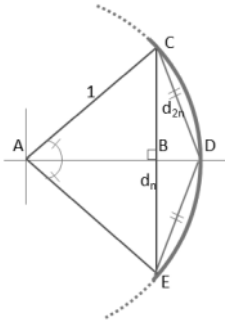


Figure 6.9. Archimedes' π approximation

A circle of radius 1 unit is centered at A, and a regular N sided polygon is inscribed within it. Fig. 6.9 shows one segment of the polygon, ACE, with the length of the side CE being d_n. After doubling the number of sides to $2N$, the triangle CAE will be bisected, making CAD and DAE two equal segments of the $2N$ polygon. The lengths CD and CE are both d_{2n}, and we want to determine a relationship between that value and d_n.

Using Pythagoras's theorem on the right angled triangle ABC:

$$\begin{aligned} \mathrm{AB}^2 + \mathrm{BC}^2 &= 1 \\ \mathrm{AB} &= \sqrt{1 - \mathrm{BC}^2} \end{aligned}$$

Given that $\mathrm{BC} = \frac{d_n}{2}$, we can rewrite:

$$\mathrm{AB} = \sqrt{1 - (\frac{d_n}{2})^2}$$

Use this to define BD:

$$\begin{aligned} \mathrm{BD} &= 1 - \mathrm{AB} \\ \mathrm{BD} &= 1 - \sqrt{1 - \frac{d_n^2}{4}} \end{aligned}$$

Using Pythagoras on the right angled triangle CBD:

$$\begin{aligned} \mathrm{CD}^2 &= \mathrm{BC}^2 + \mathrm{BD}^2 \\ &= (\frac{d_n}{2})^2 + \left(1 - \sqrt{1 - (\frac{d_n}{2})^2}\right)^2 \\ &= \frac{d_n^2}{4} + \left(1 - 2\sqrt{1 - \frac{d_n^2}{4}} + (1 - \frac{d_n^2}{4})\right) \\ \mathrm{CD}^2 &= 2 - 2\sqrt{1 - \frac{d_n^2}{4}} \\ \mathrm{CD} &= d_{2n} = \sqrt{2 - 2\sqrt{1 - \frac{d_n^2}{4}}} \end{aligned}$$

CD is one side of the polygon with $2N$ sides, and the above equation specifies its length in terms of the length of a side of the polygon with N sides.

The square is replaced by a regular octagon, with side length d_8, and then with a 16-gon with side length d_{16}. Using the above equation, these lengths can be calculated, and $nd_n \to 2\pi$.

We start the approximation by inscribing a square inside the unit circle. This means that $d_4 = \sqrt{2}$ as in Fig. 6.10.

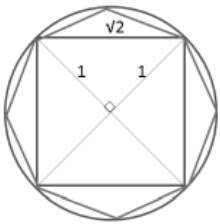


Figure 6.10. A square and octagon inside a unit circle

If we code this using Python floats, we only get around 9 decimal places of accuracy since the rounding errors are quite severe. This is due to the algorithm using subtraction to calculate the length of a side (i.e. CD in the equation above), and also using square roots.

A possible solution is to stick with the algorithm but switch to Python decimals, as in Listing 6.18 (archimedesPi.py).

```
def archPi():
  decimal.getcontext().prec += 2
  pi = D(3)
  edgeSqr = D(2); sides = 4
  while True:
    edgeDiff = 1 - edgeSqr/4
    prevES = edgeSqr
    edgeSqr = 2 - 2*edgeDiff.sqrt()
    sides *= 2
    prevPi = pi
    pi = sides * edgeSqr.sqrt()/2
    if pi == prevPi:
      break
  decimal.getcontext().prec -= 2
  return +pi
```

Listing 6.18. Archimedes' calculation of π

When the program is run:

```
> python archimedesPi.py
No. dps? 50
   3. 14159 26535 89793 23846 26433 62208 45594 33938 55342 3541
Elapsed time: 0.000 secs
```

A simple way to verify the accuracy of this result is to check it against an online version, such as "pi to 10,000 digits" at the Univ. of Utah (`https://www.math.utah.edu/~alfeld/math/pi.html`). It begins:

```
    3.14159 26535 89793 23846 26433 83279 50288 41971 ...
```

Python decimals reduce the rounding errors a little but they reappear at the 26th decimal.

A related approach was also used by Archimedes to calculate the area under a parabola (see Sec. 6.10).

6.7.7 Estimating π using Infinite Series. We'll use the geometric series:

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + \cdots$$

and trust that it converges for $x \in (-1, 1)$. If we replace x with $-x^2$, we get

$$\frac{1}{1+x^2} = 1 - x^2 + x^4 - x^6 + x^8 + \cdots \tag{6.21}$$

Consider $y = \arctan x$ and its equivalent identity $x = \tan y$. Implicit differentiation with respect to x yields

$$1 = \sec^2 y \cdot \frac{dy}{dx}$$

and using the identity $\sec^2 y = 1 + \tan^2 y$,

$$\frac{dy}{dx} = \frac{1}{1 + \tan^2 y}.$$

Since $x = \tan y$, we have

$$\frac{dy}{dx} = \frac{d}{dx} \arctan x = \frac{1}{1+x^2}$$

Apply Equ. (6.21) to the right side, then integrate both sides, to obtain

$$y = \int (1 - x^2 + x^4 - x^6 + x^8 + \cdots)\, dx$$

or

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \cdots \tag{6.22}$$

Fortunately, this series converges for $x = 1$, which let's us write:

$$\frac{\pi}{4} = 1 - \frac{1}{3} - \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$$

This formula was discovered independently by James Gregory (1638-1675) and Gottfried Leibniz (1646-1716), among others.

Although Equ. (6.22) expresses π as an easily programmed sequence, it approaches π at a crawl, as the following demonstrates:

```
> python seriesPi.py
fnm (leibniz, gregory, wallis, basel)? leibniz
Series leibniz(n)
     n         estPi              err      secs
      10: 3.041839618929; -0.099753034660; 0.000
     100: 3.131592903559; -0.009999750031; 0.000
    1000: 3.140592653840; -0.000999999750; 0.000
```

```
   10000: 3.141492653590; -0.000100000000; 0.000
 1000000: 3.141591653590; -0.000001000000; 0.110
10000000: 3.141592553589; -0.000000100001; 0.770

> python seriesPi.py
fnm (leibniz, gregory, wallis, basel)? gregory
Series gregory(n)
     n         estPi              err      secs
      10: 3.041839618929; -0.099753034660; 0.000
     100: 3.131592903559; -0.009999750031; 0.000
    1000: 3.140592653840; -0.000999999750; 0.000
   10000: 3.141492653590; -0.000100000000; 0.010
 1000000: 3.141591653590; -0.000001000000; 0.490
10000000: 3.141592553590; -0.000000100000; 4.770
```

seriesPi.py executes the specified series function up to the n-th term, where n varies from 10 up to 10 million. For each n, the summation, its error compared to math.pi (3.141592653589793), and running time are printed.

The Leibniz formula uses a list comprehension to generate two adjacent terms in the series for each i in the iteration, while the Gregory formula uses an arctan() function. Mathematically, the two approaches are identical, and produce the same estimations and errors, although Gregory's atan() is noticeably slower for larger n's.

```
def leibniz(n):
  return sum([4/i - 4/(i+2)
    for i in range(1, 2*n+1, 4)])

def gregory(n):
  return 4*atan(1,n)

def atan(x, n):
  tot = 0; mult = 1
  for i in range(n):
    tot += mult * (x**(2*i+1))/(2*
        i+1)
    mult *= -1
  return tot
```

Listing 6.19. Leibniz and Gregory formulae

6.7.7.1 Wallis' Product. In 1655, John Wallis discovered a remarkable formula that could compute the digits of π using a product that separated even and odd numbers:

$$\frac{\pi}{2} = \frac{2}{1} \times \frac{2}{3} \times \frac{4}{3} \times \frac{4}{5} \times \frac{6}{5} \times \frac{6}{7} \times \cdots$$

Wallis's long derivation is explained very nicely in Appendix A of Perkins [**Per17**].

The formula can be expressed succinctly. Unfortunately, it has a similarly slow rate of convergence as the earlier formulae.

```
def wallis(n):
  return 2 * math.prod((i*i)/((
      i-1)*(i+1)) for i in
      range(2, n, 2))
```

Listing 6.20. The Wallis formula

6.7.7.2 Euler's Solution to the Basel Problem. Pietro Mengoli posed the Basel Problem in 1644: find the numerical value of:

$$1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \frac{1}{25} + \cdots = \sum_{k=1}^{\infty} \frac{1}{k^2}$$

Wallis, the Bernoullis, and others all had an attempt, but were hindered by the series' lethargic convergence rate. Even with n set to 1 million, only five decimal places in the result are accurate. Leonhard Euler's solution in 1735 (when aged 28) brought him immediate fame, although by modern standards the proof is a little sketchy. He began by writing $\sin x$ as an infinite product of linear factors:

$$\sin x = x(1 - \frac{x}{\pi})(1 + \frac{x}{\pi})(1 - \frac{x}{2\pi})(1 + \frac{x}{2\pi}) \cdots$$

He justified this by noting that $\sin(x)$ has roots at $x = 0, \pm\pi, \pm 2\pi, \pm 3\pi, ...$, but it's not really clear whether this equivalence is valid. But let's continue:

$$\begin{aligned}
\sin \pi x &= \pi x(1 - x)(1 + x)(1 - \tfrac{x}{2})(1 + \tfrac{x}{2}) \cdots \\
&= \pi x(1 - x^2)(1 - \tfrac{x^4}{4})(1 - \tfrac{x^2}{9}) \cdots \\
&= \pi x - \pi x^3(1 + \tfrac{1}{4} + \tfrac{1}{9} + \tfrac{1}{16} + \cdots) + \\
&\quad \pi x^5(\tfrac{1}{1 \cdot 4} + \tfrac{1}{1 \cdot 9} + \cdots + \tfrac{1}{4 \cdot 9} + \cdots) - \cdots
\end{aligned}$$

We can also write $\sin(x)$ as a Taylor series:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

so

$$\sin(\pi x) = \pi x - \frac{(\pi x)^3}{3!} + \frac{(\pi x)^5}{5!} - \frac{(\pi x)^7}{7!} + \cdots$$

The x^3 terms of these two series must be the same, which means:

$$\pi\left(1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \cdots\right) = \frac{\pi^3}{6}$$

and so

$$\frac{\pi^2}{6} = \left(1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \cdots\right)$$

A more rigorous proof by Karl Weierstrass, depending on complex analysis, would only appear in the mid 1800s. Euler's series in Python:

```
def basel(n):
  return math.sqrt(6*sum([1/(x*x) for x in range(1, n)]))
```

Listing 6.21. The Basel formula for calculating π

6.7.8 Machin's Arctangent Formula. In 1706, John Machin found a way to coax the arctangent function into approximating π more quickly.

Assuming the tangent addition formula:

$$\tan(\alpha \pm \beta) = \frac{\tan(\alpha) \pm \tan(\beta)}{1 \mp \tan(\alpha)\tan(\beta)}$$

If we let $\alpha = \arctan(A)$ and $\beta = \arctan B$ then we can rearrange the equation to obtain an arctangent addition formula

$$\arctan(A) \pm \arctan(B) = arctan(\frac{A \pm B}{1 \mp AB}) \qquad (\text{mod } \pi),\ AB \neq 1.$$

We use this to double the arctan of 1/5 twice:

$$\begin{aligned} 2\arctan\tfrac{1}{5} &= \arctan\tfrac{1}{5} + \arctan\tfrac{1}{5} \\ &= \arctan\left(\frac{\frac{1}{5}+\frac{1}{5}}{1-\frac{1}{5}\cdot\frac{1}{5}}\right) = \arctan\tfrac{5}{12} \end{aligned}$$

and

$$\begin{aligned} 4\arctan\tfrac{1}{5} &= \arctan\tfrac{5}{12} + \arctan\tfrac{5}{12} \\ &= \arctan\left(\frac{\frac{5}{12}+\frac{5}{12}}{1-\frac{5}{12}\cdot\frac{5}{12}}\right) = \arctan\tfrac{120}{119} \end{aligned}$$

Combining these gives

$$\begin{aligned} 4\arctan\tfrac{1}{5} - \arctan\tfrac{1}{239} &= \arctan\tfrac{120}{119} - \arctan\tfrac{1}{239} \\ &= \arctan\left(\frac{\frac{120}{119}-\frac{1}{239}}{1+\frac{120}{119}\cdot\frac{1}{239}}\right) \\ &= \arctan\frac{28561/28441}{28561/28441} = \arctan 1 = \frac{\pi}{4} \end{aligned}$$

This kind of rewriting can generate a great number of similar identities,

$$\begin{aligned}
\pi &= 16\arctan\tfrac{1}{5} - 4\arctan\tfrac{1}{239} && \text{(Machin 1706)}\\
\pi &= 24\arctan\tfrac{1}{8} + 8\arctan\tfrac{1}{57} + 4\arctan\tfrac{1}{239} && \text{(Störmer 1896)}\\
\pi &= 48\arctan\tfrac{1}{18} + 32\arctan\tfrac{1}{57} - 20\arctan\tfrac{1}{239} && \text{(Gauss).}
\end{aligned}$$

Machin, Störmer, and Gauss's formulae are implemented in machinPi.py, using a decimal implementation of arctan() from decTrig.py. It's based on Euler's arctangent series which converges more quickly than the Taylor series:

$$\arctan(x) = \sum_{n=0}^{\infty} \frac{2^{2n}(n!)^2}{(2n+1)!}\frac{x^{2n+1}}{(1+x^2)^{n+1}}$$

machinPi.py calculates three arctan variants, producing π to 100 digits, but only their timing results are shown below:

```
> python machinPi.py
Machin 1). pi = 4 * arctan(1)
5.080 secs
Machin 2). pi = 16*arctan(1/5) - 4*arctan(1/239)
1.548 secs
Stormer). pi = 24*arctan(1/8) + 8*arctan(1/57) + 4*arctan(1/239)
1.912 secs
Gauss). pi = 48*arctan(1/18) + 32 arctan(1/57) - 20*arctan(1/239)
1.905 secs
```

It's clearly worthwhile to rewrite the arctan formula 'Machin 1)' as 'Machin 2)' since it results in a three-fold speed increase. The Störmer and Gauss variants illustrates that using more arctan() terms can be expensive.

6.7.8.1 Computing Arctangent Recursively. Can Euler's arctan() be replaced by something faster? We can compute $r\arctan(1/a)$ by means of the recurrences

$$\begin{gathered}
u_{-1} = ra, \quad u_{2n+1} = -u_{2n-1}\frac{2n-1}{(2n+1)a^2},\\
s_{-1} = 0, \quad s_{2n+1} = s_{2n-1} + u_{2n+1}.
\end{gathered}$$

Then

$$r\arctan\frac{1}{a} = \lim_{x\to\infty} s_{2n+1}.$$

```
def rarctan(r, a, n=100):
  # uses n iterations
  return s(r, a, 2*n+1)

def s(r, a, n):
  if n == -1: return D(0)
  else:
    return s(r, a, n-2) + u(r, a, n)

def u(r, a, n):
  if n == -1: return r*a
  else:
    return -u(r, a, n-2)*(n-2)/(n*a**2)
```

Listing 6.22. A recursive definition of arctangent

These relations can be converted into recursive functions in rarctanPi.py. Timings for the calculations of π with this version of arctan() are much faster:

```
> python rarctanPi.py
 Machin: 0.010 secs
Stormer: 0.020 secs
  Gauss: 0.020 secs
```

Note that rarctan() is calculating $1/a$, so the Machin, Störmer, and Gauss equations must be supplied with decimal integers not rationals.

6.7.9 Efficient Algorithms for Calculating π. We'll discuss four modern algorithms for calculating π, which converge much more quickly than the older approaches.

6.7.9.1 Ramanujan. Srinivasa Ramanujan discovered a new family of series for π, one of which is

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

ramaPi.py is a direct translation. It converges with amazing speed, and forms the basis of some of the fastest algorithms currently being used. The following outputs show that 15 decimal digits only requires 4 iterations, while 100 digits needs just 14:

```
> python ramaPi.py
No. dps? 15
No. iterations: 4
   3. 14159 26535 8979
Elapsed time: 0.000 secs

> python ramaPi.py
No. dps? 100
No. iterations: 14
   3. 14159 26535 89793 23846 26433 83279 50288 41971 69399 37510 58209
74944 59230 78164 06286 20899 86280 34825 34211 7068
Elapsed time: 0.000 secs
```

6.7.9.2 Salamin-Brent. Since 1976 we've had the quadratically convergent Salamin-Brent algorithm (`https://en.wikipedia.org/wiki/Gauss%E2%80%93Legendre_algorithm`), based on work by Gauss and Adrien-Marie Legendre (1752–1833), combined with modern multiplication and square roots techniques. It repeatedly replaces two numbers by their arithmetic and geometric means, to approximate their arithmetic-geometric mean. Its derivation is difficult but the resulting algorithm is easy to code (see `sbPi.py`). It converges faster (see Table 6.3) than any of the other algorithms in this section.

No. of digits	Running time (secs)	No. of iterations
1000	0.010	10
10,000	0.740	13
50,000	7.550	16
100,000	16.961	17
500,000	119.792	19
1 million	277.607	20

Table 6.3. Times for different digit precisions using Salamin-Brent

6.7.9.3 Chudnovsky. A similar series to Ramanujan's was devised in 1988 by G. V. and D. V. Chudnovsky at Columbia University (`https://en.wikipedia.org/wiki/Chudnovsky_algorithm`). Recently (2022), it was used in the world record calculation of 100 trillion digits of π.

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!(545140134k + 13591409)}{(3k)!(k!)^3 (640320)^{3k+3/2}}$$

There are several approaches for speeding up its execution, and we chose a simple iterative implementation detailed at `https://handwiki.org/wiki/Chudnovsky_algorithm`. The code can be found in `chudPi.py`, but its timing results are disappointing:

```
> python chudPi.py
No. dps? 1000
No. steps: 71
   3. 14159 26535 89793 23846 26433 ...
80532 17122 68066 13001 92787 66111 95909 21642 0199
Elapsed time: 0.020 secs
```

6.7.9.4 Bailey–Borwein–Plouffe. The Bailey–Borwein–Plouffe (BBP) formula was discovered in 1995 by Simon Plouffe and described by David H. Bailey, Peter

Borwein, and Plouffe soon after:

$$\pi = \sum_{k=0}^{\infty} \left(\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \right)$$

BBP is a *spigot* algorithm which can generate digits one at a time, independent of earlier digits in the sequence. BPP is a spigot for the base 16 digits of π, but another formula discovered by Plouffe in 2022 allows the extraction of the n-th decimal digit. BBP was used in the popular PiHex tool for calculating π using distributed computing (`https://en.wikipedia.org/wiki/PiHex`).

bppPi.py is a direct translation of the formula into code, and runs rather poorly.

```
> python bppPi.py
No. dps? 1000
No. steps: 828
   3. 14159 26535 89793 23846 26433 ...
80532 17122 68066 13001 92787 66111 95909 21642 0199
Elapsed time: 0.090 secs
```

Formulae of the same general form:

$$\sum_{k=0}^{\infty} \left(\frac{1}{b^k} \frac{p(k)}{q(k)} \right)$$

have been discovered for calculating many other irrationals. $p(k)$ and $q(k)$ are polynomials with integer coefficients and $b \geq 2$ is an integer base. At present, there's no technique for finding $p(k)$ and $q(k)$, except by trial-and-error.

6.7.10 Are e and π Normal? A real number is *simply normal* in an integer base b if its infinite sequence of digits is distributed uniformly so that each of the b digit values has the same density $1/b$. In other words, no digit occurs more frequently than any other.

A number is *normal* if it is simply normal in all integer bases ≥ 2. It's widely believed that e and π are normal, but no proof has yet been devised.

normalNum.py loads a number from a file, and counts its digits. The counts are plotted as bars, with the mean and standard deviation shown as lines over the bars. pi10000.txt and e10000.txt contain the first 10,000 digits of π and e, generated by Peter Alfeld at the Univ. of Utah; longer sequences are also available. Fig. 6.11 suggests that e and π are uniformly distributed, at least for their first 10,000 digits.

If e and π are normal then they could be used as a source of random numbers. But this runs counter to Kolmogorov and Chaitin's definition of randomness which states that a finite sequence is random if its shortest description (program) is about as long as the sequence itself (`https://en.wikipedia.org/wiki/Kolmogorov_complexity`). However, e and π can be produced by very

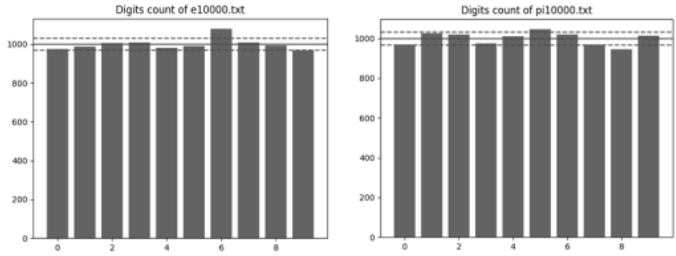


Figure 6.11. Digit counts for e and π

short programs (as we've seen), so according to this definition, their digits must be extremely non-random.

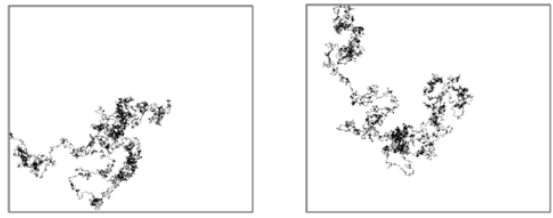
Chudnovsky believes that there must be hidden regularities in e and π, at least in some base. For instance, consider 9**(9**(9**9)) and 2**(2**(2**(2**(2**2)))). These huge numbers have very simple descriptions but their decimal representations look completely random. However, in base 9 the first number is a 1 followed by 0's, and so very regular. Similarly, the binary version of the second number is 1 followed by a huge number of 0's.

An interesting way to visualize randomness is to use a vectorgram, an approach first proposed by John Venn, and popularized by Clifford A. Pickover [**Pic01**] (see Chapter 55). Each digit is mapped to a different vector evenly arrayed around a point. Each digit of a number is drawn as a line from the cursor's current location, moving it to a new position. The overall result, as Pickover has observed, looks like the steps a drunkard wandering in an open field.

Fig. 6.12 shows the vectorgrams for the first 10,000 digits of e and π, with both starting from the center of the window (see vectorgram.py). The vectorgrams move off the edge of the screen shortly before all the digits had been rendered – e departs from the left edge, and π disappears off the top.

One question these diagrams raise is whether the drunkard ever returns to the starting point of their walk (the center of the window). In 1921 George Pólya proved that he almost surely would, but for 3 dimensions or higher, the probability decreases as the number of dimensions increases. In 3 dimensions, the probability is only about 35%. Shizuo Kakutani summarized this as: "A drunk man will find his way home, but a drunk bird may get lost forever".

In fact, the man will return infinitely often to his start point, but the expected time between returns is infinite. The drunk is rarely close to the origin and a large proportion of his time is spent far away from it.



e. Finishes at the left edge. π. Finishes at the top edge.

Figure 6.12. Vectorgrams for e and π

Exercises

(1) (Roy North.) Compute π by adding the first 500 terms of Leibniz' series accurately. Find the first and the second wrong digit. Do the same with 5000 terms. Can you explain your observations? (Hint: Look at Sec. 2.1.5 for ideas on transforming the tail of Leibniz' series.)

6.8 Least-Squares Curve Fitting

We want to determine an equation for a smooth curve which doesn't have to pass exactly through the set of given (x, y) points, but must be near to them. A 'best fit' is obtained by applying the least-squares method which minimizes the squares of the errors between the y data and the values produced by the curve. We'll look at linear and quadratic regression using least-squares, and finish by employing Gaussian elimination to solve the generated simultaneous equations.

6.8.1 Linear Regression. Linear regression utilizing least-squares finds a straight line that most closely approximates the supplied points. It determines a line $Y = b + mx$ for which $S = \sum_{i=1}^{n}(Y_i - y_i)^2$ is a minimum for the n data points. The values $Y_i - y_i$ are called *residuals*, and correspond to the vertical lines in Fig. 6.13.

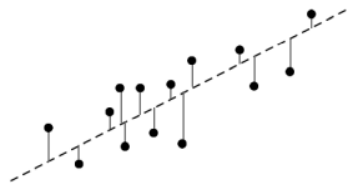


Figure 6.13. Using least-squares to fit a line to points

Minimum values for b and m are obtained by setting two partial derivatives of S with respect to b and m to zero:

$$\frac{\partial S}{\partial b} = 0 \quad \text{and} \quad \frac{\partial S}{\partial m} = 0$$

This yields two simultaneous linear equations:

$$\frac{\partial S}{\partial b} = \sum_{i=1}^{n} \frac{\partial}{\partial b}(b + mx_i - y_i)^2 = \sum_{i=1}^{n} 2(b + mx_i - y_i) = 0 \tag{6.23}$$

and

$$\frac{\partial S}{\partial m} = \sum_{i=1}^{n} \frac{\partial}{\partial m}(b + mx_i - y_i)^2 = \sum_{i=1}^{n} 2x_i(b + mx_i - y_i) = 0 \tag{6.24}$$

From Eqs. (6.23) and (6.24), we obtain:

$$\sum_{i=1}^{n} y_i = nb + m\sum_{i=1}^{n} x_i \tag{6.25}$$

and

$$\sum_{i=1}^{n} x_i y_i = b\sum_{i=1}^{n} x_i + m\sum_{i=1}^{n} x_i^2 \tag{6.26}$$

Eqs. (6.25) and (6.26) are the two conditions that must be met in order to obtain the best fitting line. From (6.25):

$$b = \frac{1}{n}\sum_{i=1}^{n} y_i - \frac{1}{n}\sum_{i-1}^{n} mx_i \;=\; \bar{y} - m\bar{x}$$

where $\bar{x}$ and $\bar{y}$ are the means of the x and y values.

Substitute this value for b into (6.26),

$$\sum_{i=1}^{n} x_i y_i = (\sum_{i=1}^{n} y_i - \frac{1}{n}\sum_{i=1}^{n} mx_i)\sum_{i=1}^{n} x_i + \sum_{i=1}^{n} mx_i^2$$

Rearrange so m is on the left:

$$m = \frac{\sum_{i=1}^{n} x_i y_i - \frac{1}{n}\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}(\sum_{i=1}^{n} x_i)^2}$$

Rearrange so divisions by n are replaced by multiplications:

$$m = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2}$$

These b and m calculations are implemented by estimateCoefs() in `linReg.py`:

```
def estimateCoefs(xs, ys):
  # estimating coefficients in y = b + mx
  n = len(xs)
  sumX = sum(xs); sumY = sum(ys)
  sumXY = sum([x*y for (x,y) in zip(xs,ys)])
  sumX2 = sum([x**2 for x in xs])
  m = (n*sumXY  - sumX*sumY)/(n*sumX2 - sumX**2)
  b = (sumY - m*sumX)/n
  return b, m
```

Listing 6.23. Estimating linear coefficients

It's also useful to measure of how well the curve (a straight line in this case) fits the data. The *correlation coefficient*, r, is always between 0 and 1, with a value closer to 1 indicating a better fit. It's calculated by assuming that the relationship between the x- and y- values follows a normal distribution so the variance in the y-values is a constant. The definition for r:

$$r = \sqrt{\frac{S_m - \text{RSS}}{S_m}}$$

where RSS, the residual sum of squares, is the value of S when the residuals use y coordinates from the line:

$$\text{RSS} = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (b + mx_i - y_i)^2$$

S_m is the spread of the points around the mean $\bar{y}$:

$$S_m = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

r is calculated by Listing 6.24 (linReg.py):

```
def calcCorrCoef(ys, yPreds):
  n = len(ys); my = sum(ys)/n
  rss = sum([(y - yP)**2 for (y,yP) in zip(ys,yPreds)])
    # residual sum of squares
  mspread = sum( [(y-my)**2 for y in ys])
    # spread of model around the mean
  return math.sqrt((mspread-rss)/mspread)
```

Listing 6.24. Calculating the correlation coefficient

The yPreds list holds the y coordinates calculated using the line equation.

The original data points are drawn as a scatter chart by plotReg() in linReg.py, and the fitted line is drawn over them, as in Fig. 6.14.

The correlation coefficient for this line is 0.976 which is a strong indication that the data does follow a linear relationship.

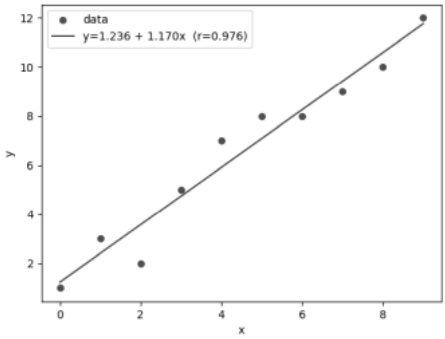


Figure 6.14. A best-fit line using least squares

6.8.2 Quadratic Regression. The least-squares approach can be generalized to fit higher-order polynomials to data, such as the quadratic, $Y = Ax^2 + Bx + C$.

Once again we aim to minimize the residuals in $S = \sum(Y_i - y_i)^2$ (from now on, we'll assume that the summations are over n values). Since there are three unknowns (A, B, and C), three partial derivatives must be set to 0.

$$\frac{\partial S}{\partial A} = 0, \quad \frac{\partial S}{\partial B} = 0 \quad \text{and} \quad \frac{\partial S}{\partial C} = 0$$

This yields three simultaneous equations:

$$\frac{\partial S}{\partial A} = \sum 2x_i^2((Ax_i^2 + Bx_i + C) - y_i) = 0$$

and

$$\frac{\partial S}{\partial B} = \sum 2x_i((Ax_i^2 + Bx_i + C) - y_i) = 0$$

and

$$\frac{\partial S}{\partial C} = \sum 2((Ax_i^2 + Bx_i + C) - y_i) = 0$$

Solving for C in the last equation gives:

$$C = \frac{\sum y_i - A\sum x_i^2 - B\sum x_i}{n}$$

Substituting this into the middle equation produces:

$$B = \frac{\sum x_i \sum y_i - n\sum x_i y_i - A(\sum x_i \sum x_i^2 - n\sum x_i^3)}{(\sum x_i)^2 - n\sum x_i^2}$$

To reduce the size of the equations, let:

$$D = (\sum x_i)^2 - n\sum x_i^2$$

Substituting the expressions for C and B into the $\frac{\partial S}{\partial A}$ equation simplifies it to $Au + v = 0$, where u and v are the rather daunting:

$$u = \frac{(\sum x_i \sum x_i^2 - n \sum x_i^3)(\sum x_i^3 - (1/n) \sum x_i \sum x_i^2)}{D} - \sum x_i^4 + \frac{1}{n}(\sum x_i^2)^2$$

and

$$v = \frac{(\sum x_i \sum y_i - n \sum x_i y_i)(\frac{1}{n} \sum x_i \sum x_i^2 - \sum x_i^3)}{D} + \sum x_i^2 y_i - \frac{1}{n} \sum x_i^2 \sum y_i$$

$A = -v/u$ is substituted into the B equation to obtain B, and A and B are used in the C equation to get C's value.

These steps are carried out by estimateCoefs() in (Listing 6.25; quadReg.py):

```
def estimateCoefs(xs, ys):
  # estimating coefficients in y = ax^2 + bx + c
  n = len(ys); sumX = sum(xs); sumY = sum(ys)
  sumXY = sum([x*y for (x,y) in zip(xs,ys)] )
  sumX2 = sum([x**2 for x in xs])
  sumX2Y = sum([x**2*y for (x,y) in zip(xs,ys)])
  sumX3 = sum([x**3 for x in xs])
  sumX4 = sum([x**4 for x in xs])
  denom = sumX*sumX - n*sumX2
  u = ((sumX*sumX2 - n*sumX3)*
    (sumX3 - (sumX*sumX2)/n))/denom - sumX4 + (sumX2**2)/n
  v = ((sumX*sumY - n*sumXY)*
    (sumX*sumX2/n - sumX3))/denom + sumX2Y - (sumX2*sumY)/n
  A = -v/u
  B = (sumX*sumY - n*sumXY - A*(sumX*sumX2 - n*sumX3))/denom
  C = (sumY - A*sumX2 - B*sumX)/n
  return A,B,C
```

Listing 6.25. Estimating quadratic coefficients

The top-level of the code for quadratic regression in quadReg.py is quite similar to the linear version, reusing its correlation coefficient and plotting functions. The only difference is that a quadratic function is used to generate the yPreds list.

The close correlation of the parabola to the data points ($r = 0.998$) in Fig. 6.15 suggests that this is the correct relationship between the points.

6.8.3 Solving Simultaneous Equations. The least-squares method for linear regression had us solve two simultaneous equations, while a quadratic curve requires three. If we want to fit higher-order polynomials, it makes sense to switch to using simultaneous equation solving tools. They'll handle the increasingly complex equation rewriting, and also deal with any accuracy problems caused by similar equations.

To find the minimum value of S that fits a m-th degree polynomial, we solve:

$$S = \sum (Y_i - y_i)^2 = \sum (k_0 + k_1 x_i + k_2 x_i^2 + \cdots + k_m x_i^m - y_i)^2$$

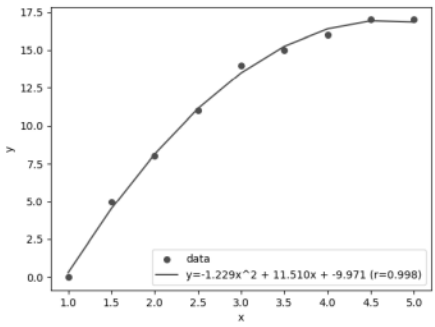


Figure 6.15. A best-fit parabola using least squares

The minimum S is now a function of $m+1$ variables, $k_0, k_1, ..., k_m$, so $m+1$ partial derivatives are set to zero:

$$\begin{aligned}
\frac{\partial S}{\partial k_0} &= \sum 2(k_0 + k_1 x_i + k_2 x_i^2 + \cdots + k_m x_i^m - y_i) = 0 \\
\frac{\partial S}{\partial k_1} &= \sum 2x_i(k_0 + k_1 x_i + k_2 x_i^2 + \cdots + k_m x_i^m - y_i) = 0 \\
&\vdots \\
\frac{\partial S}{\partial k_m} &= \sum 2x_i^m(k_0 + k_1 x_i + k_2 x_i^2 + \cdots + k_m x_i^m - y_i) = 0
\end{aligned}$$

We obtain $m+1$ simultaneous linear equations:

$$\begin{aligned}
k_0 n + k_1 \sum x_i + k_2 \sum x_i^2 + \cdots + k_m \sum x_i^m - \sum y_i &= 0, \\
k_0 \sum x_i + k_1 \sum x_i^2 + k_2 \sum x_i^3 + \cdots + k_m \sum x_i^{m+1} - \sum x_i y_i &= 0, \\
k_0 \sum x_i^2 + k_1 \sum x_i^3 + k_2 \sum x_i^4 + \cdots + k_m \sum x_i^{m+2} - \sum x_i^2 y_i &= 0, \\
&\vdots \\
k_0 \sum x_i^m + k_1 \sum x_i^{m+1} + k_2 \sum x_i^{m+2} + \cdots + k_m \sum x_i^{m+m} - \sum x_i^m y_i &= 0
\end{aligned}$$

It's convenient to express these using matrix notation,

$$[A]\,[k] = [B]$$

where

$$[A] = \begin{bmatrix} n & \sum x_i & \sum x_i^2 & \cdots & \sum x_i^m \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \cdots & \sum x_i^{m+1} \\ \vdots & & & & \\ \sum x_i^m & \sum x_i^{m+1} & \sum x_i^{m+2} & \cdots & \sum x_i^{2m} \end{bmatrix}$$

is a symmetric matrix, and

$$[k] = \begin{bmatrix} k_0 \\ k_i \\ \vdots \\ k_m \end{bmatrix}, \qquad [B] = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \\ \vdots \\ \sum x_i^m y_i \end{bmatrix}.$$

Instead of solving these equations by rewriting them by hand, we'll utilize Gaussian elimination (`https://en.wikipedia.org/wiki/Gaussian_elimination`). As a check, we'll apply this approach to obtain coefficients for the quadratic regression we considered above.

The quadratic (now labeled as $k_0 + k_1 x + k_2 x^2$) is represented by three linear equations, which in matrix form are

$$[A] = \begin{bmatrix} n & \sum x_i & \sum x_i^2 \\ \sum x_i & \sum x_i^2 & \sum x_i^3 \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 \end{bmatrix}$$

and

$$[k] = \begin{bmatrix} k_0 \\ k_i \\ k_2 \end{bmatrix}, \qquad [B] = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \\ \sum x_i^2 y_i \end{bmatrix}$$

Gaussian elimination solves an *augmented* matrix which concatenates $[A]$ and $[B]$:

$$\begin{bmatrix} n & \sum x_i & \sum x_i^2 & \sum y_i \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \sum x_i y_i \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \sum x_i^2 y_i \end{bmatrix}.$$

Our gaussian() function is passed this matrix as a 2D list, and returns a list representing $[k]$, as shown in Listing 6.26 (quadRegGauss.py).

```
def estimateCoefs(xs, ys):
  # estimating coefficients in y = ax^2 + bx + c
  n = len(ys); sumX = sum(xs); sumY = sum(ys)
  sumXY = sum([x*y for (x,y) in zip(xs,ys)] )
  sumX2 = sum([x**2 for x in xs])
  sumX2Y = sum([x**2*y for (x,y) in zip(xs,ys)])
  sumX3 = sum([x**3 for x in xs])
  sumX4 = sum([x**4 for x in xs])
  row1 = [n,     sumX,  sumX2, sumY]
  row2 = [sumX,  sumX2, sumX3, sumXY]
  row3 = [sumX2, sumX3, sumX4, sumX2Y]
  xs = gaussian([row1, row2, row3])
  return xs[2], xs[1], xs[0]   # A, B, C
```

Listing 6.26. Estimating quadratic coefficients using Gaussian elimination

Listing 6.26 returns values for A, B, and C in the same format as the function that implements quadratic regression in Listing 6.25. The next section focuses on how our gaussian() function works.

6.8.4 Gaussian Elimination. Gaussian elimination solves a system of linear equations by applying a sequence of row-wise operations to an augmented matrix holding the equations' coefficients. The aim is to produce an upper triangular matrix which has zeros in its lower left-hand corner. The row operations are restricted to:

- Swapping two rows;
- Multiplying a row by a nonzero number;
- Adding a multiple of one row to another row.

As an example, consider the equations:

$$\begin{aligned} 2x + y - z &= 8 \\ -3x - y + 2z &= -11 \\ -2x + y + 2z &= 3 \end{aligned}$$

These are translated into:

$$\begin{bmatrix} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & 11 \\ -2 & 1 & 2 & -3 \end{bmatrix}$$

The algorithm begins by eliminating the *first* element in the rows beneath the *first* row. This is done by subtracting multiples of the first row from the second and third rows. The result:

$$\begin{bmatrix} 2 & 1 & -1 & 8 \\ 0 & 1/2 & 1/2 & 1 \\ 0 & 2 & 1 & 5 \end{bmatrix}$$

The algorithm now moves on to the *second* row, with the goal of eliminating the *second* element in the lower rows. This is done by subtracting a suitable multiple of the second row from the third row, to produce:

$$\begin{bmatrix} 2 & 1 & -1 & 8 \\ 0 & 1/2 & 1/2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

The matrix is now in triangular form, which allows values for z, y, and x to be extracted. The matrix states that $-z = 1$, and so $z = -1$. That value is substituted into row 2, which specifies that $0.5y + 0.5z = 1$, which means that $y = 3$. The values for y and z are substituted into row 1, which has $2x + y - z = 8$, and so $x = 2$.

Although this algorithm works, there are some problems. The first is that it's possible to pass *singular* equations to the function – equations with no answers or an infinite number. Two examples:

$$\begin{aligned} x + y &= 1 \\ 2x + 2y &= 3 \end{aligned}$$

and

$$\begin{aligned} x + y &= 1 \\ 2x + 2y &= 2 \end{aligned}$$

The first set of equations define two lines that never intersect, while the second pair meets at an infinite number of points. gaussian() will print an error message in these cases and terminate.

A more common problem is when the set of equations are *nearly* singular. The calculations will involve the subtraction of two nearly identical numbers, and the resulting floating point error can destroy the accuracy of the result.

A simple solution, called *partial pivoting*, avoids this problem in *most* cases by rearranging the rows so that the the current row r contains a large value in cell $a_{r,r}$. It's beneficial that this value is large since it's used to calculate the multiples to subtract from later rows. If $a_{r,r}$ is large then there's less chance that this will introduce rounding errors.

The best value for $a_{r,r}$ is determined by examining all the numbers in column r. If the largest is in row z, then row r and z are swapped.

Consider the following situation when we're considering the first row of the matrix:

$$\begin{bmatrix} 0.02 & 0.01 & 0 & 0 & 0.02 \\ 1 & 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 4 \\ 0 & 0 & 100 & 200 & 800 \end{bmatrix}$$

The algorithm will employ $a_{1,1}$ to simplify the later rows, so partial pivoting examines all the values in column 1 to find the largest. That is $a_{2,1}$ in row 2, and so rows 1 and 2 are swapped:

$$\begin{bmatrix} 1 & 2 & 1 & 0 & 1 \\ 0.02 & 0.01 & 0 & 0 & 0.02 \\ 0 & 1 & 2 & 1 & 4 \\ 0 & 0 & 100 & 200 & 800 \end{bmatrix}$$

Elimination now proceeds, which subtracts multiples of row 1 from the lower rows, producing:

$$\begin{bmatrix} 1 & 2 & 1 & 0 & 1 \\ 0 & -0.03 & -0.02 & 0 & 0 \\ 0 & 1 & 2 & 1 & 4 \\ 0 & 0 & 100 & 200 & 800 \end{bmatrix}$$

For the next iteration, partial pivoting will look for the largest value for $a_{2,2}$, which will lead to rows 2 and 3 being swapped before elimination continues:

$$\begin{bmatrix} 1 & 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 4 \\ 0 & -0.03 & -0.02 & 0 & 0 \\ 0 & 0 & 100 & 200 & 800 \end{bmatrix}$$

The code for Gaussian elimination with partial pivoting:

```
def gaussian(arr):
  n = len(arr); xs = [0]*n
  for z in range(n-1):  # looking at row z
    # partial pivot
    maxVal = 0
    # look down col z for largest val, and save row val
    for r in range(z, n):
      if abs(arr[r][z]) > maxVal:
        maxRow = r
        maxVal = abs(arr[r][z])
    if maxRow != z:
      # exchange z and maxRow rows
      arr[maxRow], arr[z] = arr[z], arr[maxRow]

    # reduce all later rows
    for r in range(z+1, n):
      pivot = arr[r][z]/arr[z][z]
      for s in range(z+1, n+1):
        arr[r][s] -= pivot*arr[z][s]
        # subtract from all values on row

  # back substitution; work up the rows
  for r in range(n-1, -1, -1):
    pivot = arr[r][n]
    if r != n-1:
      for s in range(r+1, n):
        pivot -= arr[r][s]*xs[s]
    xs[r] = pivot/arr[r][r]
  return xs
```

Listing 6.27. Gaussian elimination with partial pivoting

Listing 6.27 (gaussian.py) shows that it's easy to manipulate matrices without resorting to Python's numpy module. But if the matrices are large, or you require something more than basic Gaussian elimination, then numpy is the way to go. Its `polynomial` package supports an extensive range of curve fitting algorithms, including ones based on Chebyshev and Lagrange series (`https://numpy.org/doc/stable/reference/routines.polynomials.package.html#module-numpy.polynomial`).

Exercises

(1) With the help of Listing 6.27 we can write a more general least-squares curve fitting function that tries out several polynomials, and chooses the one with the best correlation coefficient. Write a function which does these tasks.

(2) An alternative linear regression method, called *Deming regression* (`https://en.wikipedia.org/wiki/Deming_regression`) finds the line that minimizes the squares of the *perpendicular* distances between the points and the line rather than the vertical distances. While the traditional least-squares method accounts for errors in the y values only, this technique handles errors in both the x and y data. The slope and y-intercept of the line are:

$$m = \frac{s_{yy} - s_{xx} + \sqrt{(s_{yy} - s_{xx})^2 + 4(s_{xy})^2}}{2s_{xy}} \quad \text{and} \quad b = \bar{y} - m\bar{x}$$

where

- $s_{xx} = \frac{1}{n-1}\sum(x - \bar{x})^2$
- $s_{yy} = \frac{1}{n-1}\sum(y - \bar{y})^2$
- $s_{xy} = \frac{1}{n-1}\sum(x - \bar{x})(y - \bar{y})$

Write a function that implements this technique.

6.9 Polygon Circumscribing

Circumscribe a triangle about a circle, another circle around the triangle, a square outside the circle, another circle outside the square, and so on, increasing the number of sides in the polygon at each iteration. The beautiful result is Fig. 6.16a.

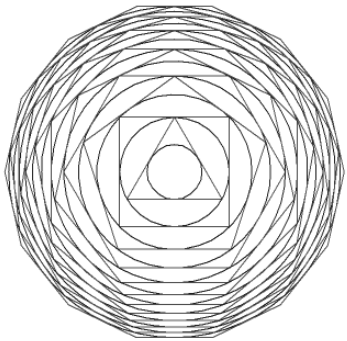
Fig. 6.16b simplifies this image to two circles and a (slightly rotated) equilateral triangle. The radii of the inner circle (the incircle) and the outer circle (the circumcircle) are related by the angle between the points where they touch the triangle – the inner circle touches half way along a side while the outer circle intersects the vertices. For our equilateral triangle, the angle between these is $\pi/3$, so we can write circumradius $\cdot \cos \pi/3 =$ inradius.

In general, if the regular polygon has n sides, then the angle will be π/n, and we'll denote the sides by $r(n-1)$ and $r(n)$, as in Fig. 6.17.

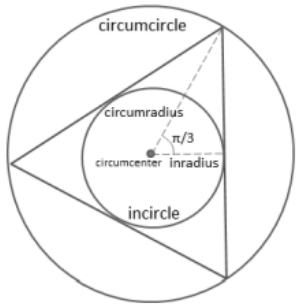
The first polygon, the triangle, requires $n = 3$, so we define $r(2) = 1$, which makes the inradius of the triangle's inner circle = 1.

The relationship between $r(n)$ and $r(n-1)$ is

$$\frac{r(n-1)}{r(n)} = \cos(\frac{\pi}{n}) \text{ or } r(n) = \frac{r(n-1)}{\cos(\pi/n)}, \quad r(3) = \frac{r(2)}{\cos(\pi/3)} = 2.$$



(a) Polygons and circles



(b) A triangle and circles

Figure 6.16. Polygon Circumscribing

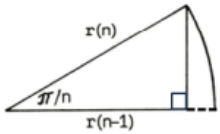


Figure 6.17. Generalizing the radius and angle

Therefore

$$\begin{aligned} r(n) &= \frac{1}{\cos(\pi/3)} * \frac{1}{\cos(\pi/4)} * \frac{1}{\cos(\pi/5)} * \cdots \\ &= \prod_{k=3}^{\infty} \frac{1}{\cos(\pi/k)} \end{aligned}$$

Fig. 6.16a is generated by using the circumcircle of an n-gon (e.g. the triangle) as the incircle for the next $(n+1)$-gon (a square). This is implemented in drawPolys.py. The most important line is:

```
r = r/math.cos(math.pi/n)
```

which converts the old r value (the circle's inradius) into the circumradius used for drawing the next regular polygon and its circumcircle.

Fig. 6.16a confirms that $r(n)$ is monotonically increasing, but also suggests that it's bounded. We can investigate this more easily without drawing the shapes, which simplifies the code to become Listing 6.28 (radius.py).

```
print("    n   r(n)")
for n in [10, 100, 1000, 10000,
    20000, 30000, 50000, 60000]:
  r = 1
  for i in range(3, n+1):
    r = r/math.cos(math.pi/i)
  print(f"{n:5d}  {r:.4f}")
```

Listing 6.28. Radii changes

```
> python radius.py
    n   r(n)
   10  5.4267
  100  8.2831
 1000  8.6572
10000  8.6957
20000  8.6979
30000  8.6986
50000  8.6992
60000  8.6993
```

This is certainly very suggestive that the limit is approximately 8.7, but of course we must develop some theory to decide.

A good introduction to this problem can be found in Chapter 18 of Cliford Pickover's *Keys to Infinity* [**Pic95**], or in the older, but still excellent, *Mathematics and the Imagination* by Kasner and Newman [**KN01**] (although there is a mistake in their calculations). A good online summary can be found at `https://mathworld.wolfram.com/PolygonCircumscribing.html`. The OEIS sequence for $r(n)$ is online at `https://oeis.org/A051762`, with the approximate limit of

8.700036625208194503222409859113004971193297949742892 09

The reciprocal of this constant is called the Kepler–Bouwkamp constant, and crops up during the generation of an infinity series of *inscribed* polygons, like those shown in Fig. 6.18. A fun exercise is to modify drawPolys.py to generate this image.

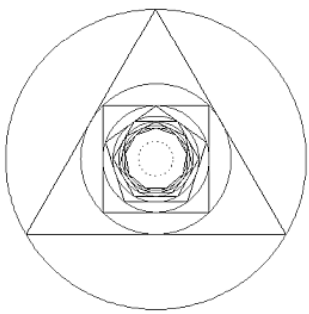


Figure 6.18. Inscribed polygons and circles

6.10 Archimedes' Integration of the Parabola

A great variety of problems can be reformulated as the determination of an area, or an integral as it's called in calculus.

Archimedes anticipated calculus by nearly 2000 years via the concepts of the infinitely small and exhaustion. He used these techniques on several area and volume problems, include the area of a circle, the surface area and volume of a sphere, the area of an ellipse, and the area under an arc of a parabola (`https://en.wikipedia.org/wiki/Quadrature_of_the_Parabola`). We'll reproduce part of his approach for that problem here.

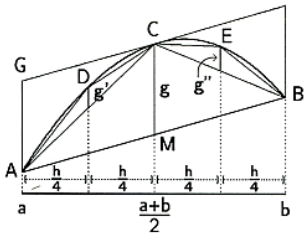


Figure 6.19. The area under a parabola

In Fig. 6.19, let P be the area of the parabolic segment cut off by the chord AB, and let $\triangle_1$ be the area of the triangle ABC. A and B are at equal distances from the vertical through C. We want to show that

$$P = \frac{4}{3}\Delta_1. \tag{6.27}$$

We see from Fig. 6.19 that $\triangle_1$ is the sum of the areas of the triangles CMA and CMB, which share a common base of length g. The sum of their heights is $h = b - a$, so $\triangle_1 = gh/2$. Let the equation of the parabola be $y = c_0 + c_1 x + c_2 x^2$, and a little computation shows that the distance g from M to C is

$$g = |c_2|\,(\frac{h}{2})^2.$$

Next we construct two smaller triangles ADC and CEB whose bases are chords of the parabola and whose vertices lie on the parabola vertically above the midpoints of the chords. It follows as above that the lengths g' and g'' satisfy

$$g' = g'' = |c_2|\,(\frac{h}{4})^2 = \frac{g}{4}.$$

The two triangles ADC and CEB can be divided into four triangles with bases g' and g'' of length $g/4$, and the sum of their altitudes is again h. Thus the sum $\triangle_2$ of their areas is $\triangle_1/4$.

We continue like this, halving the subintervals. At the n-th step we add 2^n new triangles whose areas add up to $\frac{1}{4}\triangle_{n-1}$, with the resulting polygons filling out more and more of the parabolic segment. Thus

$$P = \triangle_1 + \frac{\triangle_1}{4} + \frac{\triangle_1}{16} + \cdots.$$

Summing this infinite geometric series gives Equ. (6.27).

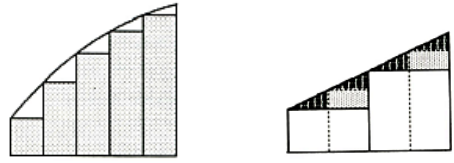
Archimedes also proved that the tangent at C is parallel to the chord AB, which has many consequences for the parabola's geometry.

For more on Archimedes and calculus, the paper "Did Archimedes do calculus?" by Jeff Powers is well worth a read (`https://maa.org/sites/default/files/images/upload_library/46/HOMSIGMAA/2020-Jeffery%20Powers.pdf`).

6.11 Numerical Integration

Let's look at some approximate formulae for the area A under a curve $y = f(x)$ between $x = a$ and $x = b$.

6.11.1 Rectangular Approximation. We can approximate A by rectangles whose height is the function value at the left endpoint of the base, as in Fig. 6.20a. This is clearly not the best choice for the heights but it will give a particularly transparent illustration of the extrapolation to the limit which we discuss in the next subsection.



(a) Initial rectangles

(b) Subdivided rectangles

Figure 6.20. Rectangular area approximation

Let R_n be the sum of the areas of the rectangles when we use n rectangles; for definiteness assume their widths are equal. Write

$$R_n = A + e_n^R \tag{6.28}$$

where e_n^R is the error in the approximation by n rectangles. Cut each subdivision in half. Then

$$R_{2n} = A + e_{2n}^R. \tag{6.29}$$

For large n there is a simple relation between e_n^R and e_{2n}^R. If we take a sufficiently small piece of any smooth curve and enlarge it, then the piece will look more and more like a straight line segment because the direction doesn't change much from one end of a short piece to the other, and direction remains the same under enlargement. Fig. 6.20b shows that, for straight lines, $e_{2n}^R = e_n^R/2$ exactly. For any smooth function we have

$$e_{2n}^R \approx e_n^R/2. \tag{6.30}$$

If $f(x)$ is increasing over part of the interval and decreasing over the rest of it, then the errors in the increasing part will be negative and the ones in the decreasing part will be positive, and it's possible that these will largely cancel out. After the subdivision the individual errors will be replaced by two errors whose sum is about half of the previous error but it could be that when we add all of these, the cancellation is less exact than before, and Equ. (6.30) may not be valid.

Our aim in this subsection and the next will be to derive formulae in which the main error term is eliminated. The validity of that computation isn't affected by the fact that for some special instances of f, a, and b the main error term has the value 0.

We shall compute more and more accurate approximations to A by bisecting the intervals we already have, and will refer to the doubling of the number of intervals as one step in the approximation process. Note that one of these steps involves evaluating and summing n new function values so that the amount of work doubles with each "step". Equ. (6.30) tells us that the error is multiplied by a factor of about 1/2 in each step, so we'll say that the rectangular approximation converges with a linear rate of 1/2.[1]

6.11.2 Trapezoidal Approximation. We approximate the area by a sum of trapezoids as in Fig. 6.21.

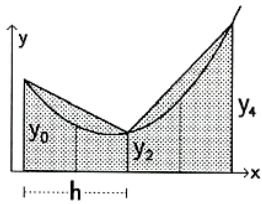


Figure 6.21. Trapezoidal area approximation

The formula expressing this is

$$T_n = h(\frac{1}{2}y_0 + y_2 + y_4 + \cdots + \frac{1}{2}y_{2n}),$$

where $y_0, y_1, y_2, ..., y_{2n}$ are the y-values at intervals of $\frac{1}{2}h$, starting from the left endpoint. (The notation is designed to make it easy to compare and combine the trapezoidal approximation with the midpoint approximation discussed next.)

This is implemented in Listing 6.29 (trapezoid.py).

[1] Our meaning of "linear" in this context is different from what algorithmic *linear time complexity*, $O(n)$. In fact, the time complexity of our algorithm is exponential since each bit of the result requires twice as much work as the previous one.

```
def trapezoid(f, a, b, n=500):
  h =(b - a)/n
  tot = 0.5 *(f(a) + f(b))
  for i in range(1,n):
    x = a + i*h
    tot += f(x)
  return tot*h
```

Listing 6.29. Summing trapezoids

```
>>> from trapezoid import *
>>> import math
>>> trapezoid(math.sin,0,math.pi,20)
1.995885972708715
>>> trapezoid(math.sin,0,math.pi,50)
1.9993419830762615
>>> trapezoid(math.sin,0,math.pi,1000)
1.9999983550656624
```

The tests above involving $\int_0^\pi \sin(x)dx$, which should return 2, show that this method needs a lot of data points to start producing accurate answers.

We're now ready to consider how the error in the trapezoidal rule changes when we double the number of trapezoids. Let T_n be the area under the polygon when we have n trapezoids. Write

$$T_n = A + e_n^T. \tag{6.31}$$

Our convention is that the area between a chord and the curve is negative if the curve is below the chord. Hence $e_n^T = -$(sum of the areas between the sides of the polygon and the curve). Similarly, we write

$$T_{2n} = A + e_{2n}^T. \tag{6.32}$$

In case the curve is a parabola, Fig. 6.19 and Archimedes' Theorem let us say exactly how e_n^T changes when we subdivide the interval. The difference between the area P bounded by the whole arc and its chord, and its two halves and their chords is $\triangle_1$. From Equ. (6.30) of the last section, this means that the sum of the two small areas is $\frac{1}{4}$ of the large one. So we conclude that for the trapezoidal rule

$$e_{2n}^T \approx \frac{1}{4} e_n^T. \tag{6.33}$$

In other words, the trapezoidal rule converges linearly with the rate $\frac{1}{4}$.

6.11.3 Midpoint Approximation. If we take the height of each rectangle to be the function value for the *midpoint* of the base as in Fig. 6.22, then the area approximation is better than the one in Sec. 6.11.1.

The following formula expresses this approach:

$$M_h = h(y_1 + y_3 + y_5 + \cdots),$$

where we use the same notation as in the Trapezoidal rule. In Fig. 6.22, we've drawn the trapezoids bounded on top by the tangents at their midpoints. With our sign convention, the error is the negative of the area between the tangent and the curve.

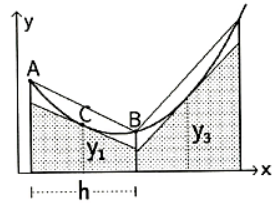


Figure 6.22. Midpoint trapezoidal area approximation

If the curve is a parabola then Archimedes' Theorem lets us determine the error exactly. For a parabola the area between the chord AB and the curve is $\frac{4}{3}\triangle_1$ where $\triangle_1$ is the area of triangle ABC. The area bounded by the chord AB, the tangent at C and the verticals through A and B is $2\triangle_1$. Thus we get:

> Two-thirds of the area of the quadrilateral bounded by the chord AB, the verticals at A and B and the tangent at C is between the parabola and the chord, and one-third is between the parabola and the tangent.

We conclude that for a smooth curve the error in the midpoint rule is approximately half the error in the trapezoidal rule and has the opposite sign. In particular, the midpoint rule converges linearly, with a convergence factor of $\frac{1}{4}$.

6.11.4 Simpson's Rule. What we have learned about the errors in the midpoint and the trapezoidal rules tells us that if we form the weighted mean of the midpoint and the trapezoidal approximations with weights $\frac{2}{3}$ and $\frac{1}{3}$, we get an approximation which is exact for arcs of parabolas. This approximation is *Simpson's rule* (`https://en.wikipedia.org/wiki/Simpson%27s_rule`). The approximation for the area over an interval of length h is $\frac{h}{6}(y_0 + 4y_1 + y_2)$, and for n equal intervals it becomes

$$S_n = \frac{h}{6}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{2n-2} + 4y_{2n-1} + y_{2n}). \qquad (6.34)$$

From a naive point of view it seems unreasonable to assign to, say, the 7th function value twice the weight of the 6th. Why would one value in the middle of the range be twice as important as the next one? Simpson's rule assigns the weights $\frac{2}{3}$ and $\frac{4}{3}$ to consecutive y-values to approximate the fitting of arcs of parabolas through triplets of equally spaced points rather than the straight line segments used by the trapezoidal and midpoint methods. This approach only works if the curve is smooth, so for irregular curves Simpson's rule will probably be less accurate than the trapezoidal rule.

Simpson's rule is implemented in Listing 6.30 (`simpson.py`).

```
def simpson(f, a, b, n=500):
  if n % 2 != 0:
    print(f"n={n} must be even
        integer!")
    n = n+1
  h = (b - a)/n
  sum1 = 0
  for i in range(1, n//2 + 1):
    sum1 += f(a + (2*i-1)*h)
  sum2 = 0
  for i in range(1, n//2):
    sum2 += f(a + 2*i*h)
  return (b-a)/(3*n)*(f(a) + f(
      b) + 4*sum1 + 2*sum2)
```

```
>>> from simpson import *
>>> import math
>>> simpson(math.sin,0,math.pi,20)
2.0000067844418012
>>> simpson(math.sin,0,math.pi,50)
2.000000173253138
>>> simpson(math.sin,0,math.pi,1000)
2.0000000000010814
```

Listing 6.30. Simpson's Rule

The same tests as before, involving $\int_0^\pi \sin(x)dx$ which should return 2, produce much better results than the trapezoid rule, offering greater accuracy with fewer data points.

6.11.5 Comparing all the methods. tramisim1.py computes π by computing four times the area under the positive quadrant of the unit circle. The trapezoid, midpoint, and Simpson's approximations calculations are combined rather than separated into individual functions. The code stops when the change in the value simp given by Simpson's rule is less than `eps*simp`. With input $a = 0$, $b = 1$, and eps $= 0.00001$ we get

```
   trapezoid       midpoint        simpson
2.0000000000  3.4641016151  2.9760677434
2.7320508076  3.2593673286  3.0835951549
2.9957090681  3.1839292206  3.1211891698
3.0898191444  3.1566869313  3.1343976690
3.1232530378  3.1469518079  3.1390522179
3.1351024229  3.1434914027  3.1406950761
3.1392969128  3.1422646720  3.1412754189
3.1407807924  3.1418303735  3.1414805131
3.1413055830  3.1416767225  3.1415530093
3.1414911527  3.1416223804  3.1415786378
```

This is a little disappointing since Simpson's rule is not much more accurate than the other two. The reason is this: we argued that Simpson's rule is particularly good because it integrates arcs of parabolas exactly, and most curves can be approximated very closely by such arcs. But as $x \to 1$, the tangent of the circular arc approaches the vertical. Since the graph of a quadratic function is nowhere vertical, Simpson can not closely approximate the circle near $x = 1$.

We can instead determine the area of the circle without approaching the vertical part by calculating 12 times the area of a 30° segment of the unit circle. One

such segment is the area bounded by the y-axis, the arc of the unit circle between $x = 0$ and $x = 0.5$ and the line $y = \sqrt{3}x$. The following output was obtained using tramisim1.py, but with $f()$ replaced by

```
def f(x):
  return 12*(math.sqrt(1-x*x)-math.sqrt(3)*x)
```

and input $a = 0$, $b = 0.5$ and eps $= 1E - 9$.

```
    trapezoid        midpoint         simpson
 3.0000000000    3.2113988080    3.1409325386
 3.1056994040    3.1594684463    3.1415454322
 3.1325839251    3.1460923941    3.1415895711
 3.1393381596    3.1427196081    3.1415924586
 3.1410288839    3.1418745201    3.1415926414
 3.1414517020    3.1416631282    3.1415926528
 3.1415574151    3.1416102728    3.1415926535
```

This time Simpson's rule is much more accurate than the other two, with its error decreasing by a factor of 1/16 at each step.

6.12 Richardson Extrapolation

All of the integration methods in the previous section had the following property: after a few steps, halving the intervals approximately multiplied their error by a factor which did not depend on the integral.

This means that if the error is multiplied by, say, 1/2 in each step, then we can calculate the error value, and hence obtain the exact integral value by considering two consecutive steps.

This extrapolation technique was first proposed by Alexander Craig Aitken, and applied to numerical integration problems by Lewis Fry Richardson. Let's apply it to two of the integration methods from the last section.

a) *The rectangular rule.* From the formulae of Sec. 6.11.1, we find that Richardson extrapolation gives

$$A \approx 2R_{2n} - R_n = h(y_1 + y_3 + y_5 + \cdots + y_{2n-1}).$$

Here $h = (b - a)/n$. This is the midpoint rule approximation M_n.

b) *The trapezoidal rule.* From the formulae of Sec. 6.11.2, the error becomes about 4 times smaller when we halve the intervals. Extrapolation gives

$$A \approx \frac{4T_{2n} - T_n}{3} = \frac{h}{6}(y_1 + 4y_1 + 2y_2 + 4y_3 \cdots + 4y_{2n-1} + y_{2n}),$$

which is Simpson's rule S_n (see Sec. 6.11.4).

These two formulae obtained with Richardson extrapolation improve the integration accuracy, but this isn't the case if we extrapolate the midpoint rule (Sec. 6.11.3). We get $A \approx (4M_{2n} - M_n)/3$, which can be written as $2S_{2n} - S_n$, which is less accurate than Simpson's rule with the same number of data points.

6.13 Romberg Integration

The Romberg algorithm produces a triangular array of numerical estimates of the definite integral $\int_a^b f(x)dx$ written as:

```
R(0,0)
R(1,0) R(1,1)
R(2,0) R(2,1) R(2,2)
R(3,0) R(3,1) R(3,2) R(3,3)
  :
R(n,0) R(n,1) R(n,2) R(n,3) ... R(n,n)
```

The first column are estimates of the integral obtained by the Trapezoid rule with decreasing step sizes. $R(n,0)$ is the result of applying the Trapezoid rule with 2^n equal subintervals. The first of them, $R(0,0)$, uses just one trapezoid:

$$R(0,0) = \frac{1}{2}(b-a)(f(a)+f(b))$$

$R(n,0)$ can be obtained from $R(n-1,0)$ by:

$$R(n,0) = \frac{1}{2}R(n-1,0) + h\sum_{k=1}^{2^{n-1}} f(a+(2k-1)h) \tag{6.35}$$

where $h = (b-a)/2^n$ and $n \geq 1$.

The second and successive columns in the array are generated using Richardson extrapolation (the details are explained in the next subsection):

$$R(n,m) = R(n,m-1) + \frac{1}{4^{m-1}}(R(n,m-1) - R(n-1,m-1)) \tag{6.36}$$

with $n \geq 1$ and $m \geq 1$.

One way to generate the triangular array is to compute several terms in the first column, $R(0,0)$ up to $R(n,0)$ say, and then use Equ. (6.36) to construct columns $1, 2, ..., n$. Another way, used in Listing 6.31 (romberg.py), is to compute the array row by row. Note, for example, that $R(1,1)$ can be calculated as soon as $R(1,0)$ and $R(0,0)$ are available.

The number of subintervals is 2^n, so a small n is hardwired into the code. A more sophisticated function would include a stopping test to terminate the calculation as soon as the error reaches a certain tolerance.

```
def romberg(f, a, b):
  r = [0]*(SZ)
  for i in range(SZ):
    r[i] = [0]*SZ
  h = b - a
  r[0][0] = 0.5 * h *(f(a) + f(b))
  # computes row-by-row
  for i in range(1, SZ):
     h *= 0.5
```

```
    tot = 0; k = 1
    while k <= math.pow(2,i)-1:
      tot += f(a + k*h)
      k += 2
    r[i][0] = 0.5 * r[i-1][0] + tot * h
      # Trapezoid rule for first column
    for j in range(1, i+1):
      r[i][j] = r[i][j-1] + \
           (r[i][j-1] - r[i-1][j-1]) /(math.pow(4,j)-1)
      # Richardson extrapolation for other columns
for i in range(SZ):  # print triangular array
  for j in range(i+1):
    print(f"{r[i][j]:13.10f}", end = '')
  print()
print()
```

Listing 6.31. Implementing Romberg integration

As an example, we'll use Listing 6.31 to calculate π by obtaining a numerical approximation for the integral

$$\int_0^5 (12\sqrt{1-x^2} - \sqrt{3}x)\,dx$$

This function was used back in Sec. 6.11.5. The calculated area is 12 times the area of a 30° sector of a unit circle.

The following triangular array is generated:

```
3.0000000000
3.1056994040 3.1409325386
3.1325839251 3.1415454322 3.1415862917
3.1393381596 3.1415895711 3.1415925137 3.1415926125
3.1410288839 3.1415924586 3.1415926511 3.1415926533 3.1415926535
```

The first column corresponds to a Trapezoid rule calculation with one trapezoid, I_1. Going down the column we encounter I_2, I_4, I_8, and finally I_{16}. So, to arrive at our final result with 10 digits of accuracy (the rightmost value in the last row) we only need to carry out 17 function evaluations (16 trapezoids means 17 points).

Well, not quite, since our implementation is somewhat wasteful: it throws away older trapezoid calculations when producing new ones (e.g., the function evaluations used to produce I_8 are discarded when evaluating I_{16}). As a result, we actually carry out $2 + 3 + 5 + 9 + 17 = 36$ function calls.

In any case, this is much better than using the Trapezoid rule on its own where the same accuracy may require many 1000s of data points. Even Simpson's rule needs roughly 100 points to produce an estimate with an absolute error of 10^{-10}.

For comparison, we obtain π by computing 4 times the area under a quarter circle using `4*sqrt(1-x*x)`. With input $a = 0$, $b = 1$, we get

```
2.0000000000
2.7320508076 2.9760677434
2.9957090681 3.0835951549 3.0907636490
3.0898191444 3.1211891698 3.1236954374 3.1242181642
3.1232530378 3.1343976690 3.1352782356 3.1354620895 3.1355061834
```

Our sophisticated processing produces little improvement here because the vertical tangent at the end of the data range violates the method's assumption that f possesses derivatives across the entire integration range.

Finally, we apply Romberg's method to the computation of $\ln x$ discussed in Sec. 6.3. With $a = 1$, $b = 2$ we get

```
0.7500000000
0.7083333333 0.6944444444
0.6970238095 0.6932539683 0.6931746032
0.6941218504 0.6931545307 0.6931479015 0.6931474776
0.6933912022 0.6931476528 0.6931471943 0.6931471831 0.6931471819
```

If we focus on the first column, which corresponds to the Trapezoid method, then the result is wrong in the 5th digit ($\log(2) = 0.69314718055994$) in row 5 (i.e. when $2^4 = 16$ intervals are used). However, the Romberg method (which generates the other columns) gives an answer at the end of row 4 which is correct to 10 digits!

6.13.0.1 Richardson extrapolation for Romberg. We begin with an Euler-Maclaurin equation [**YG88**] which expresses the error in the Trapezoid method over 2^{n-1} subintervals:

$$\int_a^b f(x)dx = R(n-1,0) + a_2h^2 + a_4h^4 + a_6h^6 + \cdots \tag{6.37}$$

$h = (b-a)/2^{n-1}$ and the coefficients a_i only depend on f, not on h.

$R(n-1,0)$ is a typical element in the first column of the Romberg array – one of the trapezoidal estimates of the integral. Note that the error is expressed in powers of h^2, and the error series is $O(h^2)$.

Richardson extrapolation is employed to reduce this error by combining $R()$ terms.

If we replace n with $n+1$ and h with $h/2$ in Equ. (6.37), we have

$$\int_a^b f(x)dx = R(n,0) + \frac{1}{4}a_2h^2 + \frac{1}{16}a_4h^4 + \frac{1}{64}a_6h^6 + \cdots \tag{6.38}$$

Subtracting Equ. (6.37) from 4 times Equ. (6.38) produces

$$\int_a^b f(x)dx = R(n,1) - \frac{1}{4}a_4h^4 - \frac{5}{16}a_6h^6 - \cdots \tag{6.39}$$

where

$$R(n,1) = R(n,0) + \frac{1}{3}(R(n,0) - R(n-1,0)) \quad \text{for } n \geq 1$$

This is the first case ($m = 1$) of the extrapolation formula Equ. (6.36) and $R(n,1)$ should be considerably more accurate than $R(n,0)$ or $R(n-1,0)$ because its error formula begins with an h^4 term.

This process can be repeated using a slightly modified Equ. (6.39), with n replaced by $n-1$ and h replaced by $2h$. The two equations are combined to eliminate the h^4 term:

$$\int_a^b f(x)dx = R(n,2) + \frac{1}{4^3}a_6h^6 + \frac{21}{4^5}a_8h^8 + \cdots \tag{6.40}$$

where

$$R(n,2) = R(n,1) + \frac{1}{15}(R(n,1) - R(n-1,1)) \quad \text{for } n \geq 2$$

which agrees with Equ. (6.36) when $m = 2$. $R(n,2)$ is an even more accurate approximation to the integral because its error series begins with an h^6 term.

Exercises

(1) Let $f(x) = 2^n$. Approximate $\int_0^4 f(x)dx$ by the Trapezoid rule using points at $x = 0$, 2, and 4. Repeat by using points $x = 0$, 1, 2, 3, and 4. Now apply Romberg to obtain a better approximation. Note: for the first two parts of this question, consider using Listing 6.29.

(2) We plan to use the Romberg method to estimate $\int_0^1 \sqrt{x}\cos x \; dx$. Will the method work? Will it work well? Explain.

6.14 Problems of Pursuit

6.14.1 Pirates Ahoy. The mathematical study of pursuit began with a problem devised by the French mathematician Pierre Bouguer in 1732. He considered the case of a pirate ship pursuing a merchant vessel, as illustrated in Fig. 6.23.

The pirate ship, P, and the merchant vessel, M, are at $(0,0)$ and $(b,0)$ respectively at time $t = 0$. The merchant vessel moves at speed V_m up the line $x = b$, while the pirate ship travels at speed V_p along a curved path which ensures that the pirates are always heading directly towards the merchant. What is the equation of the curved path?

We assume that at time t, P is located at (x,y), and M has reached $(b, V_m t)$. P is pointing along a tangent to the curve which intersects M. dy/dx at (x,y) is:

$$\frac{dy}{dx} = \frac{V_m t - y}{b - x}$$

We also know that P has sailed along the curve a distance of $V_p t$. This arc-length can be defined as:

$$V_p t = \int_0^x \sqrt{1 + (\frac{dy}{dz})^2}\, dz.$$

We solve these two equations for the variable t:

$$\frac{y - (x - b)}{V_m} \cdot \frac{dy}{dx} = \frac{1}{V_p} \int_0^x \sqrt{1 + (\frac{dy}{dz})^2}\, dz$$

Set $w(x) = dy/dx$ and differentiate both sides with respect to x to obtain a first-order differential:

$$(x - b)\frac{dw}{dx} = -\frac{V_m}{V_p}\sqrt{1 + w^2}$$

Using appropriate initial values for x and $w = dy/dx$ when $t = 0$, we solve the equation:

$$\frac{dy}{dx} = w(x) = \frac{1}{2}\left[(1 - \frac{x}{b})^{-\frac{V_m}{V_p}} - (1 - \frac{x}{b})^{\frac{V_m}{V_p}}\right]. \tag{6.41}$$

We're only really interested in the $\frac{V_m}{V_p} \leq 1$ case, when the pirates are sailing at the same speed or faster than the merchant. Even so, there's quite a few more steps before we get the pursuit equation:

$$y = \frac{n}{1 - n^2} b + \frac{1}{2}(b - x)\left[\frac{(1 - x/b)^n}{1 + n} - \frac{(1 - x/b)^{-n}}{1 - n}\right], \quad n = \frac{V_m}{V_p} \tag{6.42}$$

If you want the details, we recommend Chapter 1 of *Chases and Escapes* by Paul J. Nahin [**Nah07**].

This version of the equation only applies when V_p is faster than V_m (i.e. $\frac{V_m}{V_p} < 1$), since $n = 1$ leads to a division by zero.

One thing we'll check using our simulation is where the pirate ship intercepts the merchant. This occurs when $x = b$, which simplifies Equ. (6.42) to:

$$y = \frac{n}{1 - n^2} b$$

When $V_p = V_m$ (i.e. $n = 1$), Equ. (6.42) is simplified, and exhibits some interesting aspects of the problem:

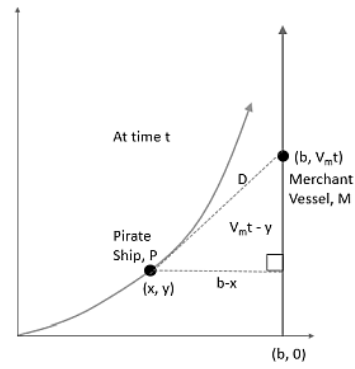


Figure 6.23. Pirates in pursuit

$$y = \frac{1}{2} b\left[\frac{1}{2}(1 - \frac{x}{b})^2 - \ln(1 - \frac{x}{b})\right] - \frac{1}{4} b \tag{6.43}$$

The equation is quadratic with a small logarithmic term. When the problem was discussed by George Boole (1815–1864) in his *Treatise on Differential Equations*

he mistakenly stated that the curve was a parabola. Plots of the curve emphasize that it can't be parabolic since it never crosses the Merchant's $x = b$ line: asymptotic behavior that's impossible for a parabola.

Equ. (6.43) also shows that the pirates will *never* capture the merchant vessel in the $n = 1$ case; after a long period, the pirates will have moved into a position a distance D behind the merchant and will stay there. The value can be calculated:

$$\begin{aligned} D^2 &= (V_m t - y)^2 + (b - x)^2 \\ &= (b - x)^2 \left[1 + \left(\tfrac{V_m t - y}{b - x}\right)^2\right] \\ &= (b - x)^2 \left[1 + (\tfrac{dy}{dx})^2\right] \end{aligned}$$

We have Equ. (6.41) for $\frac{dy}{dx}$, which we can substitute in:

$$\begin{aligned} D^2 &= (b - x)^2 \left[1 + \tfrac{1}{4}\left\{(1 - \tfrac{x}{b})^{-1} - \left(1 - \tfrac{x}{b}\right)\right\}^2\right] \\ &= \tfrac{b^2}{4}(1 - \tfrac{x}{b})^2 \left[4 + \left\{(1 - \tfrac{x}{b})^{-1} - \left(1 - \tfrac{x}{b}\right)\right\}^2\right] \\ &= \tfrac{b^2}{4}(1 - \tfrac{x}{b})^2 \left[4 + (1 - \tfrac{x}{b})^{-2} - 2 + (1 - \tfrac{x}{b})^2\right] \\ &= \tfrac{b^2}{4}\left[2(1 - \tfrac{x}{b})^2 + 1 + (1 - \tfrac{x}{b})^4\right] \end{aligned}$$

As $t \to \infty$, $x \to b$, and so:

$$\begin{aligned} \lim_{x \to b} D^2 &= \tfrac{b^2}{4}[0 + 1 + 0] \\ D &= \tfrac{b}{2} \end{aligned}$$

The separation is half the original distance between the pirate ship and merchant.

We'll model this pursuit using Python turtles (pirates.py). A `vessel` turtle starts at (100,0) and moves at a constant rate up the x-axis, taking a fixed number of steps on each iteration. The `pirate` turtle begins at the origin and moves a step during each iteration, always pointing at the merchant. This 'pointing' behavior is simple to implement using functions from the turtle module:

```
vesPos = vessel.pos()
move(pirate, vesPos, pirateStep)
```

with:

```
def move(t, pos, step):
  angle = t.towards(pos)
  t.setheading(angle)   # Set the direction to this angle
  t.fd(step)
```

The code relies on the Turtle towards() and setheading() functions which gets the angle between two positions and makes the turtle point at that angle, to face the other turtle. The complete code can be found in pirates.py.

Fig. 6.24 shows an execution of the program when the pirate step is 2.5 (which is greater than the merchant's step of 2).

The values are the number of steps needed to reach those points on the curves. The program terminates after printing:

```
No. steps: 109
Vessel pos: (100.00,218.00)
Pirate pos: (100.00,217.49)
Dist apart: 0.51
```

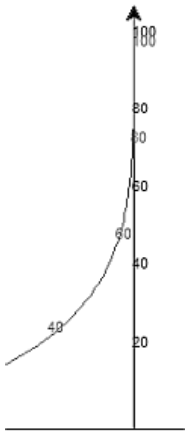


Figure 6.24. Pirates catch up

The pirate 'captures' the merchant when it is sufficiently close, and so the ships positions don't match exactly.

Table 6.4 compares the simulated capture positions with values obtained from the equation $\frac{n}{1-n^2}b$, where $n = \frac{V_m}{V_p}$. In our case, $b = 100$, and the velocities are modeled by step sizes, with the vessel's step = 2.

Pirate Step	Capture Pos.	n	$nb/(1-n^2)$
4	66	2/4	67
3.5	84	2/3.5	84
3	118	2/3	119
2.5	218	2/2.5	222
2.25	412	2/2.25	423

Table 6.4. Simulated and calculated capture positions

When the pirate step is set to 2 (the same as the merchant) then the pirate can't catch it. The code will keep looping until a maximum number of steps is reached. Fig. 6.25 shows the simulation for this case, with the output:

```
stopped after max steps: 250
Vessel pos: (100.00,500.00)
Pirate pos: (99.99,450.50)
Dist apart: 49.50
```

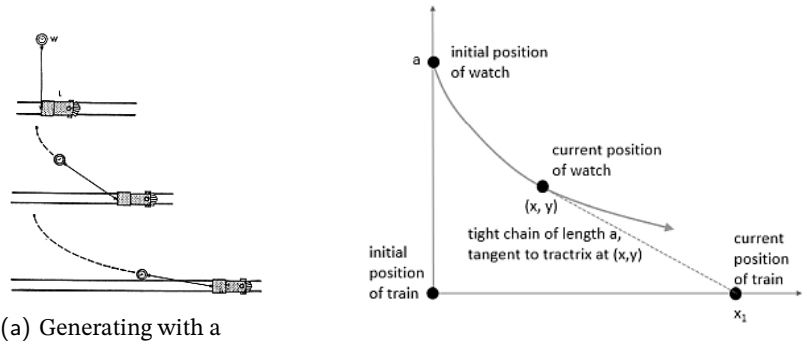
The distances between the step numbers printed in Fig. 6.25 are a rough indication that the separation of the two vessels is becoming constant. The final distance is 49.50, close to the theoretical answer $\frac{b}{2} = 50$.



Figure 6.25. Pirates do not catch up (Rotated Image)

6.14.2 The Tractrix. Forty years before Bouguer's pirates, another sort of pursuit curve was introduced by Claude Perrault, and later studied by Isaac Newton, and Christiaan Huygens who named it the tractrix.

Perrault's problem is illustrated in Fig. 6.26a: a watch-on-a-chain is placed on a table with the chain (of length a) pulled taut. The watch is initially resting on the y-axis, and the end of the chain is at the origin. If the end is pulled along the x-axis, the watch will be dragged along a path. What is its equation?



(a) Generating with a train

(b) As a graph

Figure 6.26. The tractrix

Using a toy train to do the pulling is optional, but Fig. 6.26a was too nice not to use; it comes from the excellent *Mathematics and the Imagination* by Edward Kasner and James R. Newman [**KN01**]; the illustrator was Rufus Isaacs. A more serious diagram appears in Fig. 6.26b.

The chain is always tight, which means that it always forms a tangent to the path, and has constant length a. The tangent that intersects the x-axis at $x = x_i$ is

$$\frac{dy}{dx} = \frac{y}{x - x_i}$$

Also, from Pythagoras:

$$(x - x_i)^2 + y^2 = a^2$$

Combining these two to solve for $(x - x_i)$, we get

$$\frac{y^2}{(dy/dx)^2} + y^2 = a^2$$

or

$$\left[\frac{y}{dy/dx}\right]^2 = a^2 - y^2$$

Taking the positive square root of both sides, and remembering that dy/dx is negative, we get:

$$-\frac{y}{dy/dx} = \sqrt{a^2 - y^2}$$

Separating the variables:

$$dx + \frac{\sqrt{a^2 - y^2}}{y}\, dy = 0$$

This can be integrated, when $x = 0, y = a$, making:

$$x = a \ln\left(\frac{a + \sqrt{a^2 - y^2}}{y}\right) - \sqrt{a^2 - y^2}$$

Although, matplotlib is wonderful, the quickest way to visualize a curve is probably by using an online tool such as Desmos (`https://www.desmos.com/calculator`). Fig. 6.27 shows the equation when $a = 2$.

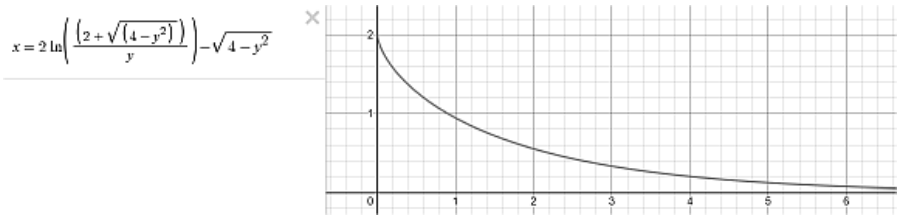


Figure 6.27. The Tractrix starting at $a = 2$

The tractrix can be extended along the negative x-axis by starting the train in reverse at the origin. The resulting line can be rotated around the x-axis to create a *pseudosphere* (see Fig. 6.28).

A pseudosphere of radius a has a constant negative Gaussian curvature of $\frac{-1}{a^2}$. Its name comes from analogy with a sphere of radius a, which has a positive Gaussian curvature of $\frac{1}{a^2}$. The term was coined by Eugenio Beltrami in his 1868 paper on models of hyperbolic geometry.

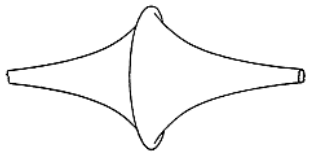


Figure 6.28. The pseudosphere

Perhaps the most mind-bending aspect of the shape is that its volume and surface area are finite despite the shape extending out to ± infinity along the x-axis.

One feature of the tractrix is that the area between the curve and its asymptote (the x-axis) is $\frac{\pi a^2}{2}$. This result can be obtained using integration, but a simpler approach is to employ Mamikon Mnatsakanian's visual calculus (`https://en.wikipedia.org/wiki/Visual_calculus`). Since all the tangents are the same length (a), it's possible to divide the area into equal length segments (see Fig. 6.29). These segments can be rearranged to form a quarter circle of radius a. If the curve extends along the negative x-axis, then the area doubles to $\frac{\pi a^2}{2}$.

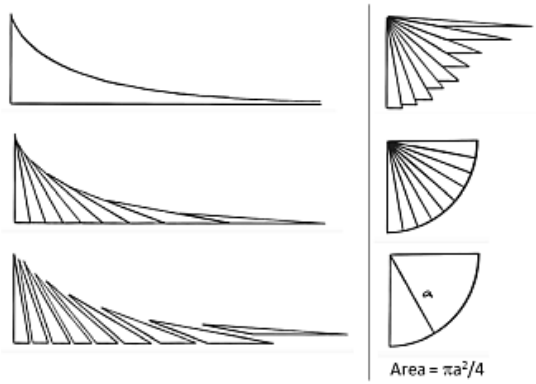


Figure 6.29. Using visual calculus for the tractrix area

An informative video on the tractrix, called "How to turn a circle inside out", with an emphasis on calculating its area using this approach, can be found at `https://www.youtube.com/watch?v=nQ2PeqGkQfk`.

Our tractrix simulation will use turtles again – this time called `man` and `watch` (see tractrix.py). The man will move along the x-axis in steps of 2, and the watch will enforce the tractrix properties:

(1) The distance (DIST = 150) between the watch and man never changes.

(2) The watch always faces the man.

This is implemented in tractrix.py using similar code as for the pirates curve:

```
man.forward(STEP)
manPos = man.pos()
watchPos = watch.pos()
watchStep = math.dist(manPos, watchPos) - DIST
       # stay at DIST distance from man
move(watch, manPos, watchStep)
```

move() is the same as earlier – it sets the heading of the watch and moves it forward the requested amount. This step is calculated so that the watch will end up DIST units away from the man.

Fig. 6.30 shows the graph generated by tractrix.py. Tangent lines are drawn every 10th step in order to see how the curve could be cut up for the visual calculus area solution.

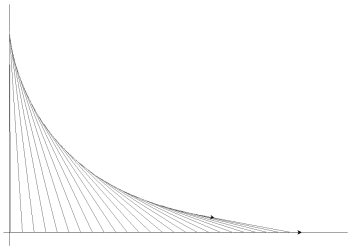


Figure 6.30. Turtles tractrix

6.14.3 Cyclic Pursuit. In the May 1877 issue of *Nouvelle correspondence mathématique*, Édouard Lucas posed the following problem:

> Three dogs are placed at the vertices of an equilateral triangle; they run one after the other. What is the curve described by each of them?

Nahin has a fascinating chapter on these cyclic pursuit forms, including some historical digging that reveals that Lucas wasn't the first to propose such problems [**Nah07**]. Lucas usually gets the credit, although Martin Gardner in his "Mathematical Games" column in *Scientific American* (July 1965) arguably made the problem famous. Rather than a triangle and dogs, Gardner places four bugs at the corners of a square, and asks: (a) what path is traveled by each bug? and (b) how far does each bug travel until they meet at the center of the square?

The problem undoubtedly got a boost because the editors of *Scientific American* pictured the bugs' journeys on that month's cover (which is reproduced in Fig. 6.31 by our mice.py). The original can be viewed at `https://www.scientificamerican.com/archive/issues/1965/`, and the Gardner column is reprinted as Chapter 24: 'Op Art' in *The Sixth Book of Mathematical Games from Scientific American* [**Gar71**].

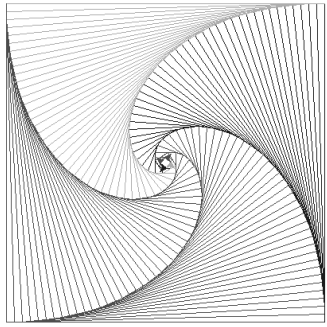


Figure 6.31. Four bugs meet

The geometry of n bugs in cyclic pursuit is represented in Fig. 6.32 for the case of two adjacent bugs, B1 and B2, located at the corners of an n-gon centered at C.

To simplify matters, bug B1 starts on the horizontal axis, and bug B2 is positioned above it at an angle of $2\pi/n$ relative to C. All the bugs move with a velocity V, and adjust their headings to keep facing their neighbor – B1 will head towards B2, B2 will head towards B3, and so on. We'll also assume that every bug starts at a distance r_0 from C.

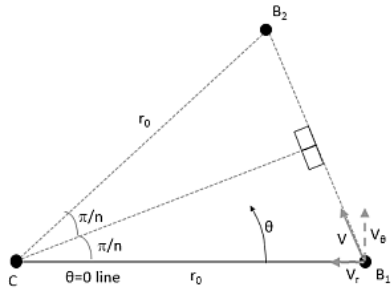


Figure 6.32. Two bugs in the n-bug problem

We can resolve the V speed into a radial component V_r directed inward towards C, and a transverse component V_θ perpendicular to V_r. From Fig. 6.32, these speed components are:

$$V_r = \frac{dr}{dt} = -V \sin(\frac{\pi}{n})$$

and

$$V_\theta = r\frac{d\theta}{dt} = V \cos(\frac{\pi}{n})$$

The minus sign in the V_r equation indicates that the velocity is directed towards C, causing the radius to decrease over time. Also, the bugs will always be located at the vertices of a regular (but shrinking) n-gon, so the central angle between adjacent bugs remains as $2\pi/n$.

We can link the two derivatives using the chain rule:

$$\frac{d\theta}{dt} = \frac{d\theta}{dr} \cdot \frac{dr}{dt}$$

Substitute in the dr/dt and $d\theta/dt$ velocity equations:

$$\frac{V}{r}\cos(\frac{\pi}{n}) = \frac{d\theta}{dr} \cdot \left(-V \sin(\frac{\pi}{n})\right)$$

This becomes

$$\frac{dr}{d\theta} = -r\frac{\sin(\pi/n)}{\cos(\pi/n)} = -r \tan(\frac{\pi}{n})$$

or

$$\frac{dr}{r} = -\tan(\frac{\pi}{n})\, d\theta$$

The integral is a logarithmic spiral using polar coordinates:

$$r(\theta) = r_0 \exp(-\theta \tan(\frac{\pi}{n}))$$

For Gardner's four-bug problem ($n = 4$), it reduces to:

$$r(\theta) = r_0 e^{-\theta}$$

Since the bugs' speed is constant (V), it's easy to calculate the total time T, from the start of the pursuit ($r = r_0$) until the bugs collide at C ($r = 0$):

$$T = \int_0^T dt = \int_{r_0}^0 \frac{dr}{(dr/dt)} = \int_{r_0}^0 \frac{dr}{-V\sin(\pi/n)}$$

So:

$$T = \int_0^{r_0} \frac{dr}{V\sin(\pi/n)} = \frac{r_0}{V\sin(\pi/n)}$$

Notice that as $n \to \infty$, $T \to \infty$. $n \to \infty$ means that the regular n-gon gradually becomes indistinguishable from a circle, so the bugs will end up traveling in a circle rather than in a spiral towards C. In other words, as $n \to \infty$ the bugs will cease to collide, making the collision time $T = \infty$.

An expression for the total distance S traveled by each bug is $S = TV$, or

$$S = \frac{r_0}{\sin(\pi/n)}$$

In the Gardner problem, the square (a regular 4-gon) had a side length of one, which means $r_0 = 1/\sqrt{2}$ (the distance from the center to the corners). Thus, S with $n = 4$ is:

$$S = \frac{1/\sqrt{2}}{\sin\frac{\pi}{4}} = \frac{1/\sqrt{2}}{1/\sqrt{2}} = 1$$

mice.py utilizes four turtles (t1, t2, t3, and t4) located at the corners of a square of sides 400 (see Fig. 6.31). t1 faces t2, t2 faces t3, t3 faces t4, and t4 faces t1. Inside a loop each turtle is updated by calling move() to adjust its heading and make a step. For example, the code for t1 is:

```
pos2 = t2.pos()
move(t1, pos2, 10)
```

The loop stops when the turtles are 'close' to one another.

It's possible to obtain an estimate for S (the total distance moved), by keeping a record of the number of steps made by a bug, and multiply it to the step size (10). The output for mice.py is:

```
Travel Dist: 420
```

Theory dictates that the result should be 400 (the side length of the initial square).

Exercises

(1) **The Fox and the Hare**. A fox always moves towards a hare. The hare tries to dodge by always running at right angles. Investigate what happens if the fox starts at the origin, and the hare begins somewhere along the x-axis, heading North initially. Fix the speed of the hare, but experiment with various step sizes for the fox to see how it affects the capture.

(2) **The Dog and the Duck**. Arthur Hathaway posed the following puzzle in *American Mathematical Monthly*, Jan. 1920:

> A dog at the center of a circular pond makes straight for a duck which is swimming counterclockwise along the edge of the pond. If the rate of swimming of the dog is to the rate of swimming of the duck as $n : 1$, determine the equation of the curve of pursuit and the distance the dog swims to catch the duck.

Unfortunately, no exact answer to the first part is possible – the differential equation can't be expressed in terms of elementary functions. However, this doesn't prevent us from using turtles to draw the curve.

The turtle command needed to make the duck move in a circle is `circle(radius, turnAngle)`. It's also fun to change the on-screen turtle icons to resemble a dog and a duck. We've included `http://coe.psu.ac.th/~ad/explore/code/06_Numer/14_Pursuit/dog.gif` and `http://coe.psu.ac.th/~ad/explore/code/06_Numer/14_Pursuit/duck.gif` for this purpose, and you'll need `turtle.addShape()` *and* `turtle.shape()`. For example:

```
scr.addshape("dog.gif")
dog.shape("dog.gif")
```

For the math behind this problem, consult Chapter 2 of Nahin [**Nah07**]. It's also analyzed by Robert Ferréol at `https://mathcurve.com/courbes2d.gb/poursuite/poursuite.shtml`.

(3) **The Lady-in-the-Lake Problem**. As with the 4-bug problem, this is a Gardner puzzle, from his "Mathematical Games" column in *Scientific American* (November and December 1965). He wrote:

> A young lady was vacationing on Circle Lake, a large artificial body of water named for its precisely circular shape. To escape from a man who was pursuing her, she got into a rowboat and rowed to the center of the lake, where a raft was anchored. The man decided to wait it out on shore. He knew she would have to come ashore eventually; since he could run four times faster than she could row, he assumed that it would be a simple matter to catch her as soon as her boat touched the lake's edge.
> But the girl – a mathematics major at Radcliffe – gave some thought to her predicament. She knew that on foot she could outrun the man [which does raise the question of why such a smart lady got herself into this situation in the first place by rowing out into a lake!]; it was only necessary to devise a rowing strategy that would get her to a point on shore before he could get there. She soon hit on a simple plan, and her applied mathematics applied successfully.
> What was the girl's strategy?

This problem was reprinted in Gardner's *Mathematical Carnival*, Chapter 9: 'The Red-Faced Cube and Other Problems', Q.6 [**Gar75**]. Nahin also includes it as one of his 'Seven Classic Evasion Problems' in [**Nah07**].

(4) Generalize `mice.py` to create n turtles located at the corners of a regular n-gon. Investigate how the total distance (S) traveled by a bug varies as n varies, and compare it to the expected theoretical distances. *Hint*: the turtles will be spaced out around a circle of radius r_0 separated by an angle $\theta = \frac{2\pi}{n}$. The initial (x, y) coordinates of each turtle can be calculated using:

$$x = r_0 \cos(t\theta) \quad \text{and} \quad y = r_0 \sin(t\theta)$$

where t varies from 0 to $n - 1$, and the circle is centered at the origin.

6.15 A Rumor Spreads Through the Town

In a town with n inhabitants (one of whom is a gossip), a rumor spreads by word of mouth. Each person who has heard it keeps passing the rumor on until he meets someone who has already heard it. Then he stops, cured forever of this bad habit.

We divide the inhabitants into three groups X, Y, Z with x, y, z members respectively.

- X contains all those who have not yet heard the rumor ('susceptible').
- Y contains all those who are actively spreading the rumor ('infected').
- Z contains the people who are no longer gossiping ('recovered').

Initially we have $x_0 = n-1, y_0 = 1,$ and $z_0 = 0$, and at each time interval, x_i, y_i, and z_i are replaced by x_{i+1}, y_{i+1}, and z_{i+1}. It's always true that $x + y + z = n$.

Infections occur through XY-encounters consisting of xy pairings. The change in x_i is proportional to $x_i y_i$, so

$$x_{i+1} = x_i - \beta x_i y_i$$

β is often called the *infection rate*.

Recovery occurs through YZ- and YY-encounters. There are yz pairs of the first kind and $y(y-1)/2$ pairs of the second. Since YY-encounters increase Z by 2, the number of recoveries is proportional to $yz + 2y(y-1)/2$. We can remove the z from that equation by recalling that $x + y + z = n$, so the recovery change is $y(n - x)$.

This means that the size of Y (the 'infected' set) is increased by βxy (inflow from X) and reduced by $\beta y(n - x)$ (outflow to Z).

In summary, we will use three difference equations:

$$\begin{aligned} x_{i+1} &= x_i - \beta x_i y_i \\ y_{i+1} &= y_i + \beta x_i y_i - \beta y_i (n - x_i) \\ z_{i+1} &= z_i + \beta y_i (n - x_i) \end{aligned}$$

rumor.py obtains values for n and β (as b) from the user, and generates data for how X, Y, and Z change over a 'time' interval of 2000 steps. Fig. 6.33 shows the plot when $n = 100$ and $b = 0.0001$.

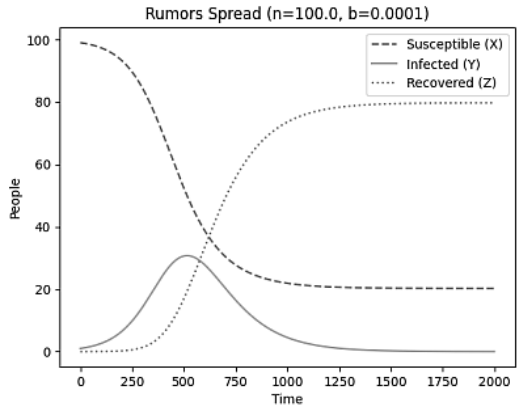


Figure 6.33. The rumor spreads

The graph indicates that from $x = n - 1$ to $x = \frac{n}{2}$, y increases; below $x = \frac{n}{2}$, y decreases. When $y < 1$ becomes true the rumor will die out, but this doesn't mean that everyone moves into the Z set (the 'recovered'). The X and Z curves level out, and rumor.py prints the final values: about 80 people recover and 20 people are left infected.

```
> python rumor.py
n p=? 100 1e-4
Final set sizes:
No. susceptible (x): 20.3
No. infected (y): 0.0
No. recovered (z): 79.7
```

The generation of the data in rumor.py (Listing 6.32) is done inside a loop that steps through the time period, generating new x, y, and z values based on their difference equations.

```
xs = [0]*NUM_ITERS  # no. susceptible
xs[0] = n-1
ys = [0]*NUM_ITERS  # no. infected
```

```
ys[0] = 1
zs = [0]*NUM_ITERS  # no recovered
t = 0
while t < NUM_ITERS-1:
  xs[t+1] = xs[t] - b*xs[t]*ys[t]
  ys[t+1] = ys[t] - b*ys[t]*(n-2*xs[t])
  zs[t+1] = zs[t] + b*ys[t]*(n-xs[t])
  t += 1
```

Listing 6.32. Calculating the rumor data

6.15.1 The SIR Model. Our rumor example uses the SIR model developed by Kermack and McKendrick in 1927 for the spread of infectious diseases (`https://en.wikipedia.org/wiki/Compartmental_models_in_epidemiology`). The 'S' stands for the "susceptible" population, those who may still acquire the disease; 'I' stands for the "infected", and 'R' stands for the "recovered" population.



Figure 6.34. The SIR model

We assume that, once recovered, an individual has built an immunity to the disease and cannot reacquire it. We also assume that the total population size is constant, so no one dies from the disease. Also, an individual always moves from a "susceptible" state to an "infected" state to a "recovered" state, where he remains, as depicted in Fig. 6.34.

These assumptions fit most common viral infections. A virus travels through a population more quickly when there are more infected people with whom a susceptible person can come into contact. Also, like most natural processes, the spread of disease is continuous, so we model these changes over small intervals of time.

Disease models of this type are often called *compartmental* models, where quantities are shuffled between fixed categories according to some rules.

The rules express changes in a small time interval Δt, so it's possible to let Δt go to zero to obtain differential equations. For example, the difference equations can be rewritten to include an explicit time interval Δt:

$$\begin{aligned} x_{i+1} &= x_i - \beta x_i y_i \Delta t \\ y_{i+1} &= y_i + \beta x_i y_i \Delta t - \beta y_i (n - x_i) \Delta t \\ z_{i+1} &= z_i + \beta y_i (n - x_i) \Delta t \end{aligned}$$

which is immediately removed as we rewrite the system as differential equations:

$$\begin{aligned} dx/dt &= -\beta xy \\ dy/dt &= \beta xy - \beta y(n - x) \\ dz/dt &= \beta y(n - x) \end{aligned}$$

A more general SIR model utilizes a separate *recovery rate*, called γ. The three differentials become:

$$\begin{aligned} dx/dt &= -\beta xy \\ dy/dt &= \beta xy - \gamma y \\ dz/dt &= \gamma y \end{aligned}$$

Many problems involving dynamic systems actually begin as differential equations with initial values, which are converted into difference equations so they can be implemented. The approach we've used implicitly in the rumor example was to model the differentials using the Forward Euler method.

6.15.2 The Forward Euler Method. Euler's method is based on the truncated Taylor series of y about x:

$$y(x+h) = y(x) + y'h$$

We assume that the derivative of y is some function of x and y:

$$y'(x) = f(x, y(x))$$

So

$$y(x+h) = y(x) + h \cdot f(x, y(x))$$

Because this equation predicts y at $x + h$ from the information available at x, it can be used to move the solution forward in steps of h, starting with given initial values for x and y.

The error in the equation is caused by truncation of the Taylor series after the second term:

$$E = \frac{1}{2}y''(e)h^2 = O(h^2), \quad x < e < x + h$$

A rough idea of the accumulated error E_{acc} can be obtained by assuming that the step error is constant over the period of integration. Then after n integration steps covering the interval x_0 to x_n we have

$$E_{\text{acc}} = nE = \frac{x_n - x_0}{h}E = O(h)$$

Hence, the accumulated error is linearly related to the step size h.

We'll continue using the forward Euler method, but should ensure that the step size in our code is *small*, to reduce the chance of errors distorting the result.

In general, forward Euler starts to become problematic when applied to rapidly oscillating systems, such as pendulums and springs. Typically, we then switch to the 4th-order Runge-Kutta method for stepping. As its name suggests, it exhibits an accumulated error of $O(h^4)$.

Exercises

(1) Run rumor.py for other values of n and b, and see how this affects the final outcome for the X, Y, and Z sets.

(2) a) There is a small boarding school of 50 staff and students where one of the pupils suddenly comes down with the flu.

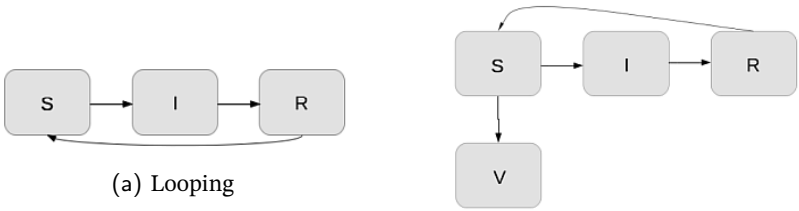
At another school where the flu has already spread, it was observed that at the start of a day there were 40 susceptibles and 8 infected, and the numbers were 30 and 18, respectively, 24 hours later. Using 1 hour as a time unit, $\beta = 10/(40 \cdot 8 \cdot 24)$. Why?

Also, on another day when there were 15 infected, it was noted that 3 recovered during the day, giving $\gamma = 3/(15 \cdot 24)$. Why?

Using these parameters for the boarding school, how will the pupils and staff be affected over the course of one month? Simulate the progress of the disease using a time step of 6 minutes (i.e. generate 10 data points for each hour of the simulation).

b) A "wash your hands" campaign successfully reduces β by a factor of 5. What now happens over the course of 60 days?

(3) **Time-restricted immunity**. Let's assume that immunity after recovering from the disease only lasts for a limited time. This means that there's a transition from the R state back to the S state as in Fig. 6.35a.



(a) Looping

(b) SIRV

Figure 6.35. Extending the SIR model

This is modeled by introducing an 'immunity loss' rate, ν. This creates an outflow from the recovered set, Z, into the susceptibles set X:

The three difference equations become:

$$\begin{aligned} x_{i+1} &= x_i - \beta x_i y_i + \nu z_i \\ y_{i+1} &= y_i + \beta x_i y_i - \gamma y_i \\ z_{i+1} &= z_i + \gamma y_i - \nu z_i \end{aligned}$$

Repeat Ex. 2 but assume ν is $1/(24*50)$, which represents 50 days of immunity. Why? Decrease β by a factor of 4, and increase the length of the simulation to 300 days to give the categories time to fluctuate.

(4) **Vaccinations**. The model in Ex. 3 can be extended again, this time to include the vaccination of susceptible individuals (see Fig. 6.35b).

As you might expect, we need to add a successful vaccination rate, ρ to the equations, and a set V. There will be an outflow of ρx from the X set (the susceptibles) into the V set.

The four difference equations:

$$\begin{aligned} x_{i+1} &= x_i - \beta x_i y_i + \nu z_i - \rho x_i \\ v_{i+1} &= v_i + \rho x_i \\ y_{i+1} &= y_i + \beta x_i y_i - \gamma y_i \\ z_{i+1} &= z_i + \gamma y_i - \nu z_i \end{aligned}$$

Modify your code from Ex. 3, and set ρ to 0.005 (very effective) or to 0.001 (poorly effective) to see how the disease progresses.

6.16 Eat and be Eaten: the Struggle for Existence

Two communities of rabbits and foxes live on a large, grassy island. The rabbits eat the grass, and the foxes eat the rabbits, making the rabbits the prey and the foxes the predators. We'll model how the two populations change over time using the Lotka–Volterra predator–prey model (LVM) (`https://en.wikipedia.org/wiki/Lotka%E2%80%93Volterra_equations`).

Let x be the population density of the prey (the number of rabbits per square kilometer) and y the population density of the predators (the foxes). The populations change over time according to a pair of first-order nonlinear differential equations:

$$\begin{aligned} \frac{dx}{dt} &= \alpha x - \beta xy, \\ \frac{dy}{dt} &= \delta xy - \gamma y \end{aligned}$$

α and β are the birth and death rates for the prey. More precisely, β is the effect the predators have on the number of prey. δ and γ are the birth and death rates for the predators. More precisely, δ is the effect the number of prey has on the number of predators.

The equations model (in a simple way) the connection between the two groups. If there were no foxes ($y = 0$), the rabbits would increase at a rate $\frac{dx}{dt} = \alpha x$. If there were only foxes ($x = 0$), they would die off at a rate $\frac{dy}{dt} = -\gamma y$.

If both foxes and rabbits are present, the number of encounters between them is proportional to xy. The number of rabbits eaten in any time interval is therefore proportional to xy. This explains the terms $-\beta xy$ and δxy.

The simplicity of this model is due to the large number of assumptions about the environment and biology which are unlikely to be true in a real situation. They include:

(1) The rate of change of the populations only depends on their size / density.

(2) The prey's food (grass) never runs out.

(3) The food supply for the predators comes entirely from the prey.

(4) The predators have limitless appetite.

(5) We ignore factors such as environmental change, evolution, spatial distribution, age distribution, disease, etc.

Nevertheless, the model does capture the interdependencies between predator and prey populations, which creates synchronized oscillations (see Fig. 6.36a below), similar to the fluctuations observed in natural populations.

Also, the LVM's general form means that it can be applied across a wide range of domains, such as economics and marketing. For instance, it can describe the dynamics in a market with several competitors, where one dominate company tries to drive the others out of business.

Like the earlier rumor model (see Listing 6.32), eatenPlot.py utilizes a loop that incrementally generates population numbers for the rabbits and foxes by stepping through two difference equations corresponding to the derivatives:

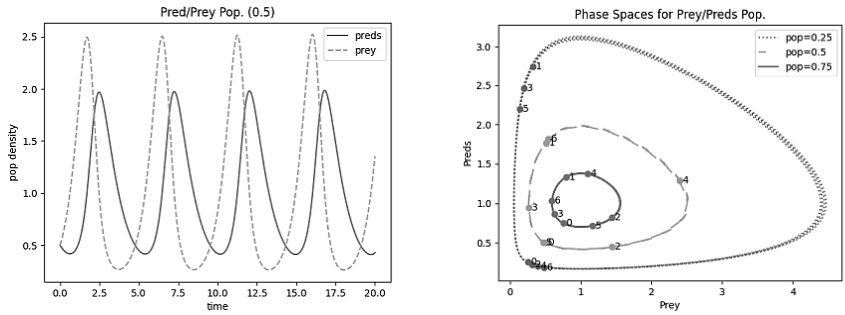
```
for i in range(N_t-1):
  prey[i+1] = prey[i] + dt*alpha*prey[i] -
                        dt*beta*prey[i]*preds[i]
  preds[i+1] = preds[i] + dt*delta*prey[i]*preds[i] -
                          dt*gamma*preds[i]
```

Fig. 6.36a shows the plot generated by eatenPlot.py when the user supplies 0.5 as the population densities for both the rabbits and foxes.

As the rabbit group grows, the foxes have more to eat, which leads to more foxes, but the increased number more rapidly consume the rabbit population. The foxes eventually have less to eat and so their numbers drop, relieving the pressure on the rabbits whose community begins to recover.

Another way to view the interaction between the populations is through a *phase space* chart, which plot the two against each other without including time.

eatenPhase.py generates the population data in the same way, but creates three phase plots. They all use the same model, but initialized with different starting densities (0.25, 0.5, and 0.75); the results are shown in Fig. 6.36b. Numbers (0 to 7) are written on each curve in ascending order of data generation to



(a) Populations over time (b) Phase plots

Figure 6.36. Predator and prey

highlight the counterclockwise direction of the 'orbits' and the number of loops. For example, the innermost curve indicates that perhaps four loops have been performed. The outermost curve is noticeably thicker along its top edge since different loops don't follow quite the same path – a result of rounding errors introduced by using the forward Euler method for stepping through the data.

The main reason for including three phase plots in Fig. 6.36b is to show that increasing the population densities reduces the sizes of the phase change. If we added more curves with larger densities, they'd eventually converge on $(1, 1)$, where population equilibrium occurs, when neither populations change, and both derivatives equal 0:

$$\begin{aligned} \alpha x - \beta xy &= 0 \\ \delta xy - \gamma y &= 0 \end{aligned}$$

There are two solutions:

$$x = 0, \quad y = 0$$

which signals societal death. The other indicates population balance:

$$x = \frac{\gamma}{\delta}, \quad y = \frac{\alpha}{\beta}$$

`eatenPhase.py` (and `eatenPlot.py`) sets these constants to $\gamma = 1, \delta = 1, \alpha = 2$, and $\beta = 2$, which explains why the fixed point occurs at $(1, 1)$.

It's possible to derive the relationship between x and y. The derivatives are divided to produce a differential equation independent of time:

$$\frac{dy}{dx} = -\frac{y}{x}\frac{\delta x - \gamma}{\beta y - \alpha}$$

This can be solved by separation of variables:

$$\frac{\beta y - \alpha}{y}\, dy = \frac{\delta x - \gamma}{x}\, dx$$

producing

$$V = \delta x - \gamma \ln(x) + \beta y - \alpha \ln(y),$$

where V is a constant depending on the initial conditions.

Exercises

(1) **Prey Limits**. Modify the LVM model to account for a restriction on growth arising from the depletion of the prey's food (grass) as its population grows. Change the prey's birth component from αx to $\alpha x(1 - x/K)$ to account for births decreasing as the population approaches a *carrying capacity* K. A good value to start with is $K = 20$, which will be visualized as a gradual decline in both populations as they fluctuate.

(2) Suppose that two predator species y and z compete for the same prey x, but don't eat each other. Nevertheless, if the supply of prey becomes low and one species is more efficient at hunting, then the other species will suffer. This type of model exhibits *indirect, exploitative competition*. It can be modeled as:

$$\begin{aligned} \frac{dx}{dt} &= \alpha x - x(\beta_1 y + \beta_2 z) \\ \frac{dy}{dt} &= \delta_1 xy - \gamma_1 y \\ \frac{dz}{dt} &= \delta_2 xz - \gamma_1 z \end{aligned}$$

The prey now has two death rates (β_1 and β_2) for the two types of predators. δ_1 and γ_1 are the birth and death rates for the y predators, while δ_2 and γ_2 are used for z.

(a) Use the following model parameters and initial conditions: $\alpha = 0.2$, $\beta_1 = 0.1$, $\beta_2 = 0.2$, $\gamma_1 = \gamma_2 = 0.1$, $\delta_1 = 1.0$, $\delta_2 = 2.0$, x(0) = y(0) = 1.7, and z(0) = 1.0.

(b) Can these two predators successfully share the same prey?

Additional Exercises for Chapters 1 to 6

(1) In a city of n people, someone thinks up a new joke. Anybody who hears this joke passes it on until they tell it to someone who has heard it before, which makes them stop. Find by simulation the percentage of the population who never hear the joke – a number practically *independent* of n. Assume that any person can meet any other person with the same probability.

(2) Starting with only 1 and 2, generate additional numbers by selecting any two existing numbers a, b, and create a new number $c = ab + a + b$, then keep repeating. What numbers do you get with this process?

(3) Let d_n be the number of fixed-point-free permutations of {1, 2, ..., n}.

a) Prove that $d_n = (n-1)(d_{n-1} + d_{n-2})$, and write an *inefficient* recursive program based on this recurrence relation.

b) From the formula in a), derive the recursion $d - n = nd_{n-1} + (-1)^n$, and write an *efficient* recursive program. (A permutation p is fixed-point-free if $p(i) \neq i$ for all i from 1..n.)

(4) Goldbach's conjecture states that all even numbers, except 2, can be expressed as the sum of two primes. When an even number can be expressed as the sum of two primes in more than one way, that number of ways is called its *Goldbach count*. For instance, 10 has a Goldbach count of 2 since 10 = 3+7 = 5+5, while $30 = 7 + 23 = 11 + 19 = 13 + 17$ has a Goldbach count of 3.

After studying Goldbach counts for a while you may notice that, except for very small numbers, a number $6n$ always has a bigger Goldbach count than its even left and right neighbors. However, show that this hypothesis is incorrect by finding a 'contact' number below 2000 and a 'crossing' value between 80000 and 80100. A 'contact' occurs when $6n-2$ or $6n+2$ has the same count as $6n$. A 'crossing' is when $6n-2$ or $6n+2$ has a larger Goldbach count than $6n$.

(5) This question first appeared in the *Lady's and Gentleman's Diary*, 1861 as Problem 1987: when three random points are used as the corners of a plane triangle, what is the probability that the triangle is acute? In 1862 Stephen Watson gave the answer 33/70, and you should check if this is correct via simulation. Write two versions, one where you choose the points at random from a 3D unit cube, and one where the points originate from a 3D unit sphere.

(6) S. W. Golomb defined the 'self-describing sequence' $f(1), f(2), \ldots$ as the only non-decreasing sequence of positive integers with the property that it contains exactly $f(k)$ occurrences of k for each k. For example $f(3) = 2$ means that the $f(n)$ sequence contain two occurrences of 3, and $f(7) = 4$ means that the sequence contain four occurrences of 7. Check by hand the other values given in Table 6.5.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
f(n)	1	2	2	3	3	4	4	4	5	5	5	6	6	6	6	7	7	7	7	8	8	8	8

Table 6.5. Golomb's $f(n)$ sequence.

a) Write a program that computes this sequence for $n = 1$ to 10000.

b) Plot $y = \ln f(n)$ versus $x = \ln n$. You will find that a straight line $y = a+bx$ is approached asymptotically.

c) From the line in b) derive the asymptotic formula $f(n) \sim cn^d$.

d) Find approximate values for c and d by substituting into the equation $n = 5000$ and $n = 10000$.

(7) *Knuth Numbers.* Knuth defined the sequence [**Knu97**]:

$$K_0 = 0; \quad K_{n+1} = 1 + \min(2K_{\lfloor n/2 \rfloor},\ 3K_{\lfloor n/3 \rfloor}).$$

a) Write a recursive program which computes the Knuth Numbers up to 10000. b) Check if $K_n > n$ in this interval.

(8) *Number of comparisons in mergesort* (Sec. 5.1.4). To sort n numbers ($n > 1$) we divide them into (almost) equal parts $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. After each part has been sorted (by the same method applied recursively), we can merge the numbers into one sorted set by doing at most $n-1$ comparisons. Let $f(n)$ be the total number of comparisons. Then

$$f(1) = 0,\ f(n) = f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + n - 1, \quad n > 1.$$

a) Write a recursive program, which for input n computes $f(n)$.

b) Guess a closed formula for $f(n)$.

(9) *Sequences which omit powers.* Check that the sequence

$$b_n = \lfloor n + ((n-0.5)^{1/m} + n - 0.5)^{1/m} \rfloor$$

omits m-th powers. Can you prove this?

(10) Two rectangles R1 and R2 with sides parallel to the x and y axes are specified by the coordinates of their lower left and upper right corners. Write a program that finds the area of R1 not contained in R2 for any two rectangles in the plane.

(11) Three people identified as 1, 2, 3 have a, b, and c chips, and play a game where each person stakes one chip in each round. A 3-sided symmetric die is rolled, and the i result means that person $\#i$ gets all three chips. The game end when any of the players runs out of chips.

Calculate $f(a,b,c)$, the expected duration of the game (i.e. the number of rolls), through simulation.

(12) Calculate the recursively intertwined functions $f(n) = n - g(f(n-1))$ and $g(n) = n - f(g(n-1)), n > 0, f(0) = 1, g(0) = 0$ in the interval $0..max$ by means of recursive and iterative code. Also, find closed expressions for $f(n)$ and $g(n)$.

(13) Select integers $a_1 < a_2 < ... < a_k$ to be the first k elements of a sequence. Now add more numbers to the sequence where each number n is *not* a sum of two (not necessarily distinct) elements already in the sequence. Find the first m elements of the sequence starting with $(1,2)$.

(14) *Pile Games.* (See "Additional Exercises for Chapters 1-4".) Start with three piles of a, b, and c chips. In each game step, an ordered pair of piles is chosen at random and a chip is moved from the first pile to the second. If a pile becomes empty, continue with two piles, but stop when only one pile is left.

a) Find by simulation the expected number $g(a,b,c)$ of steps until the game finishes.

b) Generalize to four piles with a, b, c, and d chips. Guess, and then check by simulation, the expected duration of the game.

c) Guess the duration for n piles and try to prove your guess using techniques discussed at the end of Appendix A. Also see Arthur Engel, 'The Computer Solves the Three Tower Problem', *American Mathematical Monthly*, Vol. 100, No. 1, Jan, 1993, pp. 62-64, (`https://sites.math.rutgers.edu/~zeilberg/akherim/GamblerRuin/Engel1993.pdf`).

(15) Generate 10000 triples (a, b, c) uniformly distributed in $(0, 1)$, and estimate the probability of them being the side lengths of a triangle. Solve the corresponding problem for quadrilaterals and pentagons. Try to guess a general law.

(16) Generate and count the records and anti-records in a random permutation $x_1x_2...x_n$ of $1..n$. An element x_k is a *record* if $x_i < x_k$ for $i < k$, where k is the position of the record. k is the position of an *anti-record* if $x_k < x_j$ for $j > k$. Find asymptotic formulas for these numbers in a random n-permutation.

(17) Write a program which randomly splits the sequence $1, 2, ..., 2n$ into two subsequences $a_1 < a_2 < ... < a_n$ and $b_1 > b_2 > ... > b_n$, and then computes $|a_1 - b_1| + |a_2 - b_2| + \cdots + |a_n - b_n|$. Comment on the result, and prove it.

(18) Find all representations of the numbers $a, aa, aaa, aaaa$ (a is a digit) as sums of squares $x_1^2 + x_2^2 + ...$ The x_i values must be in an arithmetic progression $m, m + d, m + 2d, ...$

(19) *Empirical Exploration.* Fill a list x of length $n = 2^k$ with random integers selected from {-1, 1}, and also add and set x[n] = x[0]. Then repeatedly apply the transformation

```
for i in range(n):
  y[i] = x[i]*x[i+1]
for i in range(n):
  x[i] = y[i]
x[n] = x[0]
```

a) Prove what eventually appears in x.

b) What happens if $n \neq 2$? Study cycle length $c(n)$ in this case.

c) When do you get a pure cycle?

d) Generalize to y[i] = x[i]*x[i+1]*x[i+2], and then to y[i] = x[i]*x[i+1] *...*x[i+p].

(20) There is a counter located at point $(1, 1)$ in a lattice of size $w \times h$. The following moves are allowed: double one coordinate (x or y), or subtract the smaller value from the larger and replace the larger by the result. Which lattice points can the counter reach?

(21) Construct a sequence of positive integers, starting with $a_1 = 1$ and $a_2 = 2$, so that any positive integer can be uniquely represented as the difference of two numbers in the sequence.

Hint: build the sequence using pairs of numbers. Suppose we already have k pairs, and want to construct the $(k + 1)$-th pair. Consider all the differences that can be made by the k pairs, and let d be the smallest difference not yet possible. Then set $a_{2k+1} = 2a_{2k} + 1$ and $a_{2k+2} = a_{2k+1} + d$.

(22) What does the following algorithm do?

```
a,b,eps = map(float, input("a,b,eps? "))
p = max(a,b); q = min(a,b)
while q >= eps:
  r = (q/p)*(q/p); s = r/(r+4)
  p = (2*s+1)*p; q = s*q
print(p)
```

eps is assigned a small number, such as 10^{-5}, so that smaller numbers can be ignored; a, b are floats.

First try different inputs to help you narrow in on the answer. For the proof, here's a strong hint: initially $Q = p^2 + q^2 = a^2 + b^2$, but what happens to it after one iteration of the loop? This cubically convergent algorithm is due to Cleve Mohler and Donald Morrison, published in *IBM J. Res. Dev.*, Vol. 27, No. 6, Nov. 1983, and online at `https://www.convexoptimization.com/TOOLS/Morrison.pdf`.

(23) Which numbers a) from 1 to 100; b) from 1 to 10000, have a *palindromic square*? Examples include 11 (121), 22 (484), and 26 (676). Also, find some palindromic squares whose squares are also palindromes.

(24) An n-digit natural number is an *Armstrong number* if it is equal to the sum of the n-th powers of its digits. All the 1-digit numbers are Armstrong numbers (e.g. $7 = 7^1$). The smallest example other than those is $153 = 1^3 + 5^3 + 3^3$. Find all the Armstrong numbers with 2, 3, 4, 5 digits. They are also known as *Narcissistic numbers*.

(25) Find all the natural numbers $< 10^6$ which are palindromes in both decimal and binary. For example, 33 qualifies since its binary version is 100001.

(26) a) Generate a random permutation $x_1 ... x_{2n}$ of the numbers $1..2n$ and check for the occurrence of $|x_i - x_{i+i}| = n$, for all $i \in 1..2n-1$. Repeat this experiment 10000 times and estimate the probability that this event occurs at least once. Guess the asymptotic value.

b) Let $p(n)$ be the probability that $|x_i - X_{i+i}| = n$ never occurs. Then we can show that $p(n) = p(n-1) + p(n-2)/(2n-1)/(2n-3)$. Check that $p(1) = 0, p(2) = 1/3$. Use this recurrence to find $p(n)$ for $n = 10000$, and check your guess. (XXX. IMO 1989.)

(27) All odd primes are of the form $4n+1$ or $4n+3$. Let $\pi_1(x)$ and $\pi_3(x)$ be the numbers of primes $< x$ of type $4n+1$ and $4n+3$, respectively. It seems that $\pi_1(x)$ is always $\pi_3(x)$, but that isn't true. Find the first x where $\pi_1(x) > \pi_3(x)$.

(28) How many of the 6-digit words from 000000 to 999999 have the property that the digital sums of their two halves are equal? (See Ex.16 after Sec. 1.7.) Write

this number as N_6^{10}, where 10 stands for the number base b and 6 for the length $2s$. One can show that

$$N_6^{10} = \frac{1}{\pi}\int_0^\pi (\sin(10x)/\sin x)^6\,dx, \quad \text{and} \quad N_{2s}^b = \frac{1}{\pi}\int_0^\pi (\sin(bx)/\sin x)^{2s}\,dx$$

a) Find N_6^{10} using `tramisim1.py` from Sec. 6.11.5.

b) We can approximate N_6^{10} with

$$N_6^{10} = \frac{1}{n}\sum_{i=1}^{n} f(x_i), \quad x_i = x_0 + i\pi/n, \tag{6.44}$$

$$f(x) = (\sin(10x)/\sin x)^6, \quad x_0 \neq k\pi/2, \quad k \in \mathbb{Z} \tag{6.45}$$

The restriction avoids 0/0. Show empirically that for $n \geq 28$ that formulae (6.44) and (6.45) are exact.

c) From the general formula

$$N_{2s}^b = \sum_{i=0}^{m}(-1)^i\binom{2s}{i}\binom{(s-i)b+s-1}{2s-1}, \quad m = \left\lfloor s(1-\frac{1}{b})\right\rfloor$$

find values of N_{2s}^b for different b and s, and empirically discover which n in terms of s and b makes this analogue of formulae (6.44) and (6.45) exact.

(29) Let $N(2n)$ be the number of $2n$-digit decimal words with the equisum property, and let $S(i,k)$ be the number of i-digit words whose digits sum to k. Show that

$$S(n+1,k) = \sum_{i=0}^{\min(k,9)} S(n,k-i), \quad \text{and} \quad N(2n) = \sum_{k=0}^{9n} S(n,k)^2.$$

Write a program which computes a table of the $S(n,k)$ numbers, and use it to find $N(2n)$.

(30) In the money changing problem of Sec. 2.8 we assumed d[1] = 1. When that condition is dropped, we get the more general Frobenius coin problem. Modify Listing 2.27 to use any set of coins to pay the amount n. For example, when $n = 30000$ and $k = 4$ with coins worth 131, 247, 353, and 661, then $a(n,k) = 638$.

(31) Modify the money changing code of the preceding problem to find G, the largest amount that is not representable. *Hint*: $i = G$ if i has 0 representations and d[1] numbers after i have more than 0 representations.

Experiment with 2, 3, and 4 coin types. For two coin types you should quickly determine a formula for G. But for three and up the problem seems hopeless.

If your program is correct you will get G = 4464 for the set of coins {131, 247, 353, 661}.

(32) We assign integers to each vertex of a pentagon which sum to $s > 0$. If any of the integers is negative, then pick one of them (e.g. x_i), add its value to both of its neighbors and replace x_i by $|x_i|$. Decide if this process always stops.

This question was considered the most difficult in the 1986 IMO. You're not expected to solve it, but you can empirically find the largest possible number $f(n)$ of steps until the process stops. Start with $(n, n, 1 - 4n, n, n)$, where n is a positive integer. You should find that $f(3) = 50$.

7

Fractals

Why Study Fractals?

- **To model natural shapes**. Fractals model irregular, self-similar structures such as coastlines, mountains, river networks, trees, plants, clouds, and turbulent flows, all of which cannot be captured easily by classical Euclidean geometry.

- **To describe complex systems**. Many chaotic systems naturally produce fractal patterns. Turbulence, weather systems, and population dynamics all show self-similar structure, and fractal dimension provides a more meaningful measure of complexity than normal geometric dimensions.

- **For their utility across many application areas.** Fractal tools help analyse signals that vary with scale, including time series, heartbeat variability, seismic data, and financial market fluctuations. Many networks exhibit fractal or self-similar organisation, including neural systems, social and communication networks. Many biological structures display fractal organisation, including the lungs, blood vessel networks, and neuron branching.

- **As an artform**. Fractals generate visually compelling patterns and digital art, benefiting from the appeal of self-similar and infinitely detailed forms. They underlie the generation of terrain, textures, and natural scenery in computer graphics and simulations.

7.1 Introduction

Benoit Mandelbrot coined the word 'fractal' to describe irregular geometric shapes containing structure at arbitrarily small scales. Many fractals appear similar at those different scales, an attribute called self-similarity, quite unlike a circle for instance, where a small portion of the perimeter sufficiently magnified is almost indistinguishable from a straight line.

We learn that doubling the sides of a polygon multiplies its area by four, and if the radius of a sphere is doubled, its volume scales by eight. However, when a fractal's length is doubled, it's scaled by a power that's not necessarily an integer. Indeed, some fractals, such as the Hilbert curve (see below), completely fill 2D space even though they are topologically only 1-dimensional.

Another way to observe this difference is by measuring length, area, and volume at different scales. For a conventional shape, say a circle of unit radius, the measurement of its perimeter at small scales will always produce an answer close to 2π. However, measuring the length of a fractal, such as a von Koch curve (see below), at smaller-and-smaller scales produces ever-increasing values.

Fractal self-similarity lends itself to a recursive definition for their construction (although iterative solutions are also possible). A common approach employs initiators and generators: an initiator is a starting shape, while the generator specifies how to create scaled copies, often by utilizing affine operations (e.g. rotations and translations) and deletion.

It should be said that almost all the well-known fractals, such as the Cantor set, the von Koch curve, the Sierpinski gasket, the Menger sponge, the Julia set, and Brownian motion, have been known for a long time. Mandelbrot's brilliance lay in grouping these ideas into a recognizable field, and showing how they could be used as an organizing principle for natural phenomena. Prior to fractal geometry, nature was usually regarded as a form of *noisy* Euclidean geometry. As Mandelbrot famously wrote in *The Fractal Geometry of Nature* [**Man83**]: 'Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightning travel in a straight line.'

7.2 The von Koch Curve

The von Koch curve, named after the Swedish mathematician Helge von Koch (1870–1924), begins as a straight line segment that's divided into three (see Fig. 7.1, (a)). The middle third is discarded in (b), replaced by two sides of an equilateral triangle pointing outwards. The end result is the generator shape in (c).

Using the labeling of Fig. 7.2, the K0 curve is replaced by K1. The next level (K2) is obtained by treating the four line segments of K1 as initiators that are

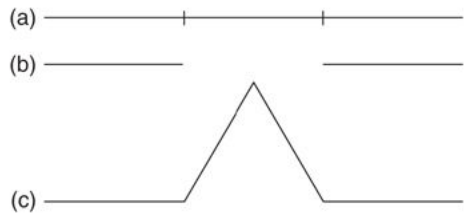


Figure 7.1. Initiator and generator for the von Koch Curve

redrawn as in Fig. 7.1 – each segment is divided into three, and the middle replaced by two outward facing lines, creating four copies of a scaled (and rotated) generator. The curve develops as this process is continued indefinitely.

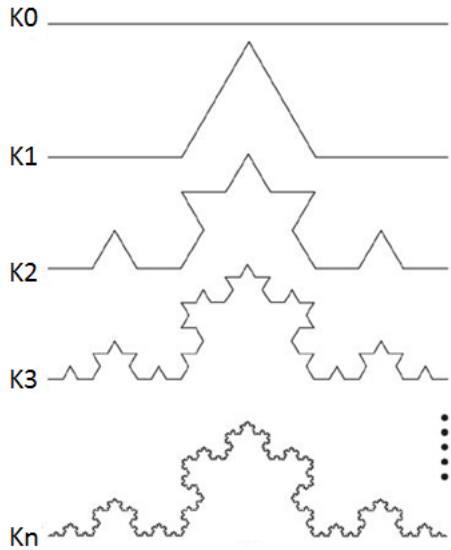


Figure 7.2. Levels of von Koch curves

The last curve in Fig. 7.2 shows the result of several iterations of the generator, and its self-similarity is quite clear.

A variant is the von Koch snowflake (Fig. 7.3), which applies the generator to the sides of an equilateral triangle.

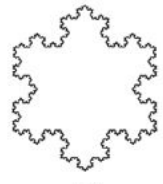


Figure 7.3. The von Koch snowflake

7.3 Fractal Dimensionality

The non-standard dimensionality of fractals becomes apparent if we consider how the length of the von Koch curve changes at each level. If the length of the initial line segment (K0 in Fig. 7.2) is l, the length L_n of the curve at the n-th level will be

$$L_n = \frac{4}{3}^n l$$

This quantity increases without bound, implying that the curve has an infinite length. However, the curve does have a bounded area even though equilateral triangles are added at each level. If one of these triangle has a side length s then its area is less than s^2 (the square containing the triangle). Hence, at step n, the area A_n "under" the curve is

$$\begin{aligned} A_n &< (\tfrac{1}{3})^2 + 3(\tfrac{1}{9})^2 + 9(\tfrac{1}{27})^2 + \cdots \\ &= \textstyle\sum_{i=1}^{n} \frac{1}{3^{i+1}} \end{aligned} \tag{7.1}$$

This is a geometric series of ratios less than one, and so converges.

7.4 Measuring Fractals

A simple way to measure fractals is by box-counting. The boxes can be any regular shape, but it's easiest to use squares, as in Fig. 7.4.

Smaller squares will obviously lead to a better approximation to the curve. Suppose we employ $N(r)$ squares of side length r, then $N(r) \cdot r$ approximates the curve's length, and $N(r) \cdot r^2$ its area. As a result, it's reasonable to assume that the relationship between $N(r)$ and $1/r$ is a power law, $N(r) = k(1/r)^d$, where d is termed the box-counting dimension and k is some arbitrary constant.

Taking logs of both sides:

$$\ln N(r) = d \ln \frac{1}{r} + \ln k$$

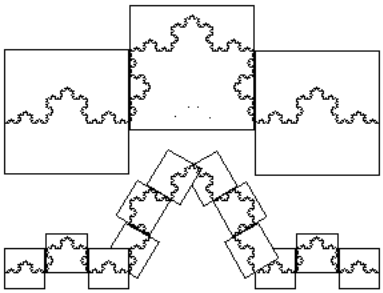


Figure 7.4. Boxes around the von Koch Curve

The expectation is that the approximation will become better for smaller r. Solving for d, and taking the limit as $r \to 0$ gives:

$$d = \lim_{r \to 0} \frac{\ln N(r) - \ln k}{\ln \frac{1}{r}}$$

Note that as $r \to 0$, $1/r \to \infty$, so $\ln 1/r \to \infty$ and $\ln k / \ln(1/r) \to 0$.

For a fractal where all of its components scale by the same factor, such as the von Koch curve and the Sierpinski gasket (see below), the equation can be simplified to

$$d = \frac{\ln N}{\ln \frac{1}{r}}$$

where N is the factor by which the number of generator copies increases at each level, and r is the scaling factor.

For the von Koch Curve, N increases by a factor of 4 at each level while the generator is scaled by 1/3 (i.e. $N = 4$ and $r = 1/3$), so we have:

$$d = \ln 4 / \ln 3 \approx 1.26186$$

The von Koch curve is more than 1-dimensional, but less than 2-dimensional. This suggests, at the very least, that the curve has a dimension not equal to the space it resides in.

Fractal dimension can be used as a measure of the complexity of a pattern, but is lacking in some respects. The number doesn't provide enough information to construct the curve, and obviously different fractals may have the same fractal dimension value.

7.5 The Sierpinski Gasket

Start with an equilateral triangle, connect the midpoints of its sides, and remove the resulting triangle, leaving three smaller, similar triangles (Fig. 7.5). The generator is recursively applied to these triangles, and after a few levels, the shape looks pleasingly complicated.

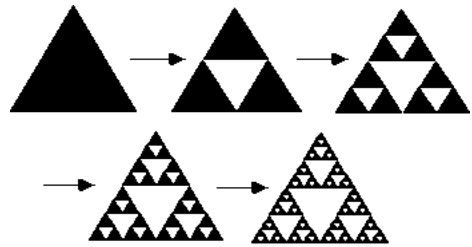


Figure 7.5. Levels of the Sierpinski gasket

The easiest way to box-count the gasket is to rotate it the left and change its initiator into a right-angled triangle. This lets the fractal be more readily covered with squares, as in Fig. 7.6.

At each level, the number of boxes, N, increases by a factor of 3 while the sides of each square are halved (i.e. $N = 3$ and $r = 1/2$), which means that:

$$d = \ln 3 / \ln 2 \approx 1.58996$$

The gasket is also more than 1-dimensional, but less than 2-dimensional, which tallies with the way that its area approaches 0 at reduced scales.

7.6 Other Ways to Define Fractals

Aside from the "initiator and generator" approach, there are several other ways to produce fractals:

- *L-systems* employ parallel string rewriting somewhat akin to the application of grammar rules. They're a popular choice for modeling plant growth, and were first used by the botanist Aristid Lindenmayer in 1968.
- *Iterated function systems* (IFSs) repeatedly apply geometric transformations to shapes, although non-linear functions, such as projective operations and circle inversion, are also possible. The popularity of IFSs is due in no small part to their use in the excellent text by Michael Barnsley, *Fractals Everywhere* [**Bar12**].

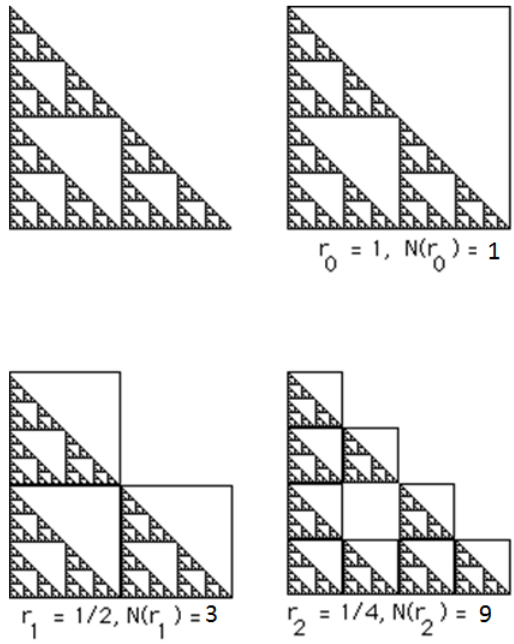


Figure 7.6. Boxing the Sierpinski Gasket

- *Escape-time fractals* apply a recurrence relation to each point in a space (usually the complex plane). After a fixed number of iterations, a test is carried out on the point to determine how it should be drawn. The Mandelbrot and Julia sets are two well-known examples.

The exercises at the end of this chapter investigate these approaches in a little more detail, with the help of code examples.

7.7 Coding Fractals

A key programming decision is how to draw the fractals. We mostly use Python's Turtle module in this section, but switch to matplotlib for plotting the Mandelbrot and Julia set coordinates (see Ex. 8) and 3D curves.

`fractals.py` lets the user choose between drawing a Sierpinski gasket, a dragon curve, a Levy curve, a von Koch snowflake, or a Minkowski island (Fig. 7.7). We'll give details on how the snowflake and gasket are generated.

7.7.1 The von Koch Snowflake. The snowflake is drawn by the following functions in `fractals.py`:

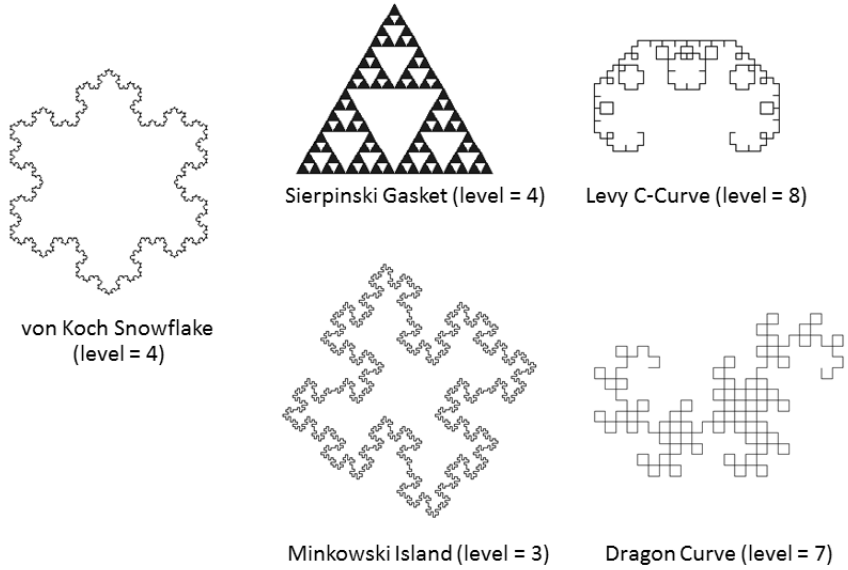


Figure 7.7. Various fractals

```
def koch(t, size, level):
  for _ in range(3):
    kochSide(t, size, level)
    t.right(120)

def kochSide(t, size, level):
  if level < 1:
    t.fd(size)
  else:
    for angle in [60, -120, 60, 0]:
      kochSide(t, size/3, level-1)
      t.left(angle)
```

Listing 7.1. Drawing the von Koch snowflake

koch() calls kochSide() three times to generate the three sides of the snowflake. When level = 1, kochSide() produces the K1 snowflake (see Fig. 7.2) using the operations illustrated in Fig. 7.8.

The curve is drawn using four forward moves and four angle changes (although the last is a turn of 0 degrees).

7.7.2 The Sierpinski Gasket. The gasket is drawn by the following function in fractals.py:

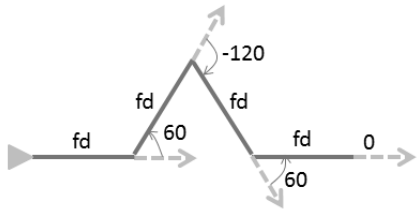


Figure 7.8. Turtle operations for Koch K1

```
def sierpinski(t, size, level):
  if level < 1:
    t.left(60)
    drawTriangle(t, size, "blue")
    t.right(60)
  else:
    sierpinski(t, size/2.0, level-1)
    t.left(60)
    t.fd(size/2.0)
    t.right(60)
    sierpinski(t, size/2.0, level-1)
    t.left(60)
    t.bk(size/2.0)
    t.right(60)
    t.fd(size/2.0)
    sierpinski(t, size/2.0, level-1)
    t.bk(size/2.0)
```

Listing 7.2. Drawing the Sierpinski gasket

drawTriangle() is imported from TurtleUtils.py to draw a triangle filled with a specified color.

The if-else branches in Listing 7.2 can be understood by considering Fig. 7.9.

The left-hand diagram in Fig. 7.9 illustrates the if-branch's execution when level = 0. The turtle is turned left before drawing, and returned to its starting position afterwards. The right-hand drawing deals with the more complicated `else` case when level = 1. The actions are split into three stages. Stage(a) represents the first line of the else-branch which calls sierpinski() to draw the left-most triangle. Stage (b) illustrates the next five lines:

```
t.left(60)
t.fd(size/2.0)
t.right(60)
sierpinski(t, size/2.0, level-1)
t.left(60)
```

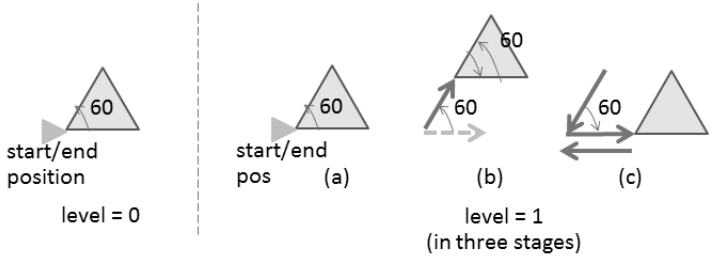


Figure 7.9. Turtle operations for two Sierpinski levels

The turtle turns left, advances to the starting point of the top triangle, and rotates right to have the correct orientation. sierpinski() draws the triangle, and the turtle is rotated left afterwards.

Stage (c) depicts the last five lines of the else-branch:

```
t.bk(size/2.0)
t.right(60)
t.fd(size/2.0)
sierpinski(t, size/2.0, level-1)
t.bk(size/2.0)
```

The turtle moves back to the bottom left corner, rotates to the right, and advances to the starting position of the right most triangle, which is drawn by the third call to sierpinski(). The turtle then returns to its starting position.

7.8 The 2D Hilbert Curve

Our most complicated initiator+generator is for the Hilbert curve, first proposed by David Hilbert in 1891. The first three levels of the curve are shown in Fig. 7.10.

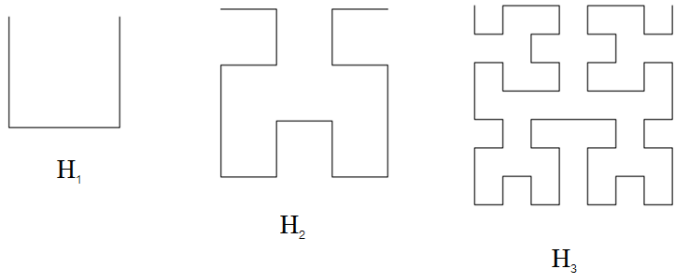


Figure 7.10. Hilbert curves (levels 1, 2, and 3)

The initiator is the ⊔ shape labeled as H_1 in Fig. 7.10. A level i curve, H_i, is obtained by the composition of four instances of H_{i-1} of half size and appropriate rotations, tying them together with three lines. Fig. 7.11 repeats the previous figure, but with those lines drawn more thickly so the sub-elements are easier to see.

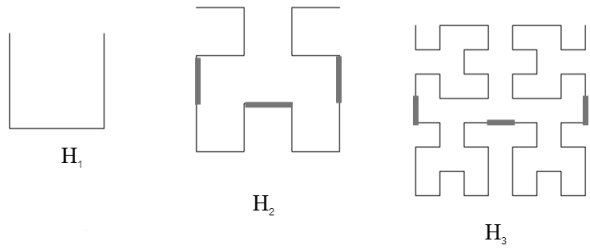


Figure 7.11. Hilbert curves (levels 1, 2, and 3) with thick connectors

The turtle code employs four recursive functions which draw the four rotations of the H_{i-1} shapes that make up H_i. We'll call these U, D, R, and L, short for "Upper", "Down", "Right", and "Left". The functions are recursive in the sense that the drawing done at level i utilizes the output of the functions at level $i-1$. The four basic drawings, denoted by U_1, D_1, R_1, and L_1, are shown in Fig. 7.12.

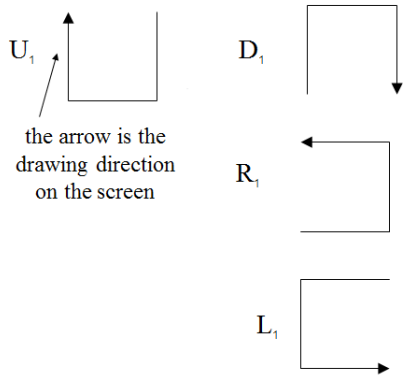


Figure 7.12. U_1, D_1, R_1, and L_1

The recursive U_i, D_i, R_i, and L_i operations follow these basic paths interspersed with calls to the other drawing functions for rendering the elements at level $i-1$ (see Fig. 7.13).

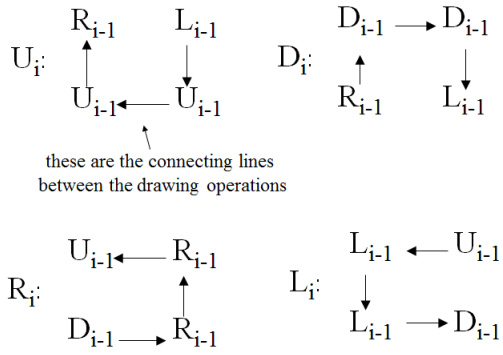


Figure 7.13. U_i, D_i, R_i, and L_i

The code for U_i (upperU()in Listing 7.3 from hilbert.py) corresponds to the U diagrams in Figs. 7.12 and 7.13. The same is true for the functions downU(), rightU(), and leftU().

```
def upperU(t, level):
  if level > 0:
    leftU(t, level-1)
    t.setheading(270)
    t.fd(step)
    upperU(t, level-1)
    t.setheading(0)
    t.fd(step)
    upperU(t, level-1)
    t.setheading(90)
    t.fd(step)
    rightU(t, level-1)
```

Listing 7.3. The U_i function

Note: setHeading() sets the turtle's orientation with 0 = east, 90 = north, 180 = west, and 270 = south

7.8.1 Dimensionality of the Hilbert Curve. The curve scales all of its components by the same factor, allowing its dimensionality to be expressed as

$$d = \frac{\ln N}{\ln \frac{1}{r}}$$

The N and r values can be obtained from the diagrams (Fig. 7.14) drawn by Hilbert on the first page of his 1891 paper (`https://terpconnect.umd.edu/~jmr/HonorsSem/Hilbert-orig.pdf`).

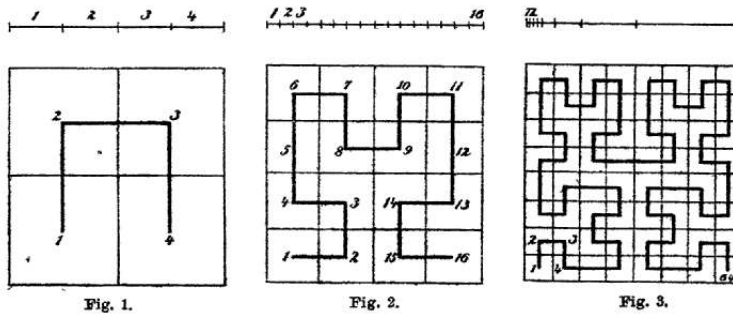


Figure 7.14. Hilbert's H1, H2, and H3

At each level, the number of squares increase by a factor of 4, and the sides of each square are halved. So $N = 4$ and $r = 1/2$, which means that

$$d = \ln 4 / \ln 2 = 2$$

The curve has the same dimensionality as the area of a standard geometric shape, which is reassuring since the curve does indeed cover its area as the number of levels increase. For example, consider H_6 in Fig. 7.15. This space-filling property has meant that the curve has found many uses related to mapping between 1D and 2D spaces.

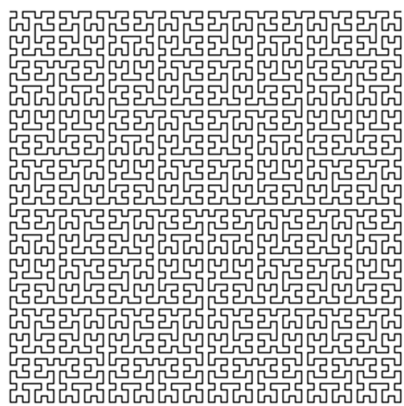


Figure 7.15. Hilbert curve (level = 6)

7.9 The 3D Hilbert Curve

It's possible to have the Hilbert curve cover 3D space. In hilbert3d.py the coordinates for the curve are generated recursively, then passed to matplotlib (Fig. 7.16). A pleasing aspect of its 3D plots is that it's possible to drag and rotate the curve to view it from various angles.

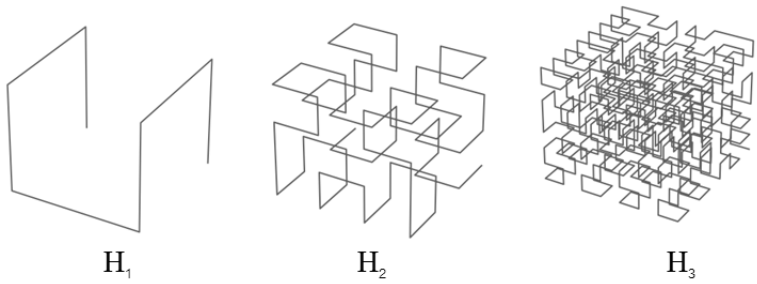


Figure 7.16. Hilbert 3D curves (levels = 1, 2, and 3)

The curve is rendered based on the ordering of the eight vertices used to define H1 (see Fig. 7.16) which are highlighted in Fig. 7.17.

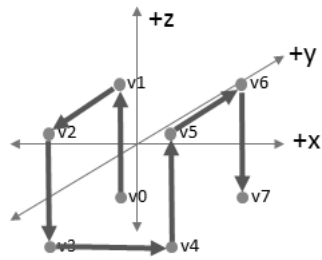


Figure 7.17. Hilbert 3D points (level = 1)

The eight points are stored in a list:

```
verts = [
    (center[0]-w/2, center[1]+w/2, center[2]-w/2),
    (center[0]-w/2, center[1]+w/2, center[2]+w/2),
    (center[0]-w/2, center[1]-w/2, center[2]+w/2),
    (center[0]-w/2, center[1]-w/2, center[2]-w/2),
    (center[0]+w/2, center[1]-w/2, center[2]-w/2),
    (center[0]+w/2, center[1]-w/2, center[2]+w/2),
    (center[0]+w/2, center[1]+w/2, center[2]+w/2),
```

```
    (center[0]+w/2, center[1]+w/2, center[2]-w/2)
]
```

Listing 7.4. Coordinates used for H1

It's challenging to see how H2 is derived from H1, so we'll focus on the four versions of H1 used to start building H2, which are circled in Fig. 7.18. The shape labeled 'call1' begins H2, followed by two copies of the same shape ('calls 2 and 3'), and then 'call4'. All of these shapes could be created by rotating and translating H1.

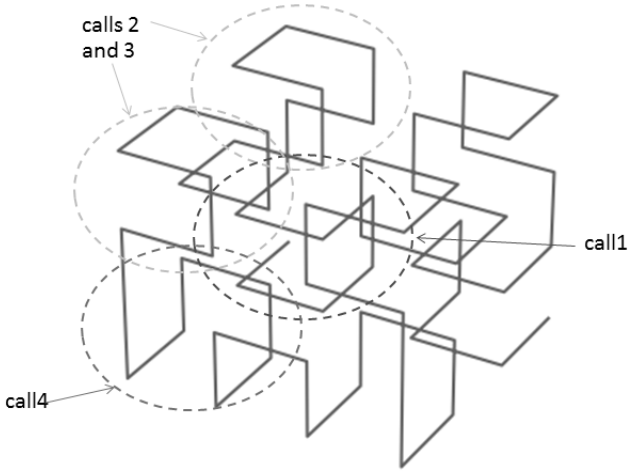


Figure 7.18. Drawing H2 Using H1s

Rather than use affine transformations, we've taken a simpler approach and defined the H1 versions by *reordering* the eight vertices in the original H1. Each reordering defines the drawing path for a variant, as shown in Fig. 7.19.

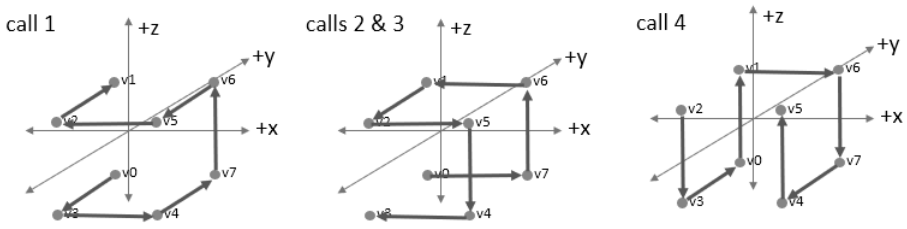


Figure 7.19. The H1 versions used to start drawing H2

The corresponding code in hilbert3d.py is quite succinct. The order of the vertices passed to each hilbert3D() call match their path order. This can be checked by comparing the path orders in Fig. 7.19 with the first four calls in the snippet in Listing 7.5 (hilbert3D.py).

```
coords = \
  hilbert3D(gray[0],w/2,d-1, v0,v3,v4,v7,v6,v5,v2,v1) + \
  hilbert3D(gray[1],w/2,d-1, v0,v7,v6,v1,v2,v5,v4,v3) + \
  hilbert3D(gray[2],w/2,d-1, v0,v7,v6,v1,v2,v5,v4,v3) + \
  hilbert3D(gray[3],w/2,d-1, v2,v3,v0,v1,v6,v7,v4,v5) + \
  hilbert3D(gray[4],w/2,d-1, v2,v3,v0,v1,v6,v7,v4,v5) + \
  hilbert3D(gray[5],w/2,d-1, v4,v3,v2,v5,v6,v1,v0,v7) + \
  hilbert3D(gray[6],w/2,d-1, v4,v3,v2,v5,v6,v1,v0,v7) + \
  hilbert3D(gray[7],w/2,d-1, v6,v5,v2,v1,v0,v3,v4,v7)
return coords
```

Listing 7.5. The coordinates for H2

Listing 7.5 indicates how the coordinates for H_i are constructed by eight calls to H_{i-1} with the w/2 argument reducing the length of each path edge by 1/2. This means that the curve's dimensionality uses $N = 8$ and $r = 1/2$:

$$d = \ln 8 / \ln 2 = 3$$

In other words, the curve has the same dimensionality as the *volume* of a standard geometric shape, which corresponds to how the curve fills 3D space.

Exercises

(1) The initiator for a Cantor set is a line segment, and its generator removes the middle third.



Figure 7.20. The first five levels of the Cantor Set

Evaluate the set to several levels, and determine its fractal dimensionality (*hint*: it's approximately 0.63093). This value may seem rather surprising since the set appears to be on it way to completely disappearing. In fact, despite the deletions there will be as many points left behind as there were at the start, implying that the set is uncountable.

Code for generating the set is in cantorset.py. Modify it to remove a different size segment or segments. How do these changes affect the set's dimension?

Note that if the generator produces parts with different sizes then we can no longer use the measure:

$$d = \frac{\ln N}{\ln \frac{1}{r}}$$

A good explanation of how to deal with more complex dimensionalities can be found in Section 2.D at the Yale "Fractal Geometry" site (`https://users.math.yale.edu/public_html/People/frame/Fractals/`), and the associated textbook [**FU16**].

Multi-dimensional Cantor sets are often called Cantor dust, such as the one shown in Fig. 7.21 which was drawn by `cantorCheese.py`; compare its code with `cantorset.py`.

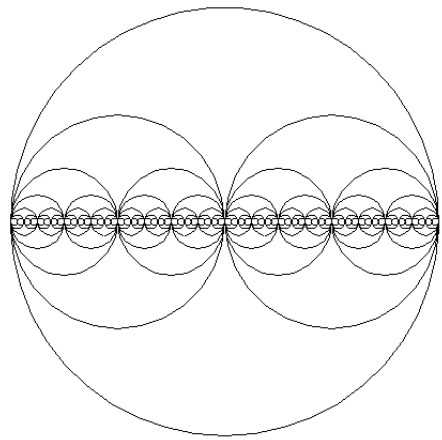


Figure 7.21. Cantor cheese

(2) Mandelbrot's "How Long Is the Coast of Britain?" [**Man67**] considers the coastline measurement problem: the fact that the measured length of a stretch of coastline depends on the size of the measuring instrument. For example, if the coastline of Great Britain is measured in units of 100 km, then the length comes to approximately 2,800 km. But with 50 km units, the total is nearer 3,400 km.

This is worth trying for yourself: obtain a map of some favorite coastline (e.g from `https://free-map.org/`) and measure its length at a variety of scales. Plot your data as a log-log graph, and ask yourself what the slope of the resulting line represents.



Figure 7.22. Measuring a coastline

A good description of this problem can be found in Section 2.H at the Yale "Fractal Geometry" site (`https://users.math.yale.edu/public_html/People/frame/Fractals/`) in the "Coastlines" subsection.

(3) The Sierpinski carpet starts with a square which the generator divides into 8 smaller squares with the middle one removed. Evaluate the set to several levels, and determine its fractal dimensionality (*hint*: it's approximately 0.63093). Code for drawing the carpet in Fig. 7.23 can be found in sierCarpet.py.

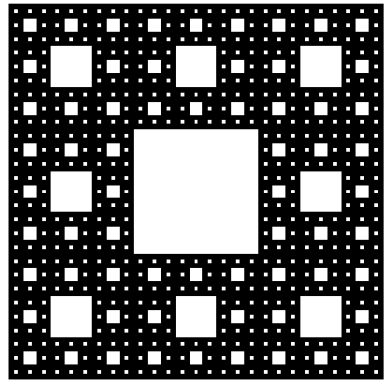


Figure 7.23. A Sierpinski Carpet

(4) The **L-system** approach to coding fractals can be tried out by running lsys.py, which can load rules for a variety of fractals stored as text files in `http://coe.psu.ac.th/~ad/explore/code/Fractals/lsys/`. Examine the

code, try out a few examples, and note the code's features and limitations. For instance, consider the rules for generating a tree in Listing 7.6 (tree.txt).

```
# axiom
F

# rule
FF+[+F-F-F]-[-F+F+F]

# angle
22.5
```

Listing 7.6. L-system tree definition

This definitions are loaded and executed like so:

```
> python lsys.py
fnm=? tree
axiom: F
rule: FF+[+F-F-F]-[-F+F+F]
angle: 22.5

level? 3
```

The user entered the filename and `level` value, and Fig. 7.24 was drawn.

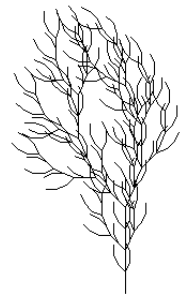


Figure 7.24. An L-system drawing of a tree

Hints: our L-system implementation offers stack pushing and popping which is necessary for state storage and retrieval (as utilized in bracketed L-systems), but only supports a single rule. This makes it impossible to implement most of the fractals listed on the L-systems wikipedia page (`https://en.wikipedia.org/wiki/L-system`). It would be a fun project to remedy that deficit by supporting multiple rules in lsys.py.

(5) The **IFS** approach to coding fractals can be tested by running our ifs.py, which can load parameters for drawing fractals stored in `http://coe.psu.ac.th/~ad/explore/code/Fractals/ifs/`. Examine the code, and note its features and limitations. For instance, consider the data for generating a Barnsley fern in Listing 7.7 (barnsley.txt).

```
4
   0.01   0.85   0.07   0.07

4 3
   0.00   0.00   0.500
   0.85   0.04   0.075
   0.20  -0.26   0.400
  -0.15   0.28   0.575

4 3
   0.00   0.16   0.000
  -0.04   0.85   0.180
   0.23   0.22   0.045
   0.26   0.24  -0.086
```

Listing 7.7. IFS definition for a Barnsley fern

This data is loaded and executed like so:

```
> python ifs.py
fnm=? barnsley
probs: [0.01, 0.85, 0.07, 0.07]

cx: [0.0, 0.0, 0.5] [0.85, 0.04, 0.075]
    [0.2, -0.26, 0.4] [-0.15, 0.28, 0.575]

cy: [0.0, 0.16, 0.0] [-0.04, 0.85, 0.18]
    [0.23, 0.22, 0.045] [0.26, 0.24, -0.086]
```

The user entered the filename and Fig. 7.25 was drawn.

The underlying mechanism used by ifs.py is a matrix-based affine transformation of an (x,y) point:

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

The a, b, c, d, e, and f parameters are read from the text file passed to ifs.py, and utilized in:

```
x0 = cx[r][0]*x + cx[r][1]*y + cx[r][2]
y0 = cy[r][0]*x + cy[r][1]*y + cy[r][2]
```



Figure 7.25. An IFS drawing of a Barnsley fern

Chapter 10 of *Fractals Everywhere* [**Bar12**] uses IFSs to generate deterministic fractals. We've added a simple notion of randomness, which you should try to locate in the code.

Also, have a look at the IFS for the Sierpinski gasket (`sierpinski.txt`) and understand its relationship to the Sierpinski initiator+generator in Listing 7.2.

(6) It can be argued that none of our initiator+generator code really matches our description of how that mechanism works. For example, the most natural way of describing the generator for the Sierpinski gasket is as the *removal* of the middle triangle of the initiator. However, our code implements this by turtle operations involving moves and turns.

The Turtle module doesn't support set operations on images (e.g. union, intersection, and (crucially) subtraction). Pillow is one library which does have these features (`https://python-pillow.org/`), and we've included a small example of how to subtract one image from another in `subtract.py`. It loads an existing image (`http://coe.psu.ac.th/~ad/explore/code/Fractals/triangle.png`) and subtracts the inner triangle, as depicted in Fig. 7.26.

Install the Pillow library, and try out the code; it could become the basis of a removal-based generator for the Sierpinski gasket.

(7) Our code examples include `mandel.py` and `julia.py` which generate the Mandelbrot and Julia sets respectively. The Mandelbrot set contains the complex numbers c for which the function $f_c(z) = z^2 + c$ does *not* diverge to infinity when iterated from $z = 0$. The corresponding pixel in the image (Fig. 7.27) is painted black if the sequence doesn't run away to infinity, otherwise it's color is based on how quickly it went to infinity.

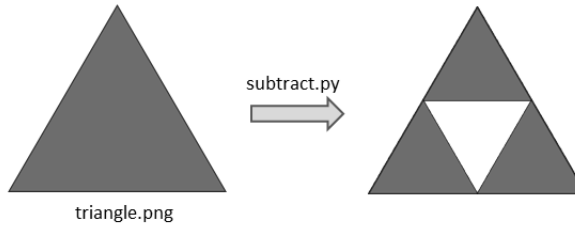


Figure 7.26. Using the Pillow module to subtract a triangle

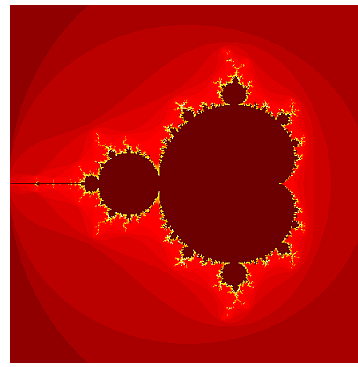


Figure 7.27. The Mandelbrot Set

One of the surprises is that the edges of the Mandelbrot set are filled with tiny copies of the entire set, each of which is surrounded by its own halo of still tinier copies, and so on, without end. All of these copies are attached to the main body, so every part of the image is connected.

Have a look at `mandel.py` to see how this is implemented. Sadly, there's no interaction, such as panning or zooming.

The Julia set contains the complex numbers z for which the $f_c(z) = z^2 + c$ doesn't diverge to infinity. Fig. 7.28 shows the image generated with c = (-0.3, 0.71j).

`julia.py` is coded in a similar way to `mandel.py`, but compare them to see how they differ. Try varying the c starting values; a few interesting ones are included as comments at the start of the program

The link between the two sets is that the Mandelbrot contains those c values for which the Julia set forms a single connected image. More visually, a particular Julia set corresponds to a particular c coordinate in the Mandelbrot image. Fig. 7.29 shows a series of different Julia sets around its edge, to

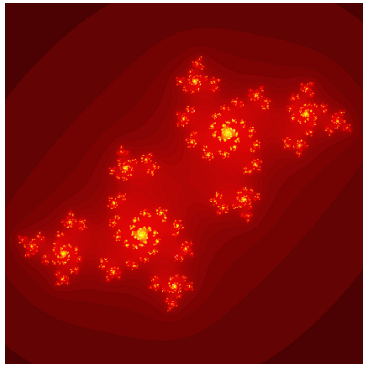


Figure 7.28. The Julia Set for c = (-0.3, 0.71j)

which we've added lines that link to their corresponding c coordinate in the Mandelbrot.

(8) sierpinskiCA.py generates the Sierpinski Gasket using a **Cellular Automaton** (CA). The output begins as shown in Fig. 7.30.

A CA is defined by four attributes:

(a) The state space represented as a collection of cells.
(b) The neighborhood of a cell – those cells whose current states affect the next state of the cell.
(c) A collection of the different states that a cell can assume.
(d) A rule (or rules) which specify how the states of the cells in the neighborhood determine the next state of a cell.

sierpinskiCA.py implements a 1-dimensional CA where a cell only has a left and right neighbor, and can be in one of two states ("*" meaning True; " " for False). The rule states that the status of the current cell is set to True if one of its two neighbors at the previous step was True, otherwise it's set to False. Rather amazingly, this rule causes the automaton to generate a Sierpinski gasket. You should check that you understand why this occurs.

Despite their simplicity, CAs have proved useful for modeling a wide range of physical, chemical, and biological problems.

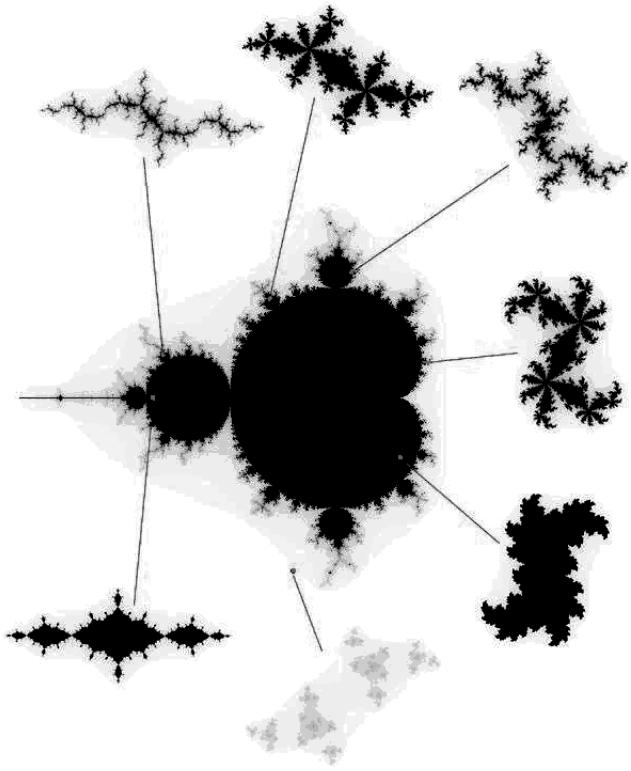


Figure 7.29. Linking the Mandelbrot and Julia Sets

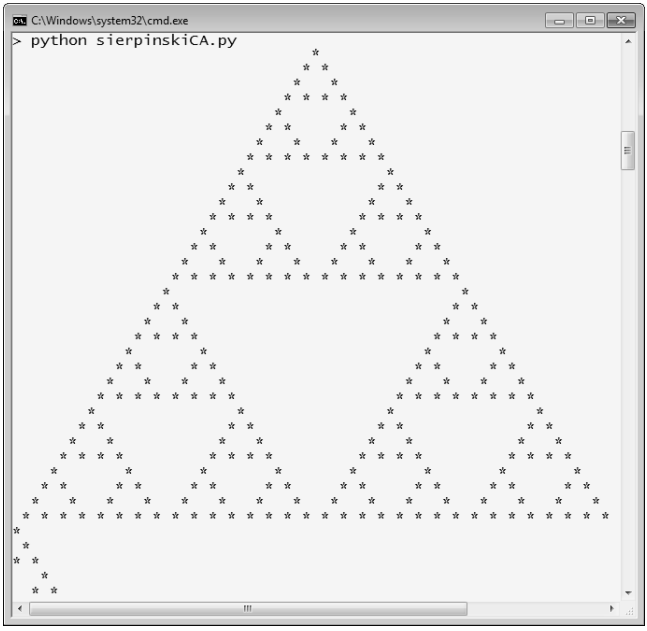


Figure 7.30. The Sierpinski Gasket using a cellular automaton

8

Chaos

Why Study Chaos?

- **It reveals the structure hidden behind apparent randomness**. Chaos theory shows that systems that look unpredictable can still follow precise deterministic laws. This helps distinguish true randomness from complex but regulated behaviour.

- **It highlights how systems are sensitivite to initial conditions**. Small uncertainties can grow exponentially in chaotic systems. Understanding this gives realistic limits on prediction in weather, climate, economics, and many physical processes.

- **It provides tools for modelling real-world complexity**. Many natural and engineered systems, such as turbulent fluids, population dynamics, electrical circuits, and chemical reactions, are chaotic. Their analysis utilizes ideas from chaos theory, including attractors, Lyapunov exponents, and bifurcations.

- **It improves forecasting and control**. Recognising when a system is near a bifurcation into chaos informa how we design stabilising systems, feedback controls, and simulations.

- **It unifies behaviour across disciplines**. Chaos reveals patterns, such as period-doubling and fractal structures, that recur in fields as different as biology, physics, engineering, and finance.

8.1 Introduction

Chaos occurs when a deterministic system behaves unpredictably, in a non-repeating and apparently random fashion. This is not due to some hidden external noise or randomness in the system, but a consequence of simple, deterministic actions.

One source of confusion is that this mathematical meaning doesn't really match chaos' common usage as a synonym for an absence of order. Another issue is that although the term originated in the mid-1970s, chaos is a part of the much larger nonlinear dynamic systems field, which dates back at least to Henri Poincaré in the 1890s.

A dynamic system is chaotic if it possesses the following properties:

(1) The system depends on a deterministic function (or functions) which, given the same input, always returns the same result.

(2) The results are aperiodic, with the function never entering a cycle.

(3) The results are bounded by upper and lower limits, so the system's aperiodicity is not just a consequence of being able to generate infinite sequences.

(4) The system exhibits *sensitive dependence on initial conditions* (SDIC), in that a very small change in its initial state will lead to very large changes in its outputs in a short time.

We'll look at two famous examples: the logistic map and the Lorenz model. For more details, Strogatz offers a great introduction [**Str15**].

8.2 The Logistic Map

The logistic map, a discrete-time version of the logistic equation for population growth, is one of the simplest nonlinear equations that exhibits chaos:

$$x_{n+1} = rx_n(1 - x_n)$$

x_n represents the population in the n-th generation, which can range between $0 \leq x \leq 1$. r is the growth rate, which varies in the interval $[0, 4]$.

The generation of x_n from some initial population x_0 can be represented using a *cobweb* diagram, such as the one on the left of Fig. 8.1. The conversion of the map's output to become its next input (e.g. $x_0 \to x_1 \to x_2 \cdots$) is modeled by 'bouncing' the map's output off the line $y = x$.

Fig. 8.1 shows that in the $r = 2.5$ case the population converges to a steady state (a fixed point) where $x = rx(1 - x)$.

Another useful view of the map's execution is a time series plot of x_n against n, shown on the right of Fig. 8.1. x_n clearly converges upon a single value (0.6), which can be confirmed by solving $x = 2.5x(1 - x)$.

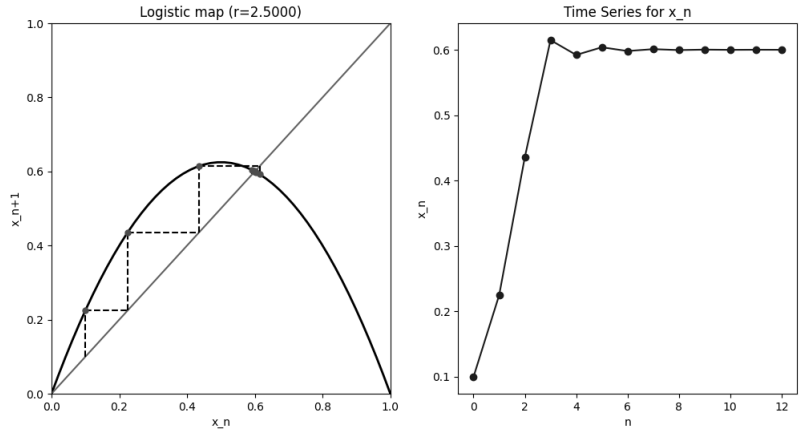


Figure 8.1. Cobweb and time series for $r = 2.5$

It's instructive to see what happens to the map by running `logCobweb.py` with different r values, but a more interactive approach offered by `animLogCobweb.py` uses a matplotlib slider (Fig. 8.2).

For small growth rates of $r < 1$, the population always goes extinct (see Fig. 8.2), while for $1 < r < 3$ the population eventually reaches a nonzero steady state (as in Fig. 8.1).

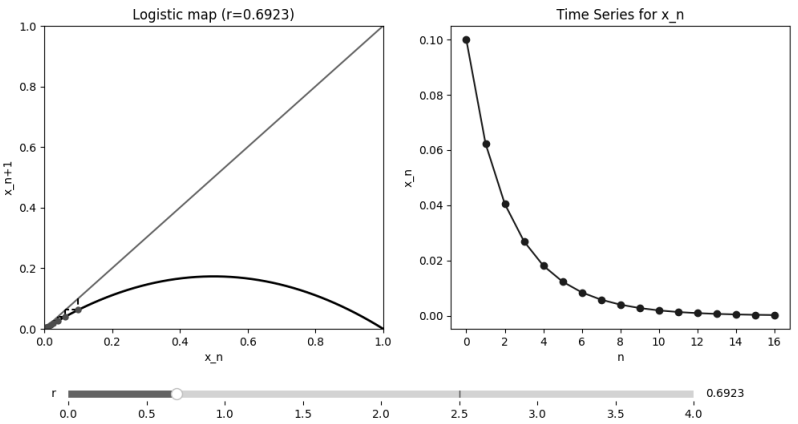


Figure 8.2. Cobweb and time series for $r = 0.6923$

Things become more interesting when $r \geq 3$. For $r = 3.3$, the population builds up but, instead of converging, gradually begins to oscillate between two population values (Fig. 8.3). This is called a period-2 cycle.

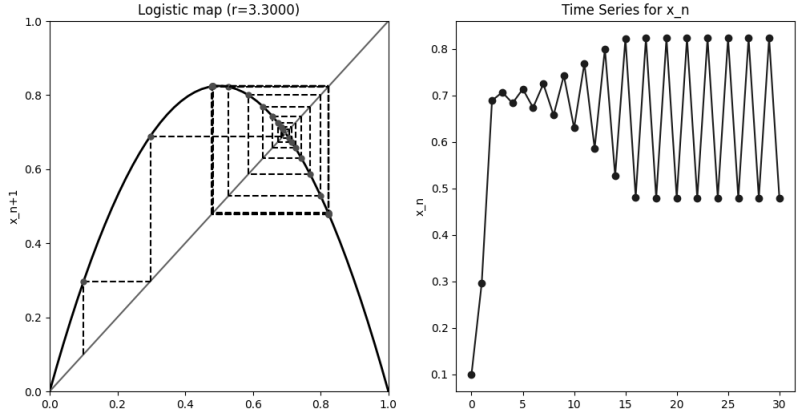


Figure 8.3. Cobweb and time series for $r = 3.3$

For a larger value, such as $r = 3.5$, the population starts repeating every four generations – a period-4 cycle (Fig. 8.4).

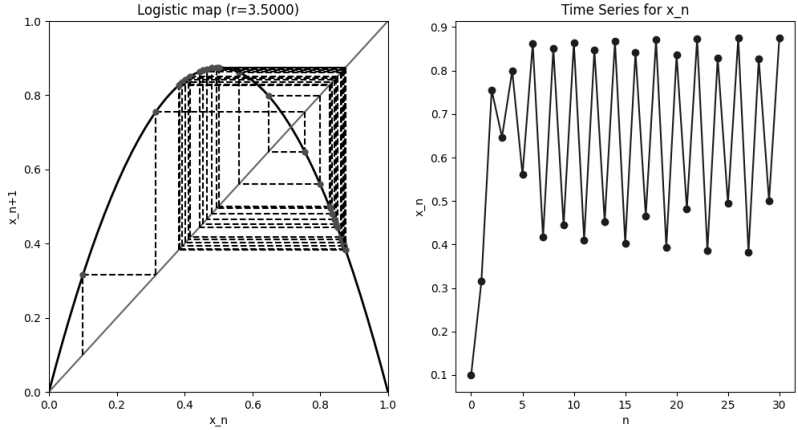


Figure 8.4. Cobweb and time series for $r = 3.5$

Doublings to cycles of period 8, 16, 32,... occur as r increases. Experimentation with `animLogCobweb.py` produces the results summarized in Table 8.1.

r	Period-i
$r_1 = 3$	2
$r_2 = 3.44949... \ (1 + \sqrt{6})$	4
$r_3 = 3.54409...$	8
$r_4 = 3.5644...$	16
$r_5 = 3.568759...$	32

Table 8.1. Period doubling for r values

Splittings (*bifurcations*) of the population's cycles occur more and more quickly, with chaos seemingly breaking out at $r \approx 3.56995$ (e.g. see $r = 3.9$ in Fig. 8.5). However, the actual behavior is more subtle, requiring a different visualization.

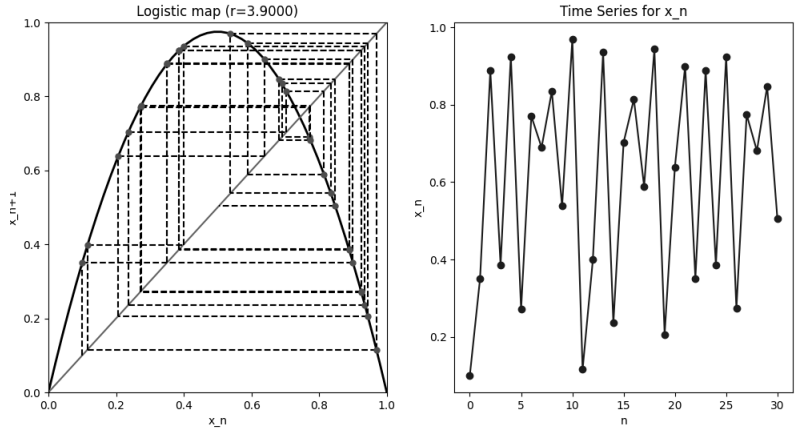


Figure 8.5. Cobweb and time series for $r = 3.9$

We switch to an orbit diagram (also called a *bifurcation diagram*) which plots values of r against the x_n's produced. Fig. 8.6 shows that the map splits into a period-2 cycle at $r = 3$ by dividing into two branches. As r increases, these branches multiply, until the map becomes chaotic.

Fig. 8.6 was produced using Listing 8.1 (bifurcation.py).

```
NUM_RS = 10000   # more r detail when zoomed-in
NUM_ITERS = 500

def logistic(r, x):
        return r * x * (1 - x)

def logistics(r):
  xs = [0.1]
```

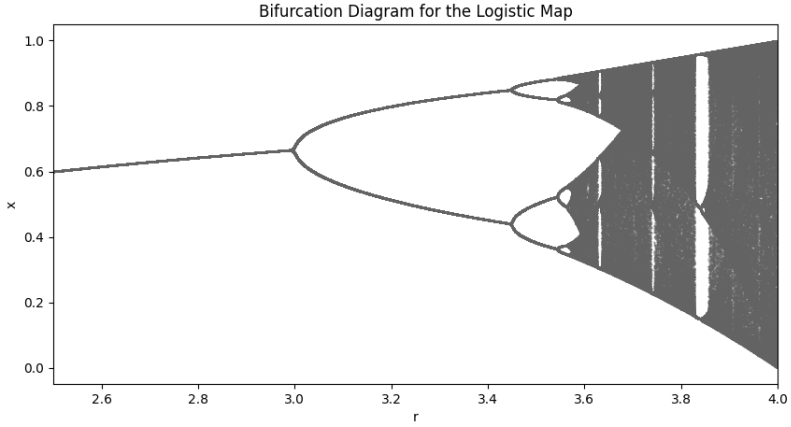


Figure 8.6. Bifurcation diagram for the logistic map

```
  for i in range(NUM_ITERS-1):
    xs.append(logistic(r,xs[i]))
  return xs[NUM_ITERS-100:]     #  last 100 iterations

rs = linspace(2.0, 4.0, NUM_RS)  # r between 2 and 4
xVals = []
rVals = []
for r in rs:
  xs = logistics(r)
  xVals.append(xs)
  rVals.append([r] * 100)
plt.figure(figsize=(10, 5))
plt.scatter(rVals, xVals, s=0.1)
plt.xlabel('r')
plt.ylabel('x')
plt.xlim(2.5, 4)
plt.title("Bifurcation Diagram for the Logistic Map")
plt.show()
```

Listing 8.1. Implementing bifurcation

The logistics() function generates a series of x values for a specific r (also called an *orbit*), but discards the first 100 since it takes some time for the map to settle down to a steady behavior. Each orbit is plotted as a points on the $x = r$ line, with r ranging between 2 and 4 in steps of 0.0001. This level of detail is necessary since matplotlib allows the graph to be panned and zoomed in upon, as in Fig. 8.7.

The enlarged bifurcation diagram in Fig. 8.7 reveals an intriguing mixture of order and chaos, with periodic empty spaces (called *windows*) interspersed

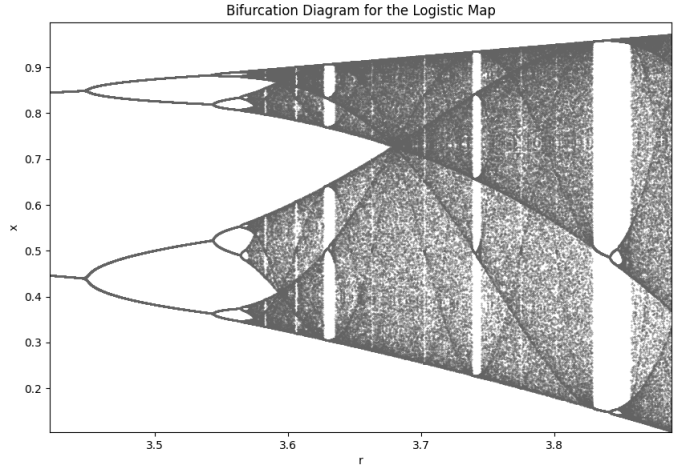


Figure 8.7. Bifurcation diagram between 3.4 and 3.9

between chaotic clouds of dots. It's also possible to discern elements of self-similarity, as in the lower-left corner of Fig. 8.7, enlarged in Fig. 8.8, which is nearly identical to the entire diagram.

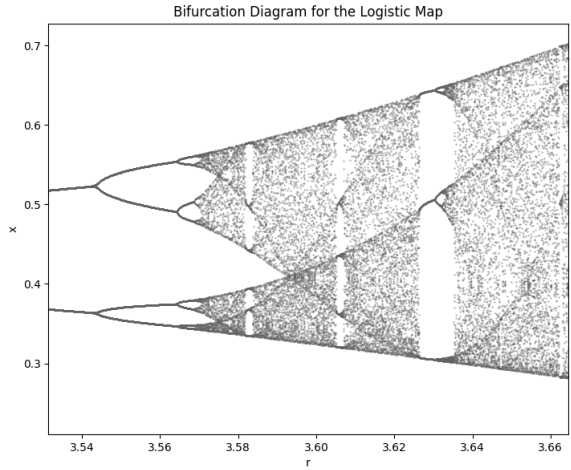


Figure 8.8. Bifurcation diagram between 3.54 and 3.66

8.3 Stability and Period Doubling

Fixed points (x_f) for the logistic map occur when $x_f = rx_f(1 - x_f)$. Hence, $x_f = 0$ and $1 - \frac{1}{r}$.

A fixed point is *stable* when mapping a x_n that's close to x_f produces a new point x_{n+1} that's even nearer. An unstable fixed point is one where the new point is further away than the original. This can be judged by looking at the map's gradient at the fixed point: x_f is stable if $|f'(x_f)| < 1$. The logistic map derivative is $f'(x) = r(1-2x)$. So for $x_f = 0$, $f'(x_f) = r$, and for $x_f = 1-1/r$, $f'(x_f) = 2-r$. This means that $x_f = 0$ is stable when $r < 1$, and $x_f = 1 - 1/r$ is stable when $1 < r < 3$.

Another concept is fixed point *superstability*. This is a fixed-point x_f where $|f'(x_f)| = 0$, causing nearby points to converge upon it at the fastest rate.

One issue is how to choose a r value to represent a given period in the logistic map. The approach we used in Table 8.1 was to list the first r that exhibited that behavior. In Fig. 8.9, these are labeled r_0, r_1, and r_2. However, it's more useful to choose an r where its sequence of x values contain a superstable fixed point. For the logistic map, where $f'(x) = r(1 - 2x)$, $|f'(x)| = 0$ occurs when x = 0.5.

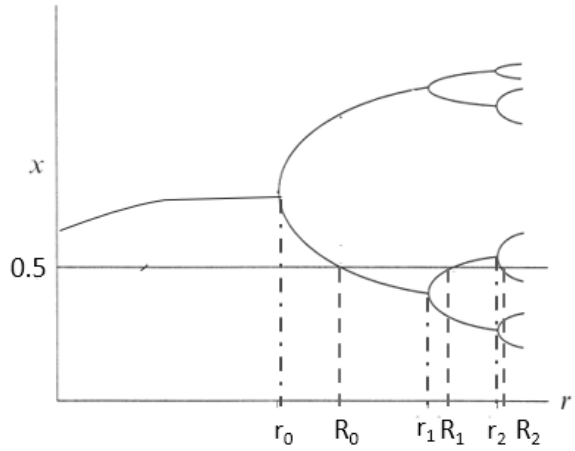


Figure 8.9. Choosing r values for period doubling

We'll denote these superstable sequences of period length 2^k using R_k, and R_0, R_1, and R_2 are shown on Fig. 8.9. From now on, when we need to compare r values of period 1, 2, 4,... We'll use these R values.

Fig. 8.10 shows the logistic map when $r = R_1 = 3.2361$. The sequence has a period length 2, which includes 0.5 as one of its fixed points.

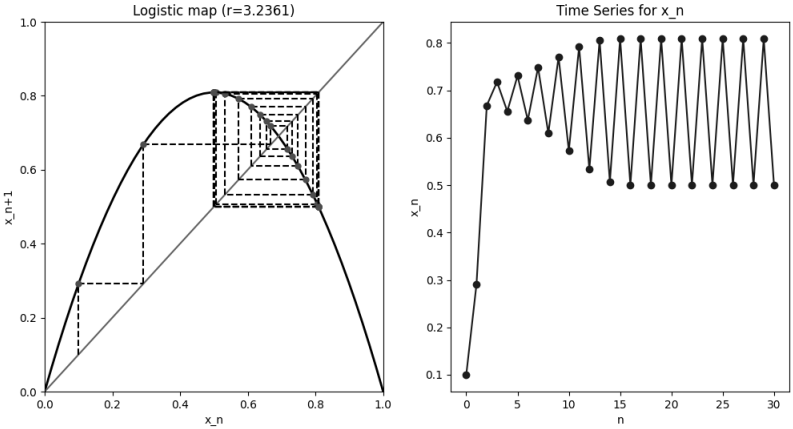


Figure 8.10. A superstable fixed point (R_1) at $r = 3.2361$

R_1 can be calculated exactly by solving $f(f(0.5)) = 0.5$ with $r = R_1$ and $x = 0.5$. Note that we are using two applications of $f()$ since the fixed point is in a sequence of period length 2.

$$\begin{aligned} f(f(0.5)) &= 0.5 \\ f(R_1 0.5(1-0.5)) &= \\ f(R_1/4) &= \\ R_1(R_1/4)(1-R_1/4) &= \\ R_1^2/4 - R_1^3/16 &= 0.5 \\ R_1^3 - 4R_1^2 + 8 &= 0 \\ (R_1-2)(R_1^2 - 2R_1 - 4)) &= 0 \end{aligned}$$

So $R_1 = 2$ or $1 \pm \sqrt{5}$, of which only $R_1 = 1 + \sqrt{5} \approx 3.2361$ is applicable here. Note that R_1 is related to the golden ratio ϕ, with $R_1 = 2\phi$.

$f(f(x))$ – often abbreviated as $f^{(2)}(x)$ – has two fixed points at $x_0 = 0.5$ and $x_1 = 0.809...$ (see Fig. 8.10). If r is increased, then these two fixed points will become unstable at the same time, and split into a period length of 4. This can be confirmed by applying the chain rule:

$$\frac{d}{dx} f(f(x))\Big|_{x=x_0} = f'(f(x))\Big|_{x=x_0} \cdot f'(x)\Big|_{x=x_0}$$

or, with $f(x_0) = x_1$,

$$\frac{d}{dx} f(f(x)) = f'(x_1) \cdot f'(x_0)$$

Note that the same result is obtained at $x = x_1$, by the symmetry of the final term. So if x_0 becomes unstable because $|f^{(2)'}(x_0)| > 1$, so does x_1 at the same r.

Now $f^{(2)}(f^{(2)}(x)) = f^{(4)}(x) = f(f(f(f(x))))$ will have four fixed points. For $r = R_2 = 3.49856...$, they are $x_0 = 0.5$, $x_1 = 0.874...$, $x_2 = 0.383...$, and $x_4 = 0.0.827...$.

We checked the answer by running WolframAlpha (`https://www.wolframalpha.com`), as shown in Fig. 8.11.

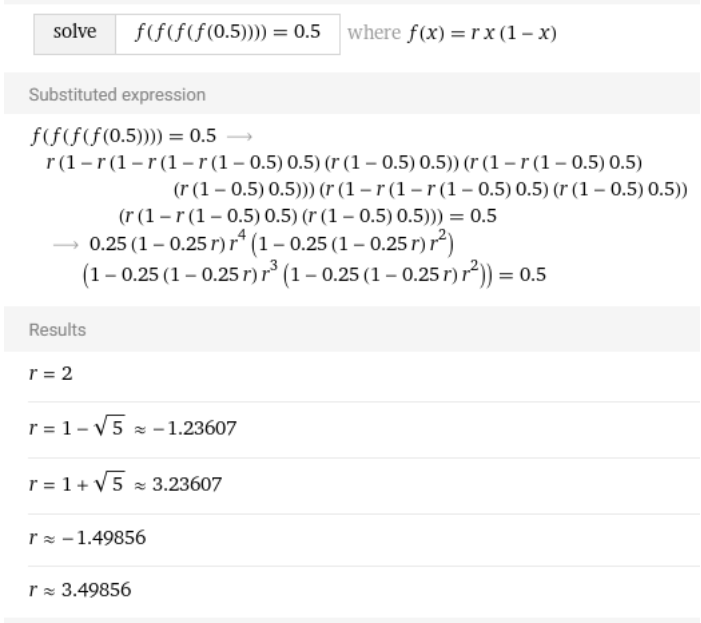


Figure 8.11. Calculating R_2 in the logistic map

8.4 The Lyapunov Exponent

One useful indicator of chaotic behavior is the Lyapunov exponent, which measures how an infinitesimally small distance between two close states grows exponentially quickly over time.

Given some initial condition x_0, consider a nearby point $x_0 + \delta_0$. Let δ_n be the separation after n iterations of the map such that $|\delta_n| = |\delta_0|e^{n\lambda}$, where λ is the Lyapunov exponent. A more computationally useful formula can be derived by utilizing

$$\delta_n = f^{(n)}(x_0 + \delta_0) - f^{(n)}(x_0)$$

Taking logs of the exponent:

$$\begin{aligned}\lambda &\approx \frac{1}{n}\ln\left|\frac{\delta_n}{\delta_0}\right| \\ &= \frac{1}{n}\ln\left|\frac{f^{(n)}(x_0+\delta_0)-f^{(n)}(x_0)}{\delta_0}\right| \\ &= \frac{1}{n}\ln\left|f^{(n)'}(x_0)\right|, \quad \text{when } \delta_0 \to 0\end{aligned}$$

The term inside the logarithm can be expanded by generalizing the chain rule technique used earlier so that:

$$f^{(n)'}(x_0) = \prod_{i=0}^{n-1} f'(x_i)$$

So:

$$\begin{aligned}\lambda &\approx \frac{1}{n}\ln\left|\prod_{i=0}^{n-1} f'(x_i)\right| \\ &= \frac{1}{n}\sum_{i=0}^{n-1}\left|f'(x_i)\right|\end{aligned}$$

λ will be negative for stable fixed points and cycles, and positive for chaotic behavior.

logLyapunov.py extends the code in bifurcation.py to generate a bifurcation diagram *and* a Lyapunov exponent curve, as in Fig. 8.12.

λ is negative for $r < 3.57$, but approaches zero at the period-doubling bifurcations. Chaos begins near $r \approx 3.57$, when λ first becomes positive, and increases for $r > 3.57$, except for the dips caused by the periodic windows.

The main change to logLyapunov.py compared to bifurcation.py is the calculation of the exponent inside the logistics() function in Listing 8.2 (logLyapunov.py).

```
def logistics(r):
  xs = [0.3]
  lyapunov = 0
  for i in range(NUM_ITERS-1):
    xs.append( logistic(r,xs[i]))
    lyapunov += math.log(abs(r - 2*r*xs[-1]))
        # sum of log( | f'(x) | )
  return xs[NUM_ITERS-100:], lyapunov/NUM_ITERS
     #  last 100 iterations
```

Listing 8.2. logistics() in logLyapunov.py

8.5 The Feigenbaum Constants

The Feigenbaum constants, δ and α, are ratios present in every bifurcation diagram of a nonlinear map with a single quadratic maximum (such as the logistic map).

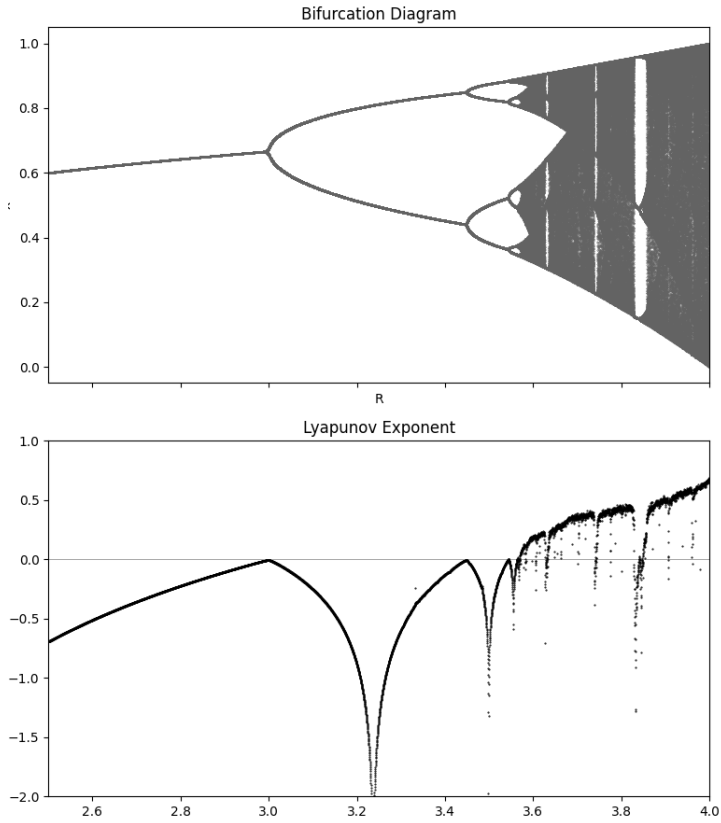


Figure 8.12. Bifurcation diagram and lynapunov exponent for the logistic map

Mitchell Feigenbaum began to study period-doubling in the mid 1970s, focusing on r_n, the value of r when a 2^n-cycle first appears. He noticed that r_n converges geometrically, with the distance between successive transitions, δ, defined as the limit:

$$\delta = \lim_{n\to\infty} \frac{r_n - r_{n-1}}{r_{n+1} - r_n} = 4.6692016091029...$$

The first few transitions are shown in Fig. 8.13.

Calculating r_i to a high level of accuracy is difficult, but a much simpler alternative is to determine the superstable values R_i. These can be used in the same way to calculate δ:

$$\delta = \lim_{n\to\infty} \frac{R_n - R_{n-1}}{R_{n+1} - R_n}$$

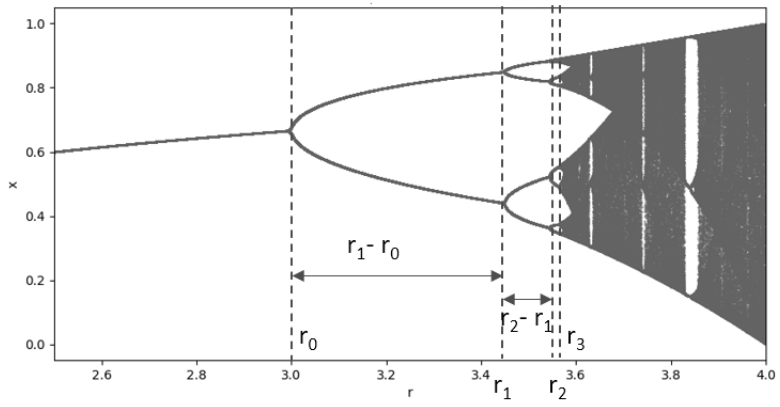


Figure 8.13. δ convergence

Feigenbaum's other constant, α, is the limit of the ratios of the distance a_i and its successor at a_{i+1}:

$$\alpha = \lim_{n\to\infty} \frac{a_n}{a_{n+1}} = 2.502907876...$$

a_i is the distance between the two branches of bifurcation diagram when R_i is detected.

Fig. 8.14 shows the placement of the first few R_i and a_i values.

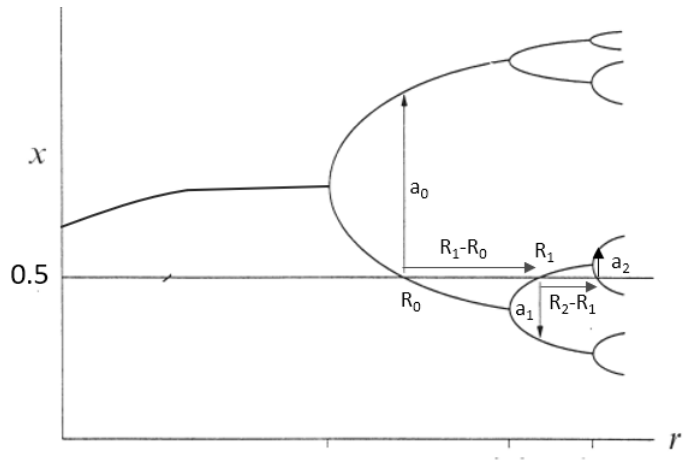


Figure 8.14. Calculating Feigenbaum's constants

Keith Briggs developed an algorithm for calculating the Feigenbaum's constants [**Bri89**], which is implemented in feigenbaum.py. His approach uses a simple quadratic equation ($x_{n+1} = a - x_n^2$). Its derivation is $f'(x) = 2x$ which means that its R_i values all occur at $x = 0$, which makes it easy to find them using Newton's root finding method. The corresponding map is:

$$\begin{aligned} b_k(a) &= a - [b_{k-1}(a)]^2, \quad k = 1, 2, 3, ... \\ b_0(a) &= 0 \end{aligned}$$

Its R_i values will occur at the zeros of b_{2^i}. These are found by starting with an approximation a_i, then using Newton's method to zero in on the correct value by looking for a root. This approach is defined using six equations:

$$\begin{aligned} a_i^0 &= a_{i-1} + \frac{a_{i-1} - a_{i-2}}{\delta_{i-1}}, \quad i = 2, 3, 4, ... \\ a_i^{j+1} &= a_i^j - \frac{b_{2^i}(a_i^j)}{b'_{2^i}(a_i^j)}, \quad j = 0, 1, 2, ... \\ b'_k(a) &= 1 - 2b'_{k-1}(a)b_{k-1}(a), \quad k = 1, 2, 3, ... \\ a_i &= \lim_{j\to\infty} a_i^j \\ \delta_i &= \frac{a_{i-1} - a_{i-2}}{a_i - a_{i-1}} \\ \delta &= \lim_{i\to\infty} \delta_i \end{aligned}$$

The first equation produces an initial approximation to the next superstable a value, which is refined using Newton's method using the derivative db/da of b. a_i^j will converge on the i-th root of b_{2^i}.

Listing 8.3 (feigenbaum.py) is an almost direct translation of these equations using Python's decimal to ensure extended precision.

```
NUM_ITERS = 20
A_ITERS = 10

def calcAIter(n, a):
  # perform Briggs equations 7, 8, 9 (first 3)
  for _ in range(A_ITERS):
    b = D(0)
    bp = D(0)
    for _ in range(2**n):
      bp = 1 - 2*bp*b   # represents bPrime()
      b = a - b**2      # represents b(); equ 5
    a = a - (b / bp)
  return a, bp


d = D('3.2')    # will become the feigenbaum constant
a2 = D(0)  # the ordering is [ a2, a1, a, ...]
```

```
a1 = D(1)
bpOld = D(1)
# perform Briggs equations 10, 11, 12 (last three)
print(" i      delta          alpha")
for i in range(2, NUM_ITERS+1):
  a, bp = calcAIter(i, a1 + ((a1 - a2) / d))
  d = (a1 - a2)/(a - a1)
  alpha = d*bpOld/bp
  bpOld = bp
  print(f"{i:2d}  {d:.10f}   {alpha:.10f}")
  # prepare for next iteration; move elems backwards
  a2 = a1
  a1 = a
```

Listing 8.3. Implementing Feigenbaum

The output:

```
> python feigenbaum.py
 i      delta          alpha
 2  3.2185114220   1.8560344182
 3  4.3856775986   -2.3948809906
 4  4.6009492765   -2.4753300767
 5  4.6551304954   -2.4974031887
          :   # more lines, not shown
18  4.6692016091   -2.5029078751
19  4.6692016091   -2.5029078751
20  4.6692016091   -2.5029078751
```

The rate of convergence is roughly linear, with one significant decimal digit of improvement every two iterations.

The calculation of α is based on the fact that:

$$\lim_{i\to\infty} \frac{b'_{i+1}(a_{i+1})}{b'_i(a_i)} = \delta/\alpha$$

This is derived from the connection between the b^i derivative and the lengths $R_i - R_{i-1}$ and a_i in Fig. 8.15.

We know

$$b_i \approx \frac{a_i}{R_i - R_{i-1}} \quad \text{and} \quad b_{i+1} \approx \frac{a_{i+1}}{R_{i+1} - R_i}$$

So

$$\begin{aligned} \frac{b_{i+1}}{b_i} &\approx \frac{a_{i+1}}{R_{i+1} - R_i} \cdot \frac{R_i - R_{i-1}}{a_i} \\ &= \frac{R_i - R_{i-1}}{R_{i+1} - R_i} \cdot \frac{a_{i+1}}{a_i} \\ &= \frac{\delta_i}{\alpha_i} \end{aligned}$$

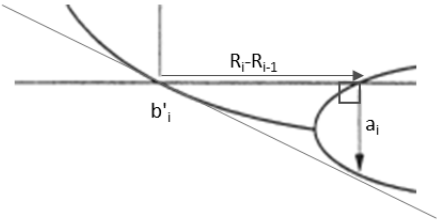


Figure 8.15. Relating b_i to the Feigenbaum constants

8.6 Self-similarity Again

We've already seen that the bifurcation diagram contains self-similarity; another example utilizes Feigenbaum's α constant = -2.5029....

R_0 ($r = 2$) is the superstable value for r in $f(x) = rx(1-x)$, while R_1 ($r = 3.236$) is the superstable value for r in $f(f(x))$. These two equations are plotted in Fig. 8.16.

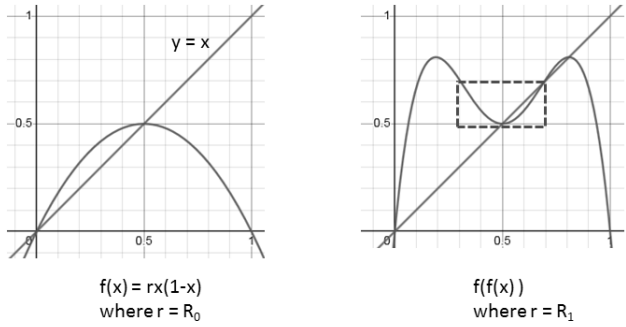


Figure 8.16. Self-similarity for the logistic map, $f(x)$

The dotted box in the plot of $f(f(x))$ when $r = R_1$ outlines an area of the curve which is similar to the $f(x)$ plot except that it's scaled by about half and reflected. This transformation, called *renormalization*, can be obtained by employing Feigenbaum's α as a scaling factor ($x \to x/\alpha$). The negative sign inverts the curve.

This scaling can be seen in each curve defined by R_i. For example, for R_2 ($r = 3.4986$), the curve is $f^{(4)}(x) = f(f(f(f(x))))$ which is plotted using Desmos (`https://www.desmos.com/calculator`) in Fig. 8.17. The scaled version

of $f(x)$ can be observed around the fixed point (i.e. when the curve touches the $y = x$ line).

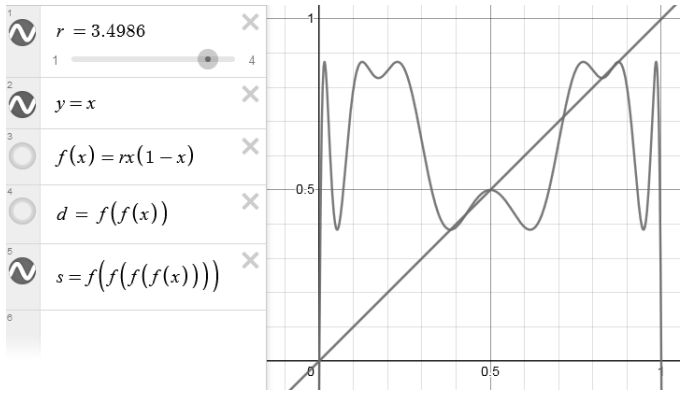


Figure 8.17. Self-similarity for R_2 in $f^{(4)}(x)$

We've included the Desmos parameters in Fig. 8.17 so you can reproduce these graphs for yourself.

8.7 The Lorenz System

The Lorenz system is a simplified model of the motion of the atmosphere as it's heated from below and cooled from above. The air rises and falls along the opposite edges of long cylinders that rotate because of the temperature difference (as depicted in Fig. 8.18).

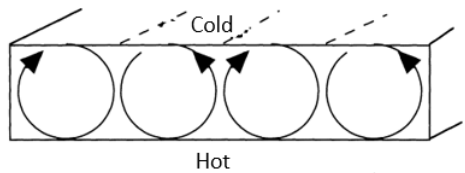


Figure 8.18. Convection in the Lorenz model

The temperature difference reduces as the air turns, causing the rotation to slow down and eventually stop. The difference will eventually build up again and the turning resumes, but now in the opposite direction. This pattern keeps repeating.

This behavior can be modeled as a system of three ordinary differential equations:

$$\begin{aligned} dx/dt &= \sigma(y-z) \\ dy/dt &= x(\rho-z)-y \\ dz/dt &= xy-\beta z \end{aligned}$$

x is the speed of the convective circulation with $x > 0$ representing a clockwise direction (and $x < 0$ being counter-clockwise). y is the temperature difference, and z the bottom-to-top linear temperature gradient. The constants σ, ρ, and β are system parameters, typically set to 10, 28, and 8/3.

lorenz.py (Listing 8.4) plots the model by treating the derivatives as difference equations and using Euler's forward method to step through time, calculating the partial derivatives at the current (x, y, z) in order to estimate the next coordinate.

```
DT = 0.01
N = 10000

def lorenz(xyz, s=10, p=28, b=8/3):
  """
  xyz : Point in 3D space as a tuple
  s, p, b : Parameters defining the Lorenz attractor
  Returns Lorenz attractor's partial derivatives at xyz
  """
  x, y, z = xyz
  x_dot = s*(y - x)
  y_dot = p*x - y - x*z
  z_dot = x*y - b*z
  return (x_dot, y_dot, z_dot)

xyz = [(0,0,0) for x in range(N+1)]
xyz[0] = (0, 1.0, 1.05)
for i in range(N):
  xyz[i+1] = tuple(map(lambda x,f: x+f*DT,
                                  xyz[i], lorenz(xyz[i])))
              # xyz[i] + lorenz(xyz[i]) * DT
fig = plt.figure(figsize=(7, 7))
ax = fig.add_subplot(projection ='3d')
xs, ys, zs = zip(*xyz)
ax.plot(xs, ys, zs, lw=0.5)
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
ax.set_title("Lorenz Attractor")
plt.show()
```

Listing 8.4. Implementing Lorenz

Fig. 8.19 is perhaps the most famous image in chaos theory (although the bifurcation diagram (Fig. 8.6) for the logistic map is a close contender).

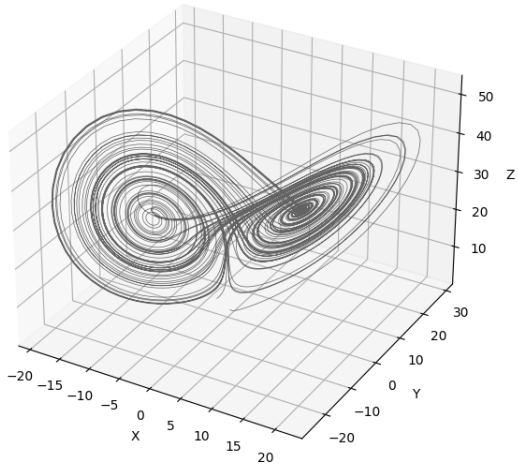


Figure 8.19. Plot of the Lorenz model

Lorenz soon noted that changing his initial conditions by even a tiny fraction would rapidly lead to totally different behavior – the SDIC hallmark of chaotic systems. This property is shown in Fig. 8.20 (lorenzSDIC.py) which uses the same lorenz() function as above but plots two curves, one starting at (0, 1.0, 1.05), the other at (0.001, 1.0, 1.05). Their x values are also charted separately to show that although the curves begin in a similar fashion, they soon diverge.

There's a strong resemblance between the Lorenz model and a butterfly's wings, and it's occasionally thought that this is the reason why SDIC is sometimes known as the butterfly effect. This isn't the case. Lorenz's original work didn't include a 3D plot, and he originally used a seagull, not a butterfly, as an example of how a small perturbation could lead to large changes in the weather system. Hilborn discusses the origins of the butterfly metaphor in his short paper "Sea gulls, butterflies, and grasshoppers", available online at `https://physics.csuchico.edu/ayars/427/handouts/AJP000425.pdf`.

Another variant of the lorenz() code appears in lorenzAnim.py which utilizes matplotlib's animation support to draw the curve incrementally. This makes its dynamic aspects much more apparent, including its clockwise and counterclockwise rotations around the two centers, and the variations in speed. Near a center, the curve slows down, almost to a stop, then speeds up as it moves further away. The two centers are at $(6\sqrt{2}, 6\sqrt{2}, 27)$ and $(-6\sqrt{2}, -6\sqrt{2}, 27)$

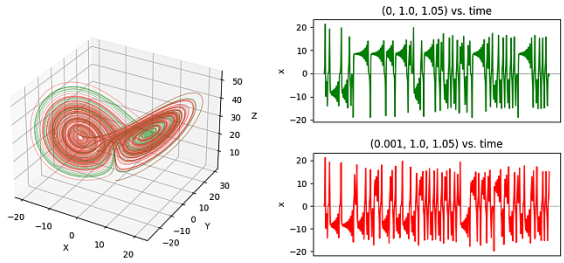


Figure 8.20. SDIC of the Lorenz model

Two snapshots of the animation are shown in Fig. 8.21. The large dot marks the current coordinate being drawn.

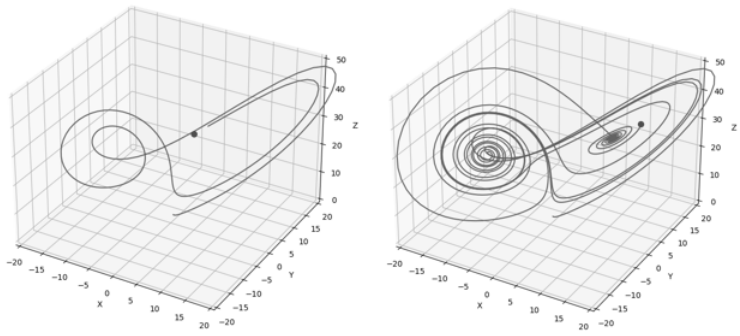


Figure 8.21. Animation of the Lorenz model

8.8 A Poincaré Section

One approach used by Lorenz to analyze the system was to plot Poincaré sections. This entails placing a constraint on one or more of the (x, y, z) coordinates to project the 3D curve onto a 2D surface or a 1D map.

Lorenz set $dZ/dt = 0$ which flattens the curve onto $z = b^{-1}xy$. Another way to understand this is that the derivative focuses only on the maximum and minimum z values. In fact, Lorenz generated a map of only the successive maximum values. He plotted the n-th maximum z value against the $(n + 1)$-th maximum, resulting in a graph similar to Fig. 8.22.

This captures the way that once a z value crosses a certain threshold, the coordinate jumps into an orbit around the other center, and the z component

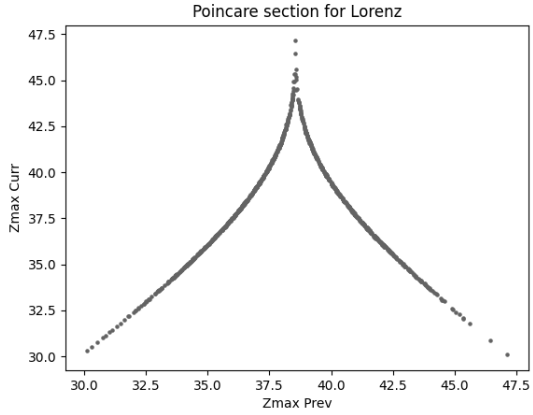


Figure 8.22. Poincaré section of the Lorenz model

shrinks. The value then starts increasing again until it reaches the threshold, then jumps back to the original orbit, and so on. Listing 8.5 (poincare.py) generates the Lorenz curve as before, but now looks for z points that are local maximums. These are stored in two lists which allow the n-th maximum z value to be plotted against the $(n+1)$-th value. Incidentally, the curve in Fig. 8.22 is sometimes called the *cusp map*.

```
currs = []
prevs = []
xyz = [(0,0,0) for x in range(N+1)]
xyz[0] = (0, 1.0, 1.05)

localMax = 0   # dummy start value
for i in range(N):
  xyz[i+1] = tuple(map(lambda x,f: x+f*DT,
                       xyz[i], lorenz(xyz[i])))
  if i > 100:   # let things settle down
    zOld = xyz[i-1][2]
    z = xyz[i][2]
    zNew = xyz[i+1][2]
    if zNew < z and zOld < z:
      prevMax = localMax
      localMax = z
      currs.append(localMax)
      prevs.append(prevMax)

plt.scatter(prevs[1:], currs[1:], s=5)  # s is point size
            # skip dummy value
plt.xlabel("Zmax Prev")
plt.ylabel("Zmax Curr")
```

```
plt.title("Poincare section for Lorenz")
plt.show()
```

Listing 8.5. Implementing Poincare

Lorenz used the fact that the absolute gradient of the map is > 1 at all points to argue that the underlying model is unstable.

As in previous examples, the curve is generated using Euler's forward step approximation, which in this case introduces some minor rounding errors on the right hand end. These disappear if we switch to a more accurate approximation such as 4th order Runge Kutta, but we decided not to complicate the code with that approach.

Exercises

(1) Modify logCobweb.py to explore the map $x_{n+1} = 1 + \frac{1}{2}\sin x_n$.

(2) Modify logCobweb.py to explore the map $x_{n+1} = 3x_n - x_n^3$. In particular consider the cobwebs starting at $x_0 = 1.9$ and 2.1.

(3) Modify bifurcation.py (Listing 8.1) to plot the following maps. Be sure to use a large enough range for both r and x to capture all the features of interest.

   (a) $x_{n+1} = r\cos x_n$
   (b) $x_{n+1} = r\tan x_n$
   (c) $x_{n+1} = rx_n - x_n^3$

(4) The constants used in the Lorenz model, σ, ρ, and β, are typically set to 10, 28, and 8/3. What happens when they are varied? Most researchers have left σ and β alone, and experimented with different values for ρ. Modify Listing 8.4, or one of its variants, to investigate the meaning of Fig. 8.23 which is taken from Section 9.5 of Strogatz [**Str15**].

   It's also interesting to try out much larger values for ρ, such as 166.3 (sometimes termed 'intermittent chaos'), 212 (noisy periodicity) and the interval $145 < \rho < 166$ (period-doubling).

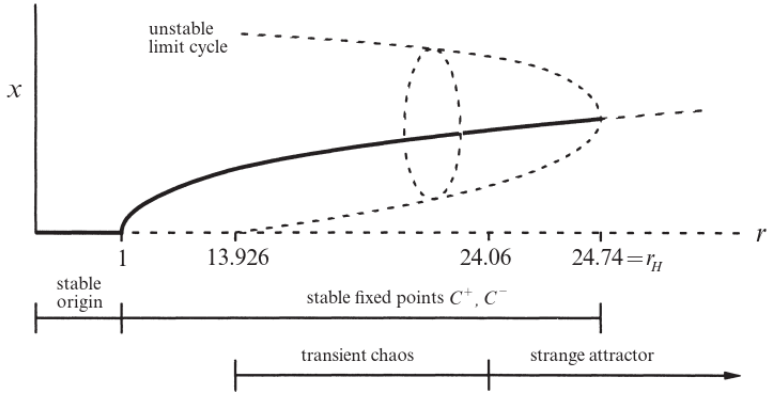


Figure 8.23. Varying ρ in the Lorenz model

9

Computer Graphics Transformations

Why Study Computer Graphics Transformations?

- **They define how graphical objects move, rotate, and scale**. Transformations give a precise mathematical language for changing an object's position, size, and orientation, which is essential for animation and modelling. Projective transformations are necessary to convert 3D scenes into 2D images.
- **They allow consistent, efficient representations of complex scenes**. Using matrices to express transformations makes it possible to combine operations cleanly. Also, matrix and vector operations are fast on modern hardware.
- **They enable different coordinate systems to work together**. Camera coordinates, world coordinates, object coordinates, and screen coordinates are all linked by transformations.
- **They are the foundation for advanced techniques**. Texture mapping, skeletal animation, ray tracing, collision detection, and physics simulations all depend on transformation mathematics.

9.1 Introduction

This chapter looks at how to employ 2D and 3D geometrical transformations, such as translation, scaling, and rotation, in computer graphics.

For the most part, we'll rely on two Python classes, `HC2Mat.py` and `HC3Mat.py`, and render shapes using Matplotlib's polygons. We'll also use Matplotlib's Affine2D class, which covers the same ground as our `HC2Mat.py` but can manipulate images and SVG drawings.

Computer graphics rests atop linear algebra due to its use of vectors, matrices, and linear, affine, and projective transformations. A good introduction can be found in *Elementary Linear Algebra* by Stephen Andrilli and David Hecker [**AH22**], and some of my examples come from their discussion of computer graphics. A truly excellent set of short videos on the subject is *Essence of linear algebra* by Grant Sanderson (3Blue1Brown) at `https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab`. A good textbook focusing directly on graphics is *Computer Graphics: Principles and Practice*, by John Hughes *et al.* [**Hug14**].

9.2 2D Transformations

We can translate points in the (x, y) plane to new positions by adding translation amounts:

$$x' = x + d_x, \quad y' = y + d_y.$$

If we define the column vectors:

$$P = \begin{bmatrix} x \\ y \end{bmatrix}, \quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \quad T = \begin{bmatrix} d_x \\ d_y \end{bmatrix}$$

then the translation can be succinctly expressed as $P' = P + T$.

Points can be scaled by s_x along the x-axis, and by s_y along the y-axis, with the multiplications:

$$x' = s_x \cdot x, \quad y' = s_y \cdot y.$$

In matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{or} \quad P' = S \cdot P.$$

where S is the scaling matrix. Note that this defines a scaling about the origin; techniques for scaling about other points are discussed below.

Points can be rotated through an angle θ about the origin with

$$x' = x \cdot \cos\theta - y \cdot \sin\theta, \quad y' = x \cdot \sin\theta + y \cdot \cos\theta.$$

Using matrices:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{or} \quad P' = R \cdot P$$

where R is the rotation matrix. Positive angles are measured counterclockwise from the x-axis toward the y-axis. For negative (clockwise) angles, the identities $\cos(-\theta) = \cos\theta$ and $\sin(-\theta) = -\sin\theta$ can be used.

The rotation equation is easily derived from Fig. 9.1, in which a rotation by θ transforms $P(x, y)$ into $P'(x', y')$. Because the rotation is about the origin, the distances from the origin to P and to P', labeled r in the figure, are equal.

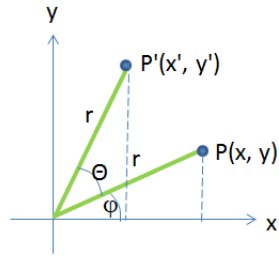


Figure 9.1. Derivation of the rotation equation

By trigonometry,

$$x = r \cdot \cos\phi, \quad y = r \cdot \sin\phi$$

With the angle sum rule, x' and y' are:

$$\begin{aligned} x' &= r \cdot \cos(\theta + \phi) = r \cdot \cos\phi \cdot \cos\theta - r \cdot \sin\phi \cdot \sin\theta, \\ y' &= r \cdot \sin(\theta + \phi) = r \cdot \cos\phi \cdot \sin\theta + r \sin\phi \cdot \cos\theta \end{aligned}$$

These equations can be rewritten in terms of x and y:

$$\begin{aligned} x' &= x \cdot \cos\theta - y \cdot \sin\theta, \\ y' &= x \cdot \sin\theta + y \cdot \cos\theta \end{aligned}$$

which can be expressed in the matrix form from above.

9.3 Homogeneous Coordinates

The matrix representations for translation, scaling, and rotation from the last section are:

$$P' = T + P, \quad P' = S \cdot P, \quad P' = R \cdot P.$$

Translation is treated differently from scaling and rotation, but we'd prefer to express all three in a consistent manner so that transformations can be combined more easily.

One answer is to define points using *homogeneous coordinates*, which enables every transformations to be defined using matrix multiplication alone. A point (x, y) becomes (x, y, W), and we assume that (x, y, W) and (x', y', W') represent the same point when one is a multiple of the other (e.g. $(2, 3, 6)$ and $(4, 6, 12)$ are the same). If W is nonzero, we divide through by its value, so (x, y, W) becomes $(x/W, y/W, 1)$, and $(x/W, y/W)$ denotes the point in 2D space. However,

if $W = 0$ then the point is deemed to be located at infinity, which is useful for projective transformations such as perspectives and warping.

Now that 2D points are triplets, our transformation matrices become 3×3. This allows translation to be encoded using a single matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 0 & d_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This can be expressed as

$$P' = T(d_x, d_y) \cdot P$$

where

$$T(d_x, d_y) = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 0 & d_y \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplying out the matrix produces the desired translations:

$$x' = x + d_x, \quad y' = y + d_y.$$

What happens if a point P is translated by $T(d_{x_1}, d_{y_1})$ to P' and then by $T(d_{x_2}, d_{y_2})$ to P''? We expect a combined translation $T(d_{x_1} + d_{x_2}, d_{y_1} + d_{y_2})$. Initially:

$$\begin{aligned} P' &= T(d_{x_1}, d_{y_1},) \cdot P \\ P'' &= T(d_{x_2}, d_{y_2}) \cdot P' \end{aligned}$$

Substituting the first equation into the second, we obtain:

$$\begin{aligned} P'' &= T(d_{x_1}, d_{y_1}) \cdot (T(d_{x_2}, d_{y_2}) \cdot P) \\ &= (T(d_{x_2}, d_{y_2}) \cdot T(d_{x_1}, d_{y_1})) \cdot P \end{aligned}$$

The matrix product for $T(d_{x_2}, d_{y_2}) \cdot T(d_{x_1}, d_{y_1})$ is

$$\begin{bmatrix} 1 & 0 & d_{x_2} \\ 0 & 0 & d_{y_2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & d_{x_1} \\ 0 & 0 & d_{y_1} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_{x_1} + d_{x_2} \\ 0 & 0 & d_{y_1} + d_{y_2} \\ 0 & 0 & 1 \end{bmatrix}.$$

This shows that the net translation is indeed $T(d_{x_1} + d_{x_2}, d_{y_1} + d_{y_2})$.

Scaling is represented by

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

If we define

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

then

$$P' = S(s_x, s_y) \cdot P.$$

Let's check that successive scalings are multiplicative. Given

$$\begin{aligned} P' &= S(s_{x_1}, s_{y_1}) \cdot P \\ P'' &= S(s_{x_2}, s_{y_2}) \cdot P' \end{aligned}$$

then substituting the first equation into the second, we get

$$\begin{aligned} P'' &= S(s_{x_2}, s_{y_2}) \cdot (S(s_{x_1}, s_{y_1}) \cdot P) \\ &= (S(s_{x_2}, s_{y_2}) \cdot S(s_{x_1}, s_{y_1})) \cdot P \end{aligned}$$

The matrix product $S(s_{x_2}, s_{y_2}) \cdot S(s_{x_1}, s_{y_1})$ is

$$\begin{bmatrix} s_{x_2} & 0 & 0 \\ 0 & s_{y_2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_{x_1} & 0 & 0 \\ 0 & s_{y_1} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{x_2} \cdot s_{x_1} & 0 & 0 \\ 0 & s_{y_2} \cdot s_{y_1} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus, scalings are multiplicative.

Rotations become

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

If we define

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

then

$$P' = R(\theta) \cdot P.$$

Consecutive rotations are encoded by matrix multiplication, but the combined result *adds* the rotations. This can be proved using similar angle sum rewrites as employed for the rotations in the previous subsection.

Shearing is another common transformation, either along the x- or y-axis. The x-axis operation is represented by

$$SH_x = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The product $SH_x(x, y)$ is $[x + ay, y, 1]^T$, demonstrating the proportional change in x as a function of y.

Similarly,

$$SH_y = \begin{bmatrix} 1 & 0 & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

shears along the y-axis.

Reflection about a line through the origin (i.e. by using $y = mx$) becomes the matrix

$$\frac{1}{1+m^2}\begin{bmatrix} 1-m^2 & 2m & 0 \\ 2m & m^2-1 & 0 \\ 0 & 0 & 1+m^2 \end{bmatrix}$$

This can be understood by assuming that the line forms an angle θ with the x-axis. Then the reflection can be divided into three steps:

(1) rotate the line by $-\theta$ to make it horizontal;

(2) invert the y coordinate;

(3) rotate the line back by θ.

These three operations become rotation, inversion, and rotation matrices composed using multiplication. Further simplifications are possible by noting that since $y = mx$, then $\tan\theta = m$, and so $1 + m^2 = 1/\cos^2\theta$.

The reflection transformation:

$$\begin{aligned} F &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ \sin\theta & -\cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \cos^2\theta\begin{bmatrix} 1 & -\tan\theta & 0 \\ \tan\theta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & \tan\theta & 0 \\ \tan\theta & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \frac{1}{1+m^2}\begin{bmatrix} 1 & -m & 0 \\ m & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & m & 0 \\ m & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \frac{1}{1+m^2}\begin{bmatrix} 1-m^2 & 2m & 0 \\ 2m & m^2-1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Note that the matrices are organized in a right-to-left order since we're mimicking function composition.

In the case when the line of reflection is the y-axis, the matrix reduces to

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Do you see why?

9.4 Composition of 2D Transformations

Consider the rotation of a shape about some arbitrary point $P_1(x_1, y_1)$. R assumes that any rotation is about the origin, so we must bracket the rotation with suitable translations:

(1) Translate P_1 to the origin (and move the shape by the same amount);

(2) Rotate the shape about the origin using R;

(3) Translate the point at the origin back to P_1 (and move the rotated shape by the same amount).

The translation to the origin will be by $(-x_1, -y_1)$, and the translation back (x_1, y_1). The net transformation, performed in right-to-left order, is:

$T(x_1, y_1) \cdot R(\theta) \cdot T(-x_1, -y_1)$

$$\begin{aligned} &= \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta & -\sin\theta & x_1(1-\cos\theta) + y_1 \sin\theta \\ \sin\theta & \cos\theta & y_1(1-\cos\theta) - x_1 \sin\theta \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

A similar translation-bracketing approach is needed to scale an object about an arbitrary point: translate P_1 to the origin (along with the shape), then scale, then translate back to P_1 (along with the scaled shape). The net transformation is:

$T(x_1, y_1) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1)$

$$= \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_1(1-s_x) \\ 0 & s_y & y_1(1-s_y) \\ 0 & 0 & 1 \end{bmatrix}$$

Suppose we want to scale *then* rotate a shape about P_1? Crucially, we only need to utilize translation-bracketing once, so the composite matrix is made up of *four* operations performed in a right-to-left order:

$$T(x_1, y_1) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1).$$

9.5 Transforming a Knee

Let's illustrate how to manipulate the two-dimensional figure (a "Knee") shown in Fig. 9.2 using transformations implemented in Python. (The examples in this section (but not the code) comes from *Elementary Linear Algebra* by Stephen Andrilli and David Hecker [**AH22**].)

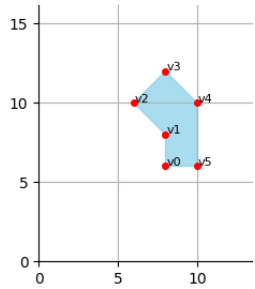


Figure 9.2. The "Knee" Polygon

The shape is encoded as a list of tuples:

```
coords = [(8,6),(8,8),(6,10),(8,12),(10,10),(10,6)]
```

Each tuple is a pair of x- and y-coordinates, but is converted into a column vector using homogeneous coordinate ($[x, y, 1]^T$) while the shape is being transformed, which means that the matrices are structured as in the previous section. The results are converted back to a list of tuples before they're rendered, which is done by converting them into a Matplotlib polygon.

Fig. 9.2 was generated using the following lines in `transformPoly.py`:

```
_, ax = plt.subplots()

coords = [(8,6),(8,8),(6,10),(8,12),(10,10),(10,6)]

m = identMat()  # the identity matrix; a 'do-nothing' op.
printMat(m)
nCoords = [applyMat(m, c) for c in coords]

plotPoly(ax, nCoords, 'skyblue')
plt.show()
```

identMat(), printMat(), applyMat(), and plotPoly() are imported from `HC2Mat.py` which collects functions for representing 2D homogeneous coordinates and their transformations. For example, identMat() returns an identity matrix:

```
def identMat():
  return [ [1, 0, 0],
```

```
          [0, 1, 0],
          [0, 0, 1] ]
```

plotPoly() plots a list of coordinates as a closed polygon and labels the vertices.

```
def plotPoly(ax, coords, color):
  poly = Polygon(coords, closed=True, color=color, alpha=0.7)
  ax.add_patch(poly)
  # Label each vertex
  for i, (x, y) in enumerate(coords):
    ax.plot(x, y, 'o', color='red', ms=4)  # small red dot
    ax.text(x + 0.05, y + 0.05, f'v{i}',
                    fontsize=8, color='black')
```

`Polygon()` is described at `https://matplotlib.org/stable/api/patches_api.html`.

applyMat() applies a transformation to a list of coordinates by multiplying the matrix to each coordinate after converting it to the vector $[x, y, 1]^T$:

```
def applyMat(matrix, coord):
  # apply the matrix to a (x,y) coord, returning
  # a new (x,y) coord
  vec = [coord[0], coord[1], 1]  # homogeneous
  nVec = [0, 0, 0]
  for i in range(3):
    for k in range(3):
      nVec[i] += matrix[i][k] * vec[k]
  return (nVec[0]/nVec[2], nVec[1]/nVec[2])
```

Rotation. Suppose we rotate "Knee" through an angle of 90° about the point (12, 6). We replace each (x, y) with its vector $[x, y, 1]^T$, then translate from (12, 6) to (0, 0) in order to center the point at the origin. We perform a rotation through 90° about the origin, then translate back to (12, 6). The net effect of these operations is

$$\underbrace{\begin{bmatrix} 1 & 0 & 12 \\ 0 & 1 & 6 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (0,0) back to (12,6)}} \underbrace{\begin{bmatrix} \cos 90° & -\sin 90° & 0 \\ \sin 90° & \cos 90° & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{rotate about (0,0) through angle 90°}} \underbrace{\begin{bmatrix} 1 & 0 & -12 \\ 0 & 1 & -6 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (12,6) to (0,0)}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This reduces to

$$\begin{bmatrix} 0 & -1 & 18 \\ 0 & 1 & -6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

We apply this transformation to all the vertices of the "Knee" to obtain

$$\begin{bmatrix} 0 & -1 & 18 \\ 0 & 1 & -6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 8 & 8 & 6 & 8 & 10 & 10 \\ 6 & 8 & 10 & 12 & 10 & 6 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 12 & 10 & 8 & 6 & 8 & 12 \\ 2 & 2 & 0 & 2 & 4 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The (x, y) coordinates are saved as a list of tuples, and drawn as a polygon in Fig. 9.3.

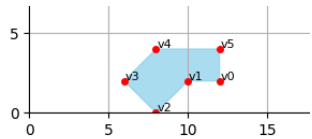


Figure 9.3. The "Knee" rotated through 90° about $(12, 6)$

In transformPoly.py, the transformation is encoded as:

```
m = multMatsList([ transMat(12,6), rotMat(90), transMat(-12,-6) ])
```

Note the right-to-left ordering of the list passed to multMatsList(). transMat() and rotMat() come from HC2Mat.py:

```
def transMat(a, b):
  # translate by 'a' along the x-axis, 'b' along y-axis
  return [ [1, 0, a],
           [0, 1, b],
           [0, 0, 1] ]

def rotMat(angle):
  # rotate by angle degrees counter-clockwise around origin
  theta = math.radians(angle)
  return [ [math.cos(theta), -math.sin(theta), 0],
           [math.sin(theta),  math.cos(theta), 0],
           [             0,                0,  1]  ]
```

Reflection. Let's reflect the original "Knee" about the line $y = -3x + 30$. In this case, $m = -3$, and the y intercept $b = 30$. As before, we replace (x, y) with its equivalent vector $[x, y, 1]^T$, then translate from $(0, 30)$ to $(0, 0)$ in order to lower the line 30 units vertically so its intercept passes through the origin. Now, we perform a reflection about the resulting line $y = -3x$, then translate back to $(0, 30)$. The net effect is to reflect each vertex of the "Knee"about the line $y = -3x + 30$ in three steps:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 30 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (0,0) back to (0,30)}} \underbrace{\frac{1}{1+(-3)^2}\begin{bmatrix} 1-(-3)^2 & 2(-3) & 0 \\ 2(-3) & (-3)^2-1 & 0 \\ 0 & 0 & 1+(-3)^2 \end{bmatrix}}_{\text{reflect about the line } y=-3x} \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -30 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (0,30) to (0,0)}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This reduces to

$$\frac{1}{10}\begin{bmatrix} -8 & -6 & 180 \\ -6 & 8 & 60 \\ 0 & 0 & 10 \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Applying this matrix to all the vertices of "Knee" produces

$$\frac{1}{10}\begin{bmatrix} -8 & -6 & 180 \\ -6 & 8 & 60 \\ 0 & 0 & 10 \end{bmatrix}\begin{bmatrix} 8 & 8 & 6 & 8 & 10 & 10 \\ 6 & 8 & 10 & 12 & 10 & 6 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 8 & 6.8 & 7.2 & 4.4 & 4 & 6.4 \\ 6 & 7.6 & 10.4 & 10.8 & 8 & 4.8 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The (x, y) coordinates are drawn as a polygon in Fig. 9.4.

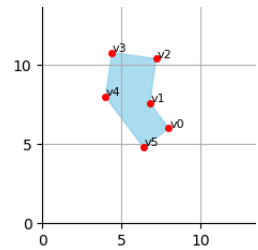


Figure 9.4. "Knee" reflection about line $y = -3x + 30$

The transformation as coded in `transformPoly.py` as:

```
m = multMatsList([ transMat(0,30), reflectMat(-3), transMat(0,-30) ])
```

reflectMat() in `HC2Mat.py` is:

```
def reflectMat(m):
  # reflect about line y = mx
  den = 1 + m*m
  return [ [(1-m*m)/den,     2*m/den,  0],
           [    2*m/den, (m*m-1)/den,  0],
           [          0,           0,  1] ]
```

Scaling. Let's scale "Knee" about the point $(6, 10)$ with a scaling factor of $c = 1/2$ in the x-direction and $d = 4$ in the y-direction. As before, the resizing must be bracketed by translations:

$$\underbrace{\begin{bmatrix} 1 & 0 & 6 \\ 0 & 1 & 10 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (0,0) back to (6,10)}} \underbrace{\begin{bmatrix} 1/2 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{scale about (0,0) using scale factors } \frac{1}{2} \text{ and 4, respectively}} \underbrace{\begin{bmatrix} 1 & 0 & -6 \\ 0 & 1 & -10 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (6,10) to (0,0)}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1/2 & 0 & 3 \\ 0 & 4 & -30 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The scaling of "Knee" produces

$$\begin{bmatrix} \frac{1}{2} & 0 & 3 \\ 0 & 4 & -30 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 8 & 8 & 6 & 8 & 10 & 10 \\ 6 & 8 & 10 & 12 & 10 & 6 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 7 & 7 & 6 & 7 & 8 & 8 \\ -6 & 2 & 10 & 18 & 10 & -6 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The polygon built from the (x, y) coordinates is shown in Fig. 9.5. Two of the vertices have negative y-values, so aren't in the region displayed by Matplotlib.

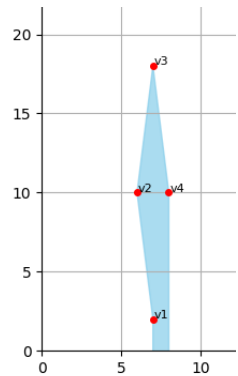


Figure 9.5. "Knee" scaling with c=1/2, d=4 about (6, 10)

The transformation in transformPoly.py is:

```
m = multMatsList([ transMat(6,10), scaleMat(0.5,4), transMat(-6,-10) ])
```

scaleMat() in HC2Mat.py:

```
def scaleMat(a, b):
  # scale by 'a' along the x-axis, 'b' along y-axis
  return [ [a, 0, 0],
           [0, b, 0],
           [0, 0, 1] ]
```

9.6 Multiple Transformations

As a grand finale, let's rotate "Knee" through 300° about (8, 10), then reflect it about the line $y = -(1/2)x + 20$. The key values are $\theta = 300°, m = -1/2$, and $b = 20$, and the composition utilizes six matrices:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 20 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (0,0) back to (0,20)}} \underbrace{\left(\frac{1}{1+(-1/2)^2}\right)\begin{bmatrix} 1-(-1/2)^2 & 2(-1/2) & 0 \\ 2(-1/2) & (-1/2)^2-1 & 0 \\ 0 & 0 & 1+(-1/2)^2 \end{bmatrix}}_{\text{reflect about the line } y=(-1/2)x} \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -20 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (0,20) to (0,0)}}$$

$$\underbrace{\begin{bmatrix} 1 & 0 & 8 \\ 0 & 1 & 10 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (0,0) back to (8,10)}} \underbrace{\begin{bmatrix} \cos 300° & -\sin 300° & 0 \\ \sin 300° & \cos 300° & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{rotate about (0,0) through angle 300°}} \underbrace{\begin{bmatrix} 1 & 0 & -8 \\ 0 & 1 & -10 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (8,10) to (0,0)}}$$

The first three matrices deal with the second part of the transformation – the reflection in a line not through the origin, while the last three are for the rotation around a point. The composition reduces to (approximately)

$$\begin{bmatrix} 0.993 & 0.120 & 3.661 \\ 0.120 & -0.993 & 28.571 \\ 0 & 0 & 1 \end{bmatrix}$$

Applying this matrix to "Knee" produces

$$\begin{bmatrix} 0.993 & 0.120 & 3.661 \\ 0.120 & -0.993 & 28.571 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 8 & 8 & 6 & 8 & 10 & 10 \\ 6 & 8 & 10 & 12 & 10 & 6 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 12.32 & 12.56 & 10.81 & 13.04 & 14.79 & 14.31 \\ 23.57 & 21.59 & 19.36 & 17.61 & 19.84 & 23.81 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Its coordinates are shown in Fig. 9.6.

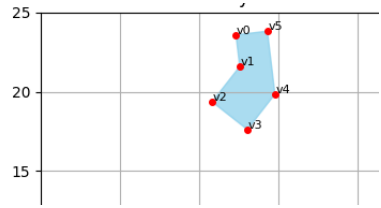


Figure 9.6. "Knee" rotation then reflection

The relevant line in `transformPoly.py`:

```
m = multMatsList( [
      transMat(0,20), reflectMat(-0.5), transMat(0,-20),
      transMat(8,10), rotMat(300), transMat(-8,-10) ])
```

9.7 Baravelle Spirals

A Baravelle spiral is a geometric shape constructed by subdividing a large regular polygon into triangles. The construction bisects each of the sides and joins the midpoints consecutively to create the triangles, leaving a smaller version of the polygon inside. The procedure is repeated, and the spirals are formed from adjacent nested triangles turning counter-clockwise, which shrink and converge upon the polygon's center. The National Curve Bank (`https://old.nationalcurvebank.org/baravelle/baravelle.htm`) has an online tool for generating Baravelle spirals for several types of regular polygon, as shown in Fig. 9.7.

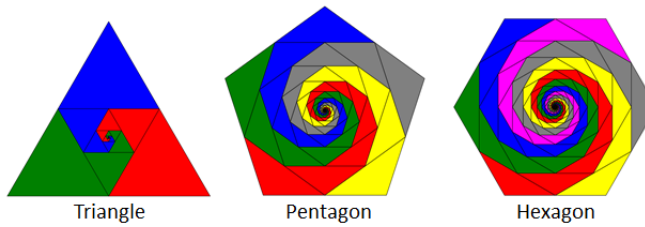


Figure 9.7. Baravelle Spirals

For further details, the paper by Suzanne Harper explains how these curves can be used to introduce infinite series in secondary mathematics [**Har01**].

baravelle.py generates baravelle spirals starting from a large square, as shown in Fig. 9.8.



Figure 9.8. Baravelle Spirals for a Square

The code begins by placing four copies of a right-angled triangle at the corners of a square, as in Fig. 9.9.

The code for this stage is in createColorCoords():

```
def createColorCoords():
  coords = [ (0,0), (SIZE,0), (SIZE,SIZE)] # basic triangle
  ''' set start positions of triangles at the
    four corners of a 2*SIZE square centered at origin
  '''
```

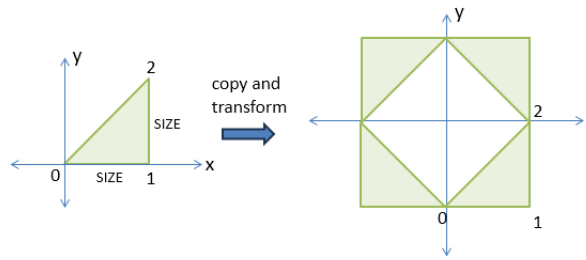


Figure 9.9. Initializing the Baravelle Square

```
m = multMatsList([ transMat(SIZE,0), rotMat(90) ]) # top right
coords0 = [applyMat(m, c) for c in coords]

m = multMatsList([ transMat(0,SIZE), rotMat(180) ]) # top left
coords1 = [applyMat(m, c) for c in coords]

m = multMatsList([ transMat(-SIZE,0), rotMat(270) ]) # bottom left
coords2 = [applyMat(m, c) for c in coords]

m = multMatsList([ transMat(0,-SIZE) ])   # bottom right
coords3 = [applyMat(m, c) for c in coords]

# specify a fill color for each triangle
return  [(coords0,'skyblue'), (coords1,'lightgreen'),
         (coords2, 'violet'), (coords3, 'salmon')]
```

Spiral generation is carried out by repeatedly shrinking, rotating, and moving a triangle to the midpoint of the diagonal of its parent (see Fig. 9.10).

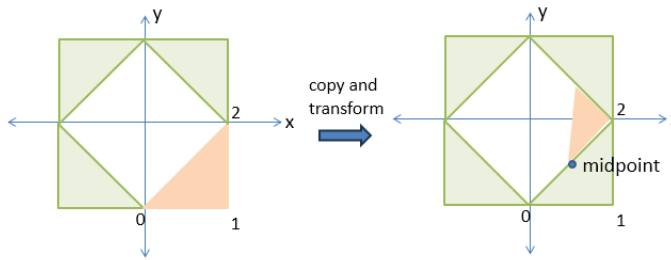


Figure 9.10. Shrink, Rotate, and Move Triangle to the Midpoint

This could be expressed recursively, but is just as simple to write iteratively:

```
cCoords = createColorCoords()
for coords, color in cCoords:
  drawPoly(ax, coords, color)
  for i  in range(STEPS):
    coords = moveIn(coords)
    drawPoly(ax, coords, color)
```

moveIn() scales, rotates, and moves a triangle:

```
def moveIn(coords):
  pt0 = coords[0]
  nPt0 = tuple((a+b)/2 for a, b in zip(coords[0], coords[2]))
         # midpoint of diagonal
  m = multMatsList([ transMat(nPt0[0], nPt0[1]),
         rotMat(ANGLE), scaleMat(SCALE, SCALE),
         transMat(-pt0[0], -pt0[1])  ])
  return [applyMat(m, c) for c in coords]
```

The bracketed translation in moveIn() is a little different from before. The triangle is moved back to the origin by using its first vertex (pt0), but is positioned after scaling and rotation at the midpoint of its parent's diagonal (nPt0).

9.8 Hilbert Curves Again

We've already met the Hilbert curve family in Section 7.8 on fractals. The approach used there was based around drawing a line using turtle operations such as Turtle.forward() and Turtle.setHeading(), and by calling four inter-linked recursive functions for drawing the U_i, D_i, R_i, and L_i move sequences. However, a more intuitive way to view these moves is as rotations applied to a shape represent H_0 (Fig. 9.11). (Note that H_0 is a "U" shape; the arrowhead was added by me to indicate the drawing order – the drawing begins at the top of the right arm then moves clockwise to the top of the left arm.)

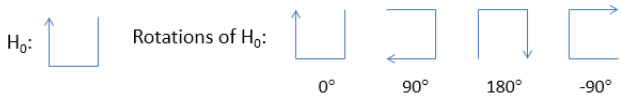


Figure 9.11. Four Rotations of H_0

As in the fractals section, H_i is created by utilizing four versions of H_{i-1} composed together. However, they're modified using transformations rather than with turtle functions. The necessary steps to convert four copies of H_0 into H_1 are shown in Fig. 9.12.

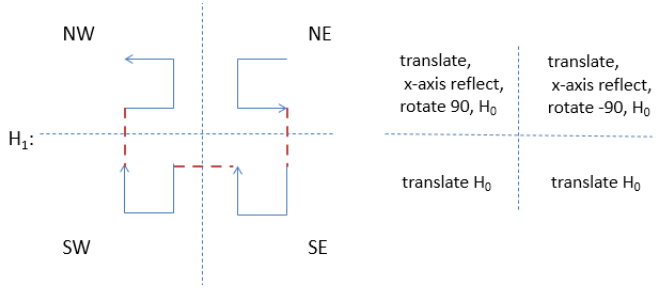


Figure 9.12. Four H_0s Transformed into H_1

The transformations in the four quadrants of Fig. 9.12 are applied to H_0 in the usual right-to-left order, producing four rotated and translated copies. They are linked by the dashed lines in the figure to produce H_1.

You should try out these same four transformations on H_1 to confirm that they generate H_2. Indeed, these transformations remain the same for any H_i produced from H_{i-1}, and so are conveniently collected together in transformCurve() in hilbert.py:

```
L = 50   # length of one side of H0

def transformCurve(h):
  # Use hilbert curve h to generate next one

  # build parts in 4 quadrants
  m = multMatsList([ transMat(L,L),
                reflectXMat(), rotMat(-90) ])
  ne = [applyMat(m, c) for c in h]  # north east quadrant

  se = [applyMat(transMat(L,-L), c) for c in h]

  sw = [applyMat(transMat(-L,-L), c) for c in h]

  m = multMatsList([ transMat(-L,L),
                reflectXMat(), rotMat(90) ])
  nw = [applyMat(m, c) for c in h]

  hNew = ne + se + sw + nw # concatenate parts
  return [applyMat(scaleMat(0.5,0.5), c) for c in hNew]
```

The composition of the four shapes, which is denoted by the three dashed lines in Fig. 9.12, requires nothing more than list concatenation since the lines are stored as lists of tuples. However, care must be taken that the lists are in the correct drawing order, which requires the NW and NE shapes to be reflected in the x-axis.

The top-level of the program could use recursion, but (as with the baravelle spirals) iteration is just as simple:

```
h = [ (L/2,L/2), (L/2,-L/2), (-L/2,-L/2), (-L/2,L/2)]
      # H0 centered on origin (a U-shape)
for i in range(hOrder-1):
  h = transformCurve(h)
```

9.9 Matplotlib's Affine2D Classes

The drawing of a shape by Matplotlib is achieved by converting its list of coordinates into a polygon (`https://matplotlib.org/stable/api/patches_api.html`). The relevant lines in plotPoly() in HC2Mat.py are:

```
def plotPoly(ax, coords, color):
  poly = Polygon(coords, closed=True, color=color, alpha=0.7)
  ax.add_patch(poly)

  # Label each vertex
  #  :
```

It's not surprising that Matplotlib also offers geometric transformations that can be applied to shapes via subclassses of matplotlib.transform (see `https://matplotlib.org/stable/api/transformations.html`). To keep things simple, we'll focus on the Affine2D class (`https://matplotlib.org/stable/api/transformations.html#matplotlib.transforms.Affine2D`) which defines its transformations in terms of 2D homogeneous coordinates (i.e. as 3×3 matrices). The matrices are implemented using Numpy arrays, but it's possible to ignore this complication when using the class, which includes methods such as translate(), rotate(), scale(), and skew() (but not reflection).

The earlier "Knee" examples were implemented in transformPoly.py. To aid comparison, we've reimplemented them using Affine2D transformations in affinePoly.py. The "Knee"'s vertices are created in the same way as before, but transformations are applied to its polygon:

```
coords = [(8,6),(8,8),(6,10),(8,12),(10,10),(10,6)]
poly = Polygon(coords, closed=True, color='skyblue', alpha=0.7)
```

Note that the coordinates don't require W homogeneous values, but they are added by Affine2D when transformations are applied.

The last example in transformPoly.py rotates "Knee" 300° around $(8, 10)$, then reflects it around the line $y = -0.5x + 20$. The resulting matrix is:

```
m = multMatsList( [
      transMat(0,20), reflectMat(-0.5), transMat(0,-20),
      transMat(8,10), rotMat(300), transMat(-8,-10) ])
```

The corresponding transformation using Affine2D in affinePoly.py is:

```
trfm = Affine2D().translate(-8,-10). \
            rotate_deg(300).translate(8,10) + \
       Affine2D().translate(0,-20) + \
       reflect(-0.5) + \
       Affine2D().translate(0,20)
```

A key difference is that Affine2D transformations are performed in left-to-right order. Also, although Affine2D allows the chaining of operations using ".", this is not possible when using reflection. Affine2D doesn't include a reflection method, but one can be defined manually, as in our reflect():

```
def reflect(m):
  ''' Build reflection Affine2D instance using the matrix
        a c e
        b d f
        0 0 1
      with from_values(a,b,c,d,e,f)
  '''
  den = 1 + m*m
  return Affine2D.from_values(
          (1-m*m)/den, 2*m/den, 2*m/den, (m*m-1)/den, 0, 0)
```

Sadly, the matrix returned by reflect() can not be chained, but Affine2D operations can also be composed using "+", which we used to create `trfm` above.

Drawing the polygon in affinePoly.py is similar to transformPoly.py:

```
poly.set_transform(trfm + ax.transData) # add axis offset
ax.add_patch(poly)
```

One new thing is to explicitly add a transformation so the shape is correctly positioned relative to the axes. The output is shown in Fig. 9.13, and should be compared to the image produced by transformPoly.py in Fig. 9.6.

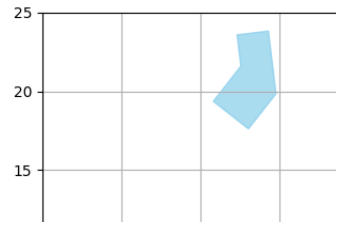


Figure 9.13. "Knee" rotation then reflection Using Affine2D

One reason for preferring Affine2D over our HC2Mat.py is efficiency: Affine2D utilizes Numpy arrays for its matrices rather than 2D lists, and so will be faster

when transforming shapes with many coordinates. Two other reasons for preferring Affine2D is that it also works with images and SVG (Scalable Vector Graphics) polylines, as detailed in the next three subsections.

9.10 Affine2D Images

imPlot.py loads a picture of a cat, displays it and a copy that is reflected, rotated, and translated (see Fig. 9.14).

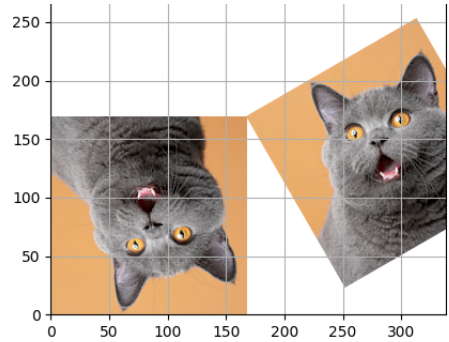


Figure 9.14. Two Cats, one Transformed

The cat on the left is the original image, which is shown with its origin at the *top-left* corner and its y-axis increasing downwards – the standard way that PNG images are encoded. The relevant code snippets in implot.py are:

```
_, ax = plt.subplots()

im = mpimg.imread('cat.png')  # load image
height, width = im.shape[0], im.shape[1]
    :  # more code
ax.imshow(im)
```

The image on the right of Fig. 9.14 has been reflected in the x-axis, rotated, and translated. The relevant lines are:

```
show2 = ax.imshow(im.copy())   # show first, then transform
trans = Affine2D().scale(1, -1). \
                   rotate_deg(30). \
                   translate(width, height)
show2.set_transform(trans + ax.transData)
```

There are two things of note: the first is that we didn't use our Affine2D reflect() function to reflect the image but got by with a 'tricky' use of scaling. A second point is that it's necessary to call imshow() on the image *before* applying the transformation.

9.11 Affine2D Paths

A good visual overview of Matplotlib shapes can be found at `https://matplotlib.org/stable/api/patches_api.html`. A particularly powerful type is PathPatch (`https://matplotlib.org/stable/api/_as_gen/matplotlib.patches.PathPatch.html#matplotlib.patches.PathPatch`) which converts a Path into a renderable shape (a patch). A path is more general than just a list of coordinates; it can be a series of possibly disconnected, possibly closed, line and/or curve segments. An introduction to creating path-based shapes can be found at `https://matplotlib.org/stable/users/explain/artists/paths.html`.

Another approach, better suited to drawing complicated shapes, is to use a SVG editor (e.g. the "Free SVG Editor" at `https://freesvgeditor.com/en/svg-editor-online`). The result is an SVG file (a text file using XML) which typically encodes multiple SVG paths. These are similar to paths in Matplotlib, but may be considerably more complex. Fortunately, there are parsing tools that can convert SVG paths into Matplotlib ones; we've used the third-party svgpath2mpl module which offers the parse_path() function (`https://github.com/nvictus/svgpath2mpl`). The result of parsing an SVG file will usually be multiple Matplotlib paths, which are converted into a single shape using the Matplotlib PathCollection class.

This approach is admittedly a bit complicated, so we've included sample code in ShowSVG.py. Fig. 9.15 shows two windows: the one on the left is a SVG viewer's display of house.svg which we quickly knocked together with the "Free SVG Editor". The window on the right shows the plot generated by ShowSVG.py when passed that file.

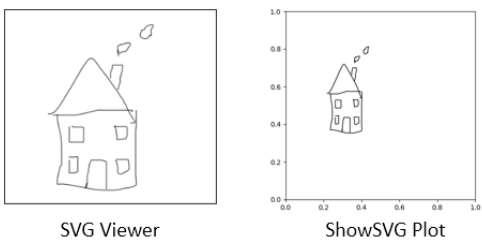


Figure 9.15. SVG and Matplotlib Houses

It's reasonable if you feel somewhat reluctant to follow this approach since SVG is a *very* sophisticated format, including features such as image compression, interactivity, and animation. The good news is that svgpath2mpl's parse_path()

will accept much simpler input, including a list of operations that specifies a single SVG "d" path. For example, in the text file f.path you'll find:

```
M .1, .1
L .9, .1
L .9, .2
L .2, .2
L .2, .4
L .6, .4
L .6, .5
L .2, .5
L .2, .9
L .1, .9
L .1, .1
```

Mozilla offers a good tutorial on SVG paths at `https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorials/SVG_from_scratch/Paths`, but if you only know that 'M' and 'L' are short for "Move to" and "Line to", then you can probably already draw this 'F' shape on some graph paper.

We've put together sample code in ShowPath.py that loads a 'path' file, parses it, and plots it. The result for f.path is shown in Fig. 9.16.

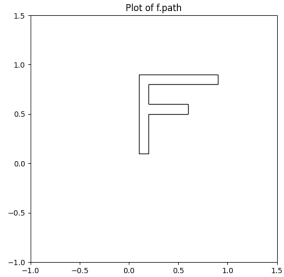


Figure 9.16. Plotting f.path

The relevant snippets of code in ShowPath.py are:

```
pathStr = open(fnm+'.path', 'r', encoding='utf-8').read()
path = parse_path(pathStr)
path = path.transformed(  # flip and move to first quadrant
                Affine2D().scale(1,-1).translate(0,1))

_, ax=plt.subplots(figsize=(6, 6))
ax.add_patch(PathPatch(path, fill=False))
```

The transformations are applied to the Matplotlib path and then PathPatch() convert it into a shape (a Matplotlib patch).

Paths generated using SVG tools utilize the same conventions for their coordinates as PNG images: the y-axis origin is at the top-left, and y-values increase downwards. This is remedied using similar transformations to those used on the 'cat' image. Scaling flips the path in the x-axis so its origin is at the bottom-left, and translation moves the path up from below the x-axis.

9.12 Henderson's Functional Geometry

Peter Henderson's functional geometry is an elegantly simple algebra for transforming and composing pictures based around operations such as rot(), flip(), above(), and beside(). Nevertheless, this small set of primitives are sufficiently powerful to draw the Escher woodcut, Square Limit (`https://www.wikiart.org/en/m-c-escher/square-limit`).

His ideas are presented in a classic paper (from 1982, revised in 2002) that beautifully illustrates the power of functional programming for data abstraction (`https://eprints.soton.ac.uk/257577/1/funcgeo2.pdf`. Another source is section 2.2.4 of Abelson and Sussman's magnificent *Structure and Interpretation of Computer Programs* [**ASS96**]. The second edition is freely available at `https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/`.

Massimo Santini has implemented a version of Henderson's geometry in Python, using Matplotlib's patches, paths, and Affine2D (`https://mapio.github.io/programming-with-escher/`). He defines a Tile class which uses code similar to our ShowPath.py to load and convert a SVG-style path into a Matplotlib path, which is displayed as a PathPatch shape.

Henderson's transformations, such as rot() and flip(), are encoded using Affine2D, while compositional operations, such as above() and beside(), rely on Matplotlib's Path.make_compound_path(). As the name suggests, this takes two separate paths and packages them as a single compound path.

We've prepared a (slightly simplified) version of Santini's code in Tile.py. When run standalone, it generates a 2×4 grid of plots of various manipulations of the f.path shape (see Fig. 9.17).

Fig. 9.17 highlights that a tile always fits into a unit square even after several tiles have been combined.

squareLimit.py follows the steps outlined by Henderson to define Escher's 'Square Limit' woodcut. It requires the definition of two new functions, side() and corner(), which draw the sides and corners of the picture. It also employs nonet() which creates a 3×3 tile grid. With these, drawing Escher's image is a (long) one-liner:

```
def squarelimit(n):
  return nonet(   # 3 x 3 grid made of
```

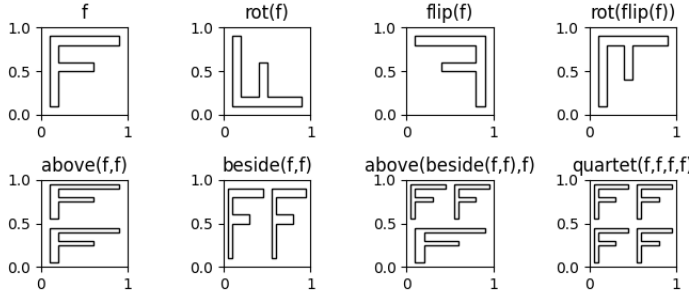


Figure 9.17. Tile Operations Upon "f"

```
    corner(n),              side(n),  rot(rot(rot(corner(n)))),
    rot(side(n)),               f4,    rot(rot(rot(side(n)))),
    rot(corner(n)), rot(rot(side(n))), rot(rot(corner(n)))
)
```

The resulting image is Fig. 9.18.

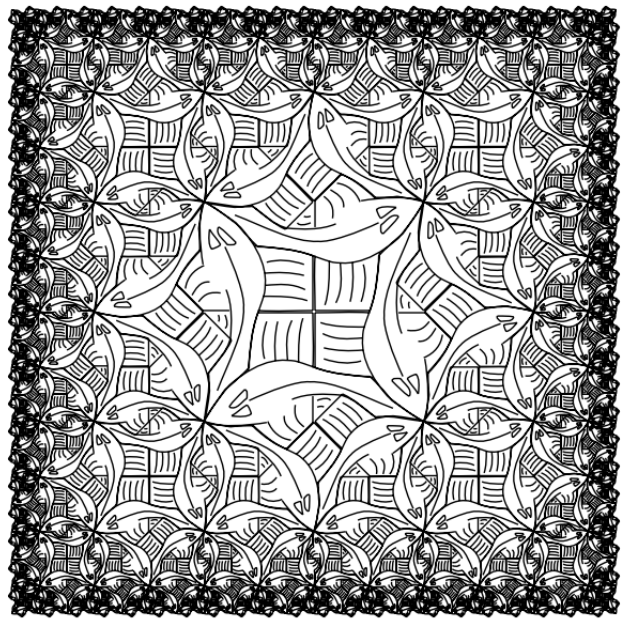


Figure 9.18. Tile Version of Escher's Square Limit

You can utilize Matplotlib's window controls to zoom in on a corner or side to see the tiling in more detail.

9.13 Hierarchy of Transformations

The mix of transformations supported by HC2Mat.py can be organized into the hierarchy shown in Fig. 9.19.

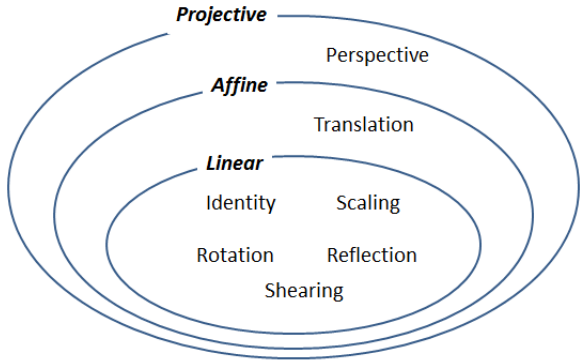


Figure 9.19. Transformation Hierarchy

The figure is a bit misleading since HC2Mat.py doesn't include any projective transformations, such as perspectives or warping. Even so, HC2Mat.py's use of homogeneous coordinates makes it simple to add such operations.

The following sections briefly describe the hierarchy, which helps to explain some of the choices when implementing and using HC2Mat.py transformations.

Linear Transformations. A linear transformation L is a mapping $V \rightarrow W$ between two vector spaces that preserves the operations of vector addition and scalar multiplication. In other words, for any two vectors $u, v \in V$ and any scalar c the following conditions are satisfied:

$$\begin{aligned} L(u+v) &= L(u)+L(v) \\ L(cu) &= cL(u) \end{aligned}$$

From these it follows that $L(0) = 0$, which means that linear transformations leave the vector space's origin unaffected.

A set B is a *basis* for a vector space V if its elements are independent, and can be linearly combined to represent every vector in V. For example, when V is the 2D space $\mathbb{R}^2$, it's natural to define B as the unit vectors along the axes: $\{[1,0]^T, [0,1]^T\}$. A suitable combination of these, with scaling, allows any vector in $\mathbb{R}^2$ to be defined. Similarly, a natural basis set for $\mathbb{R}^3$ are the unit vectors for its axes: $\{[1,0,0]^T, [0,1,0]^T, [0,0,1]^T\}$.

The basis for a space can be employed to define how any linear function affects the space. For instance, for $L : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ we can define its behavior by

focusing on how the basis is changed:

$$\begin{aligned} L([1,0,0]^T) &= [2,-1,4]^T \\ L([0,1,0]^T) &= [1,5,-2]^T \\ L([0,0,1]^T) &= [0,3,1]^T \end{aligned}$$

Linearity makes it trivial to determine how L affects other vectors given this information. For example,

$$[2,-3,-2]^T = 2[1,0,0]^T + 3[0,1,0]^T - 2[0,0,1]^T$$

and so

$$\begin{aligned} L([2,3,-2]^T) &= 2L([1,0,0]^T) + 3L([0,1,0]^T) - 2L([0,0,1]^T) \\ &= 2[2,-1,4]^T + 3[1,5,-2]^T - 2[0,3,1]^T \\ &= [7,7,0]^T \end{aligned}$$

Another advantage of linearity is that any transformation can be encoded as a *standard* matrix that stores the transformations of its basis as column vectors. So,

$$L(\mathbf{v}) = A\mathbf{v} = \begin{bmatrix} 2 & 1 & 0 \\ -1 & 5 & 3 \\ 4 & -2 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Now $L([2,3,-2]^T)$ can be expressed using the A matrix as:

$$L(\mathbf{v}) = A\mathbf{v} = \begin{bmatrix} 2 & 1 & 0 \\ -1 & 5 & 3 \\ 4 & -2 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -2 \end{bmatrix} = \begin{bmatrix} 7 \\ 7 \\ 0 \end{bmatrix}$$

Determining the standard matrix for a given transformation is simply a matter of applying the transformation to each of the basis vectors, and collecting the results. For example, if $L([x,y,z]^T) = [2x+y, 0, x+z]^T$, then the standard matrix is obtained by applying L to $[1,0,0]^T$, $[0,1,0]^T$, and $[0,0,1]^T$. For the x-axis:

$$L([1,0,0]^T) = [2,0,1]^T$$

and the complete matrix for L is:

$$A = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

More generally, linearity makes it possible to manipulate an infinite n-dimensional space in terms of a basis with a finite number of elements (n). The matrix correspondence also lets us tap into a vast assortment of tools such as Gauss-Jordan elimination and the simplex algorithm.

Linear geometric operations include rotation, scaling, reflection, and shearing, but not translation which takes us to the next level in the hierarchy.

Affine Transformations. Although affine transformations, such as translation, don't leave the origin untouched, they do preserve other geometry properties, such as parallelism.

It can be shown that any affine transformation $A : U \to V$ can be written as $A(x) = L(x) + v_0$, where v_0 is some vector from V, and $L : U \to V$ is a linear transformation. In other words, any affine function is the composition of a linear function and a translation.

A very useful feature of linearity is that the composition of transformations (e.g. a scaling followed by a rotation) can be encoded using only matrix multiplication, but this fails if we define an affine operation as $A(x) = L(x) + v_0$. As we've seen above, it's much more convenient to switch to matrices that utilize homogeneous coordinates, and restrict $W = 1$:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

However, care must still be taken when combining translation with linear operations such as rotation and scaling since they assume that their centers of rotation/scaling are the origin. This explains the frequent appearance of translations bracketed around linear operations – to move a shape's center to and from the origin.

Projective Transformations. A projective transformation doesn't necessarily preserve parallelism, but collinearity and incidence are maintained (since lines are mapped to lines). Fig. 9.20 illustrates the effect.

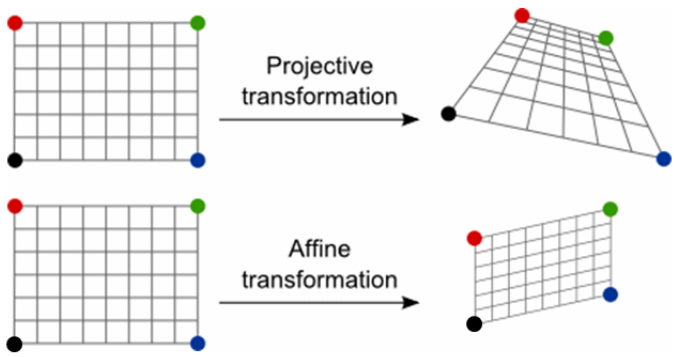


Figure 9.20. Projective vs. Affine

Homogeneous coordinates support projective transformations by allowing the last row of the matrix to contain any values:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

9.14 3D Transformations as Matrices

Just as 2D transformations can be represented by 3×3 matrices using homogeneous coordinates, so 3D transformations can utilize 4×4 matrices. Instead of representing a point as (x, y, z), it becomes (x, y, z, W), with two quadruples denoting the same point if one is a nonzero multiple of the other. As in 2D, the standard representation of (x, y, z, W) with $W \neq 0$ is $(x/W, y/W, z/W, 1)$, and points whose W coordinate is zero are considered to be at infinity.

Translation in 3D:

$$T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

That is, $T(d_x, d_y, d_z) \cdot [x, y, z, 1]^T = [x + d_x, y + d_y, z + d_z, 1]^T$.

Scaling:

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

So $S(s_x, s_y, s_z) \cdot [x, y, z, 1]^T = [s_x \cdot x, s_y \cdot y, s_z \cdot z, 1]^T$.

Our earlier 2D rotations can be treated as 3D rotations about the z-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This is easily verified: a 90° rotation of the x-axis unit vector, $[1, 0, 0, 1]^T$, should produce the y-axis unit vector, $[0, 1, 0, 1]^T$. Evaluating

$$\begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

gives the predicted result.

X-axis rotation is

$$R_x(0) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y-axis rotation:

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Any number of rotation, scaling, and translation matrices can be multiplied together, but the result always has the form

$$M = \begin{bmatrix} r11 & r12 & r13 & tx \\ r21 & r22 & r23 & ty \\ r31 & r32 & r33 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The 3×3 upper-left submatrix contains the aggregate rotation and scaling, while the right-hand column holds the translation.

There are three 3D shear matrices, corresponding to the planes they work upon. The (x, y) shear along the z-axis is

$$SH_{xy}(sh_x, sh_y) = \begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 0 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Applying SH_{xy} to the point $[x, y, z, 1]^T$ produces $[x + sh_x \cdot z, y + sh_y \cdot z, z, 1]^T$. Shears along the x- and y-axes have a similar form.

Just as we packaged the functions for 2D transformations in HC2Mat.py, the 3D transformations are located in HC3Mat.py. For example, a rotation around the z-axis is defined by:

```
def rotZMat(angle):
  theta = math.radians(angle)
  return [ [math.cos(theta), -math.sin(theta), 0, 0],
           [math.sin(theta),  math.cos(theta), 0, 0],
           [             0,                0,  1, 0],
           [             0,                0,  0, 1]  ]
```

There are also support functions for 4×4 matrix multiplication, printing, and applying a matrix to a coordinate. For instance, multiplication:

```
def multMats(matrixA, matrixB):
  result = [[0 for _ in range(4)] for _ in range(4)]
```

```
  for i in range(4):
    for j in range(4):
      for k in range(4):
        result[i][j] += matrixA[i][k] * matrixB[k][j]
  return result
```

9.15 Composition of 3D Transformations

A desired transformation is obtained by composing primitive transformations such as T, R_x, R_y, and R_z. However, the linear transformations (rotation, scaling, shearing, and reflection) assume that the origin is at $(0, 0)$, and so, as in the 2D case, the composition of these operations will be frequently bracketed by translations. The first moves the shape's center or origin at P to the 3D space's origin. The final translation moves the transformed shape's center or origin back to P.

letter3D.py illustrates 3D transformation composition. An "R" shape (actually a box and a triangle) initially rests on its back on the XY plane, with the bottom, left corner at the origin (see Fig. 9.21).

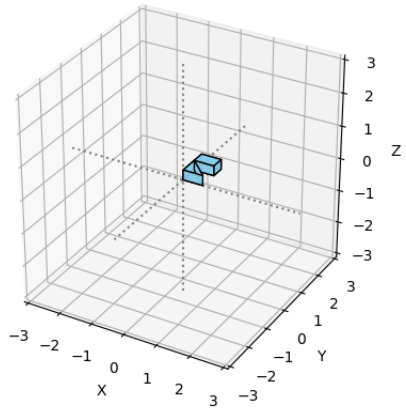


Figure 9.21. The Initial "R" Shape

The shape is defined using a list of vertices *and* a list of face indices which indicate how the coordinates are grouped to form the shape's faces:

```
# vertices for R shape
verts = [
  # Bottom (z = 0)
  (0.0, 0.0, 0.0),  # 0
  (0.0, 1.0, 0.0),  # 1
  (0.6, 1.0, 0.0),  # 2
```

```
  (0.6, 0.6, 0.0),  # 3
  (0.0, 0.6, 0.0),  # 4
  (0.6, 0.0, 0.0),  # 5

  # Top (z = 0.3)
  (0.0, 0.0, 0.3),  # 6
  (0.0, 1.0, 0.3),  # 7
  (0.6, 1.0, 0.3),  # 8
  (0.6, 0.6, 0.3),  # 9
  (0.0, 0.6, 0.3),  # 10
  (0.6, 0.0, 0.3),  # 11
]

# "R" faces using vertex indices
faceIndices = [
  [0, 1, 2, 3, 4, 5],      # bottom face
  [6, 7, 8, 9, 10, 11],    # top face
  [0, 1, 7, 6],            # side 1
  [1, 2, 8, 7],            # side 2
  [2, 3, 9, 8],            # side 3
  [3, 4, 10, 9],           # side 4
  [4, 5, 11, 10],          # side 5
  [5, 0, 6, 11],           # side 6
]
```

Fig. 9.22 shows two views of this data. The left hand image indicates the position of the first six vertices, on the XY plane ($z = 0$). The right-hand image shows all the vertices numbered with the indices used by the faces.

Rendering utilizes Matplotlib's Poly3DCollection class (`https://matplotlib.org/stable/api/_as_gen/mpl_toolkits.mplot3d.art3d.Poly3DCollection.html`) which requires a list of coordinates grouped into sublists for each face. The relevant code in letter3D.py is:

```
# no transformation applied
nVerts = [applyMat(identMat(), vert) for vert in verts]

# build "R" poly, and show
faces = [[nVerts[i] for i in face]
                    for face in faceIndices]
poly = Poly3DCollection(faces, facecolors='skyblue',
           linewidths=1, edgecolors='black', alpha=0.95)
ax.add_collection3d(poly)
```

The first line applies the identity matrix to every coordinate, leaving the shape unchanged.

Moving the "R": Our transformation will do four things:

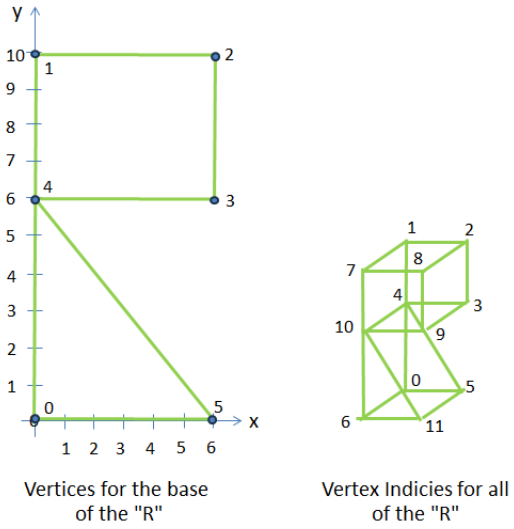


Figure 9.22. Defining the "R" Shape

(1) an x-axis rotation by 90° (to make the "R" stand on the XY plane);

(2) a z-axis rotation by 30° (to turn the "R" counterclockwise over the XY plane;

(3) scaling by 4 (i.e. "R" will now be 4 units high);

(4) translate the shape's origin to $(-1, 1, -3)$, moving the rest of the shape in the process.

Thankfully, the shape's origin (its vertex 0) is already at $(0, 0, 0)$, so there's no need to bracket these operations in translations. The resulting image is shown in Fig. 9.23.

You can confirm the transformation's result by rotating the viewpoint of the Matplotlib window. For instance, vertex 0 is at $(-1, 1, -3)$ and the shape is four units tall.

The transformation code:

```
# two rotations, scale, translate
mats = [transMat(-1,1,-3), scaleMat(4,4,4), rotZMat(30), rotXMat(90)]
mat = multMatsList(mats)
nVerts = [applyMat(mat, vert) for vert in verts]
```

As in the 2D case, the transformations are performed in right-to-left order.

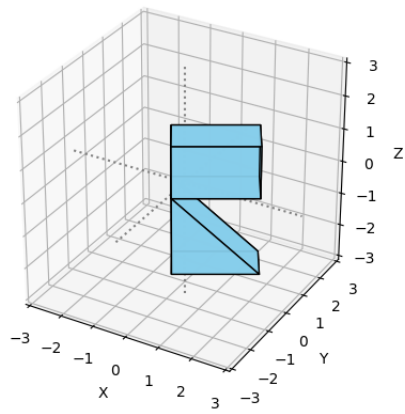


Figure 9.23. The Final Position of the "R" Shape

9.16 Recursive Cube Drawing

Fig. 9.24 shows how a single cube can be redrawn multiple times using recursive translation and scaling.

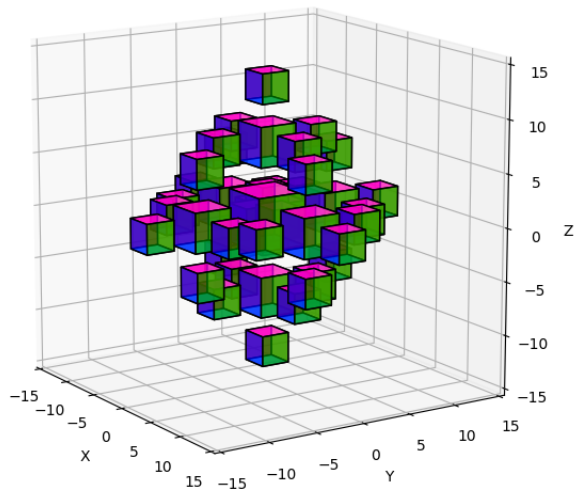


Figure 9.24. A Grid of Shrinking Cubes

cubes.py begins by defining the vertices and face indices for a unit cube with its front lower corner positioned at the origin, and its base resting on the XY plane. Its vertex indices are shown in Fig. 9.25.

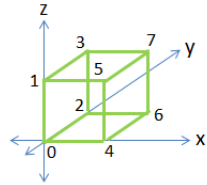


Figure 9.25. A Cube's Vertex Indices

The six faces are defined in cubes.py:

```
# unit cube vertices
verts = [
  [0, 0, 0],  # 0
  [0, 0, 1],  # 1
  [0, 1, 0],  # 2
  [0, 1, 1],  # 3
  [1, 0, 0],  # 4
  [1, 0, 1],  # 5
  [1, 1, 0],  # 6
  [1, 1, 1]   # 7
]

# cube faces using vertex indices
faceIndices = [
  [0, 1, 3, 2],  # Left
  [4, 5, 7, 6],  # Right
  [0, 1, 5, 4],  # Front
  [2, 3, 7, 6],  # Back
  [0, 2, 6, 4],  # Bottom
  [1, 3, 7, 5],  # Top
]

faceColors = ['red', 'green', 'blue', 'yellow',
              'cyan', 'magenta']
```

The ordering of the `faceColors` list corresponds to the ordering of the faces in `faceIndicies`. This will mean, for example, that the top, front, and right sides of the cube will be magenta, blue, and green at run time, as in Fig. 9.24.

Initially the cube is centered at the origin, and enlarged:

```
# center and resize the cube
side = verts[1][2] - verts[0][2]   # x-axis length
m = multMatsList([ scaleMat(SIZE,SIZE,SIZE),
                   transMat(-side/2, -side/2, -side/2)
                 ])
```

```
verts = [applyMat(m, v) for v in verts]

drawCubes(ax, verts, 3)  # use 2, 3 or 4
```

The resursive drawCubes() generates a grid of smaller cubes; its numerical argument is the depth of the recursion.

There are six recursive calls inside drawCubes(), one for each x-, y-, and z-direction. They're encoded using:

```
dirs = [(1,0,0), (-1,0,0), (0,1,0), (0,-1,0), (0,0,1), (0,0,-1)]

def drawCubes(ax, verts, depth):
  # draw a cube and then recurse in every direction
  if depth > 0:
    drawCube(ax, verts)
    for dir in dirs:
      drawCubes(ax, transformCube(verts, dir), depth-1)
```

drawCube() calls Matplotlib's Poly3DCollection to draw the cube in the same way as the letter-drawing in letter3D.py.

transformCube() scales and translates a copy of the cube relative to its center. This requires that the transformation be bracketed by a translation of the current cube's center at P (for example) to the origin, and back to P after the transformation:

```
def transformCube(verts, dir):
  P = calcCenter(verts)  # the cube's midpoint
  side = verts[4][0] - verts[0][0]   # x-axis length
  step = side*1.4
  m = multMatsList([
         transMat(P[0], P[1], P[2]),
         transMat(step*dir[0], step*dir[1], step*dir[2]),
               # translate by step*dir
         scaleMat(0.75, 0.75, 0.75),
         transMat(-P[0], -P[1], -P[2])
      ])
  return [applyMat(m, vert) for vert in verts]

def calcCenter(verts):
  # center coordinate of cube
  xc = (verts[0][0] + verts[4][0])/2
  yc = (verts[0][1] + verts[2][1])/2
  zc = (verts[0][2] + verts[1][2])/2
  return (xc, yc, zc)
```

The new cube is a copy of the current one scaled by 0.75 and translated by 1.4× the direction vector. Only the cube's vertices have to be computed since its face indices never change.

Exercises

(1) Verify each of the following:

(a) A 2D rotation about (r, s) through angle θ is represented by the matrix

$$\begin{bmatrix} \cos\theta & -\sin\theta & r(1-\cos\theta)+s(\sin\theta) \\ \sin\theta & \cos\theta & s(1-\cos\theta)-r(\sin\theta) \\ 0 & 0 & 1 \end{bmatrix}$$

(b) A 2D reflection about the line $y = mx + b$ is represented by the matrix

$$\frac{1}{1+m^2}\begin{bmatrix} 1-m^2 & 2m & -2mb \\ 2m & m^2-1 & 2b \\ 0 & 0 & 1+m^2 \end{bmatrix}$$

(c) A 2D scaling about (r, s) with scale factors c in the x-direction and d in the y-direction is represented by the matrix

$$\begin{bmatrix} c & 0 & r(1-c) \\ 0 & d & s(1-d) \\ 0 & 0 & 1 \end{bmatrix}$$

(2) Modify the code in `transformPoly.py` (or `affinePoly.py`) to carry out the following transformations of the "Knee":

(a) Translate by $(3, 1)$, and then rotate 45° about the origin.
(b) Translate by $(-2, 3)$, and then scale the x-coordinates by 0.8 and the y-coordinates by 1.2.
(c) Rotate through 60° about the point $(6, 8)$.
(d) Reflect through the x-axis, and then rotate 30° about the origin.
(e) Rotate by 30°, and then reflect the shape through the x-axis.

(3) This exercise investigates whether certain transformations are commutative.

(a) Let L_1 be a scaling about the point (r, s) with equal scale factors in the x- and y-directions, and let L_2 be a rotation about the point (r, s) through angle θ. Show that L_1 and L_2 commute. (That is, show $L_1 \cdot L_2 = L_2 \cdot L_1$.)
(b) Give a counterexample to show that, in general, a reflection and a rotation do not commute.
(c) Give a counterexample to show that, in general, a scaling and a reflection do not commute.

(4) A 3×200 matrix D contains the 2D homogeneous coordinates of 200 points. Calculate the number of multiplications required to transform these points using two arbitrary 3×3 matrices A and B. Consider the two possibilities $A(BD)$ and $(AB)D$. Discuss the implications of your results for graphics calculations.

(5) A rotation is sometimes implemented as the product of two shear-and-scale transformations, which can speed up the calculations. The first transformation A_1, shears vertically and then compresses each column; the second transformation A_2 shears horizontally and then stretches each row. Let

$$A_1 = \begin{bmatrix} 1 & 0 & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad A_2 = \begin{bmatrix} \sec\theta & -\tan\theta & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Show that the composition of the two transformations is a rotation in $\mathbb{R}^2$.

(6) A rotation in $\mathbb{R}^2$ usually requires four multiplications. Compute the product below, and show that the matrix for a rotation can be factored into three shears (each of which requires only one multiplication).

$$\begin{bmatrix} 1 & -\tan\theta/2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ \sin\theta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -\tan\theta/2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(7) Write a recursive function called drawTriangles() which draws an equilateral triangle and then recursively calls itself three times to draw smaller triangles offset from its three sides. For example, if the function recurses three times, the image in Fig. 9.26 is drawn.

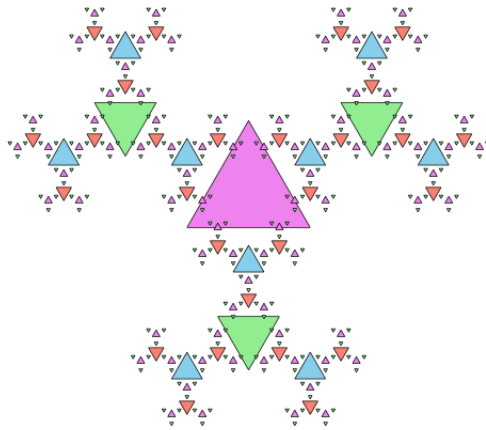


Figure 9.26. Recursive Triangles

The initial triangle could be defined like so:

```
base = math.sqrt(3)/6
coords = [ (-0.5,-base), (0.5,-base), (0, 2*base)]  # unit lengths
```

which would place its center at the origin.

The transformation applied to the triangle is composed from a scaling, translation, and a rotation, each performed three times as in Fig. 9.27. The diagram suggests that the translation should be twice the distance to the base, and the scaling factor is a 1/2.

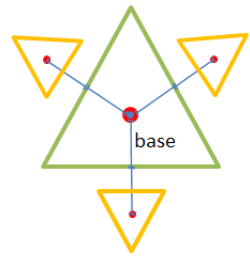


Figure 9.27. Generating Three Smaller Triangles

(8) Generate the tiling pattern shown in Fig. 9.28.

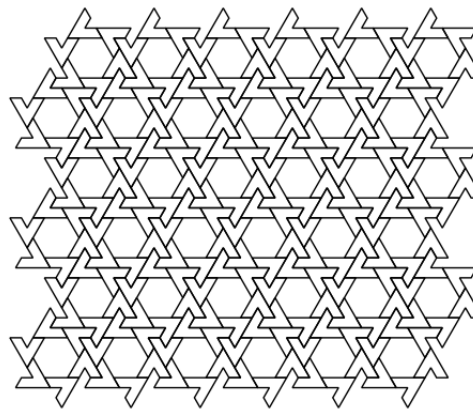


Figure 9.28. Tiling using Hexagons and ”V”s

One way to implement this is to start with a hexagon and "V" shape, as in Fig. 9.29.

A function could rotate and translate copies of "V" around the hexagon. The tiling then becomes a matter of translating copies of the hexagon before the "V"'s are drawn around them.

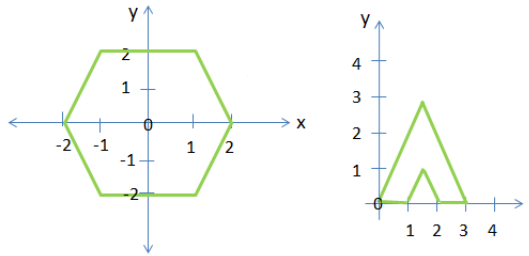


Figure 9.29. A Hexagon and a "V"

The easiest way of determining suitable translations for the hexagons is by sketching out Fig. 9.28 on some graph paper.

(9) Baravelle spirals (Sec. 9.7) can be created from any regular polygon. Modify the code in baravelle.py to start from a different shape than a square. You can check your results against the equilateral triangle, pentagon, and hexagon utilized by the National Curve Bank (`https://old.nationalcurvebank.org/baravelle/baravelle.htm`).

(10) Truchet tiles are decorated with patterns that are *not* rotationally symmetric, causing them to generate intriguing patterns when tiled over the plane with random 90° rotations (`https://en.wikipedia.org/wiki/Truchet_tiles`). Two such tiles are available on online in tri.png and arc.py, shown in Fig. 9.30.



Figure 9.30. Two Truchet Tiles

Write code using Matplotlib's Affine2D class to generate a 12 × 12 grid of randomly rotated Truchet tiles using one of the PNG images as the basic tile. For example, the code might generate the patterns shown in Fig. 9.31.

(11) The transformational approach to generating the Hilbert curve family in section 9.8 can of course be applied to other fractal curves. Modify the code in hilbert.py to generate a Koch curve, Dragon curve, or another of your choosing (see Fig. 9.32).

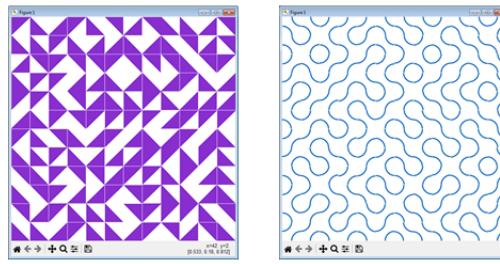


Figure 9.31. Truchet Patterns using tri.png and arc.png

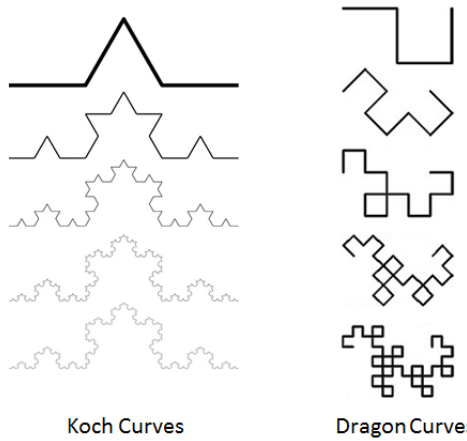


Figure 9.32. Two Fractal Curve Families

(12) The *Functional Geometry* paper by Henderson (sec. 9.12) is well worth a read. It can be downloaded from `https://eprints.soton.ac.uk/257577/1/funcgeo2.pdf`.

You can experiment with his approach using several path and SVG files available online: `edge.path`, `triangle.path`, `house.svg`, and `wave.svg`.

Remember that if you employ SVG files containing multiple 'd' paths (as in the two SVG examples) then you'll need to modify `Tile.py` to manipulate Matplotlib PathCollection objects rather than paths and PathPatch. Have a look at `ShowSVG.py` for coding details.

10

Computational Geometry

Why Study Computational Geometry?

- **It unifies discrete and continuous mathematics**. Computational geometry sits at the intersection of analysis, topology, and algorithm design.
- **It offers fast and reliable geometric computation**. Many tasks in graphics, robotics, CAD, GIS, and simulation involve computational geometry operations such as intersection tests, convex hulls, triangulations, and nearest-neighbour queries.
- **It provides algorithms with guaranteed performance**. Within computational geometry, combinatorics, graph theory, topology, convexity, and numerical analysis allow the design of algorithms with known time and space bounds. This is crucial for applications that must run in real time, or have to scale to massive datasets.

10.1 Introduction

Computational geometry became a hot topic in computer science in the 1990s, although mathematicians such as Dirichet and Voronoi had been developing similar ideas in the 1850s and 1900s respectively. The modern aim is to develop data structures and algorithms for geometric objects (e.g. for points, lines, line segments, triangles, polygons) that are both exact and fast. Typical application areas include graphics, robotics (e.g. path finding), Geographical Information Systems, CAD/CAM (e.g. for the construction and simplification of objects using set operations), database search, and chip layout.

Devising exact algorithms has proved troublesome because of the computer's inherent limited numerical precision when dealing with floats, and also because many 'intuitive' algorithms become much more complex when edge cases are considered. For instance, should a line segment that is just touching another be considered to be intersecting it, and how should 'touching' be defined?

A common solution to the floats problem is to pepper code with approximation tests that determine if two values are sufficiently 'close'. This works most of the time but isn't mathematically sound. For instance, point a may be close to point b, and b close to point c, but a may not necessarily be close to c, and so transitivity fails.

As a consequence, computational geometry should be treated in the same way as cryptography – develop your own code for the purpose of learning the concepts, but turn to a trusted library when you need to write real applications. We suggest CGAL, the Computational Geometry Library, which is robust, fast (it utilizes C++ internally), and offers a very wide range of algorithms. A downside is that the Python port (`https://github.com/CGAL/cgal-swig-bindings/wiki`) is only partial and its installation is a little tricky. An alternative is Shapely (`https://shapely.readthedocs.io/en/stable/`) which has less features than CGAL.

In this chapter, we'll look at three problems: segment intersection, finding a convex hull, and searching for the closest pair of points. Of course, computational geometry is a huge subject, so for more details please consult one of the excellent standard texts, such as Berg [**dBOCvKO08**] or O'Rourke [**O'R98**].

10.2 Geometry Data Structures

Our Python classes for representing points, line segments, and vectors are stored in `point.py`, `segment.py`, and `vector.py` respectively, and a range of tests are in `GeomTests.py`.

The Point class is based around the manipulation of an x and y coordinate. For instance, the equality function is:

```
def __eq__(self, p):
  if not Point.isPoint(p):
    return False
  return abs(self.x - p.x) < EPS and \
         abs(self.y - p.y) < EPS
```

Currently, the epsilon value EPS is set to 1e-8.

The class contains functions for calculating the distance between points, the mid-point between points, and the angles between them. There's also a draw() method which uses matplotlib.

10.3 The Dot and Cross Products

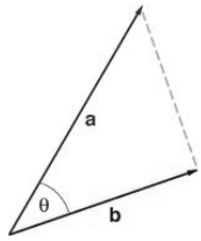
The dot and cross products are at the core of many Computational Geometry algorithms. They're implemented in our Vector class:

```
def dot(u, v):   # u and v are vectors
  return ((u.x * v.x) + (u.y * v.y))

def cross(u, v):
  return ((u.x * v.y) - (u.y * v.x))
```

Both operations have trigonometric and geometric interpretations.

10.3.1 The dot product. The dot product can be defined as the multiple of the magnitudes of the two vectors and the cosine of the angle between them, as illustrated in Fig. 10.1.



$$\mathbf{a} \cdot \mathbf{b} = |a||b| \cos\theta$$

Figure 10.1. The dot product as trigonometry

In terms of geometry, the operation can be viewed as the length obtained when a vector **a** is projected onto the unit vector version of **b**, $\frac{\mathbf{b}}{\|b\|}$ (see Fig. 10.2).

The sign of the projected length can be employed as a measure of the relative directions of the two vectors. A positive value means that **a** and **b** are roughly pointing in the same direction, a negative value means they are pointing in opposite directions, and a 0 signifies that they are perpendicular (i.e. **a** produces no 'shadow' when projected onto **b**).

It's possible to use the dot product to define a perpendicular, as in Fig. 10.3.

The vector from points o to p is the unit vector $\hat{\mathbf{b}}$ in that direction multiplied by the projected length of **a**:

$$\begin{aligned} \overrightarrow{op} &= \frac{\mathbf{a}\cdot\mathbf{b}}{\|b\|} * \frac{\mathbf{b}}{\|b\|} \\ &= \frac{\mathbf{a}\cdot\mathbf{b}}{\|b\|^2}\mathbf{b} \end{aligned}$$

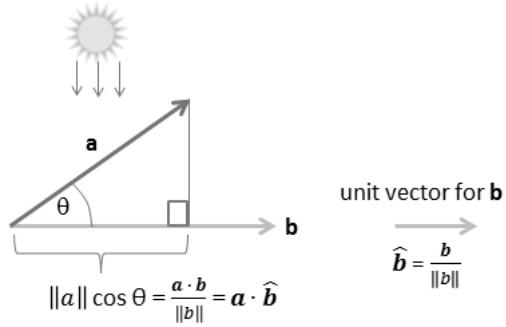


Figure 10.2. The dot product as geometry

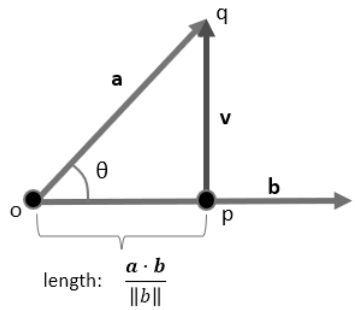


Figure 10.3. The perpendicular vector **v**

The perpendicular **v** is obtained by vector arithmetic:

$$\mathbf{v} \quad = \quad \mathbf{a} - \frac{\mathbf{a} \cdot \mathbf{b}}{\|b\|^2} \mathbf{b}$$

10.3.2 The cross product. The cross product is usually defined using a determinant. For example, the cross product of two vectors $\mathbf{p} = (x_1, y_1)$ and $\mathbf{q} = (x_2, y_2)$ is:

$$\begin{aligned} \mathbf{p} \times \mathbf{q} &= \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} \\ &= x_1 y_2 - x_2 y_1 \end{aligned} \tag{10.1}$$

It's helpful to think of the operation as a 3D concept which, given two vectors, defines a perpendicular according to the 'right-hand rule'. It considers the vectors in a counter-clockwise order, as shown in Fig. 10.4(a).

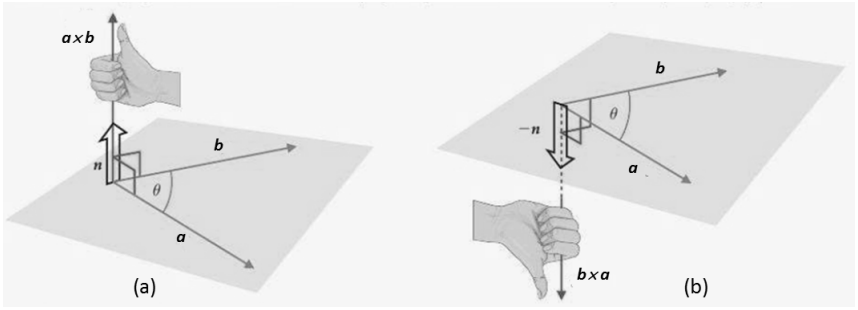


Figure 10.4. The right-hand rule

The cross product of $\mathbf{a} = (a_1, a_2, a_3)$ and $\mathbf{b} = (b_1, b_2, b_3)$ can be written as:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

where **i**, **j**, and **k** are unit vectors along the x-, y-, and z- axes. This expands to:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \end{vmatrix} \mathbf{i} - \begin{vmatrix} a_1 & a_3 \\ b_1 & b_3 \end{vmatrix} \mathbf{j} + \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix} \mathbf{k}$$

A consequence of the right-hand rule is that if the vectors are considered in clockwise order, as in Fig. 10.4(b), then the perpendicular will point in the opposite direction. This offers a way to determine the relative ordering of vectors based on the cross product sign.

The operation also has a trigonometric interpretation:

$$\mathbf{a} \times \mathbf{b} = \|a\| \, \|b\| \sin\theta \, \mathbf{n}$$

The magnitude of the perpendicular vector **n** is equal to the area of the parallelogram that the vectors span, as depicted in Fig. 10.5.

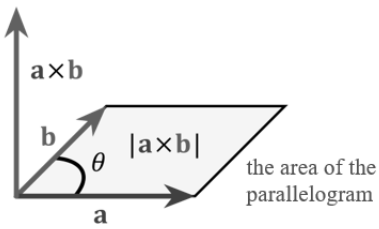


Figure 10.5. The area of a cross product

This area is the base of the parallelogram ($\|\mathbf{a}\|$) multiplied by its height ($\|\mathbf{b}\| \sin\theta$), which also provides a way to obtain the angle θ between the vectors.

The following example comes from GeomTests.py:

```
orig = Point(0, 0)
pt1 = Point(3, -5)
pt2 = Point(2, 6)
print("pt1, pt2:", pt1, pt2)
print("Dot product:", Vector.dot(pt1, pt2))
print("Cross product:", Vector.cross(pt1, pt2))
print(f"Angle: {math.degrees( Vector.angle(pt1, orig, pt2)):.2f} degrees")
```

The output is:

```
pt1, pt2: (3.0, -5.0) (2.0, 6.0)
Dot product: -24.0
Cross product: 28.0
Angle: 130.60 degrees
```

10.4 The ccw() function

Many computational geometry algorithms rely on the ccw() function which determines if the two vectors supplied as three points are in counter-clockwise order. The function returns 1 if they are, -1 if they are in clockwise order, or 0 if they are collinear. Fig. 10.6 shows the three possibilities for ccw(p_0, p_1, p_2).

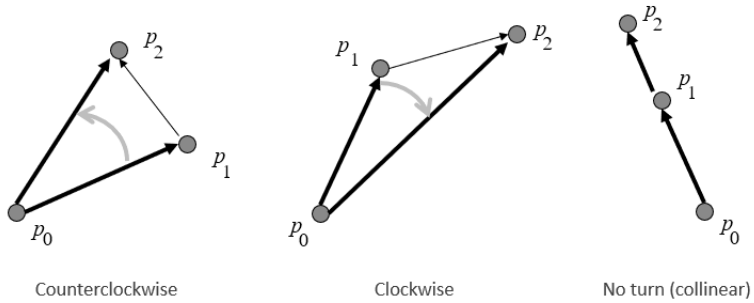


Figure 10.6. Vector ordering using ccw()

ccw() can be defined in a few lines by converting its point arguments to vectors and calling the cross product (see Listing 10.1; vector.py).

```
@staticmethod
def ccw(p, q, r):
    '''
```

```
    returns 0, 1, or -1
     0 if p-->r and p-->q are collinear
     1 if p--> r is left (counter-clockwise) of p-->q
    -1 if p--> r is right (clockwise) of p-->q
    '''
    pq = Vector.toVec(p, q)
    pr = Vector.toVec(p, r)
    cross = Vector.cross(pq, pr)
    if abs(cross) < EPS:
      return 0
    elif cross > 0:
      return 1
    else:
      return -1
```

Listing 10.1. ccw() in the vector class

Float errors can arise in ccw() because of its reliance on Vector.cross() which uses subtraction (see the code for cross() at the start of Sec. 10.3). For example, if one of the points is nearly collinear to the line segment defined by the other two, then rounding errors associated with the subtraction can overwhelm the result. Such a situation is shown in Fig. 10.7, where ccw(p,s,r) *might* incorrectly determine that $\overrightarrow{pr}$ is a clockwise rotation from $\overrightarrow{ps}$.

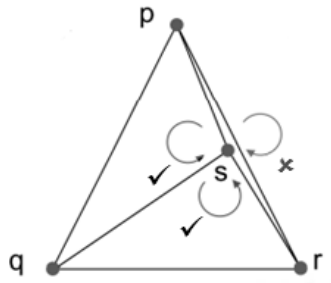


Figure 10.7. Failure of ccw()

This could have other consequences. For instance, it might not be possible to determine that s is inside the $\triangle pqr$.

10.5 Segment Intersection

A segment is defined in terms of two endpoints, and its class includes functions related to segment intersection. They can be implemented by using ccw() to compare the directions of the four points making up the two segments. The situation is illustrated in 10.8.

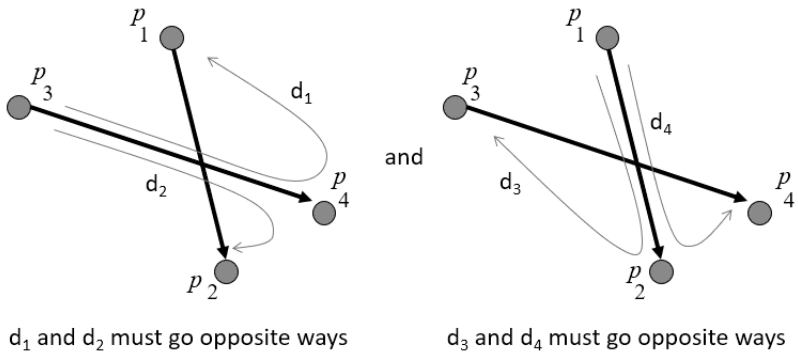


Figure 10.8. Segment intersection using ccw()

The code is in Listing 10.2 (segment.py).

```
def intersects(self, seg):
  # does seg intersect with this one?

  # Find the 4 orientations required
  d1 = Vector.ccw(self.p1, self.p2, seg.p1)
  d2 = Vector.ccw(self.p1, self.p2, seg.p2)
  d3 = Vector.ccw(seg.p1, seg.p2, self.p1)
  d4 = Vector.ccw(seg.p1, seg.p2, self.p2)

  # General case
  if ((d1 != d2) and (d3 != d4)):
      return True

  # Special Cases
  # collinear if seg.p1 lies on this segment
  if ((d1 == 0) and self.onSegment(seg.p1)):
      return True

  # collinear if seg.p2 lies on this segment
  if ((d2 == 0) and self.onSegment(seg.p2)):
      return True

  # collinear if this p1 lies on other segment
  if ((d3 == 0) and seg.onSegment(self.p1)):
      return True

  # collinear if this p2 lies on other segment
  if ((d4 == 0) and seg.onSegment(self.p2)):
      return True

  # If none of the cases
```

```
    return False
```

Listing 10.2. intersect() in the segment class

As is typical with computational geometry, most of the function is taken up with checking for boundary cases, when one of the points is located on the other segment.

The following shows the use of intersects() and also the drawing of the segments using matplotlib in Fig. 10.9.

```
seg1 = Segment(Point(1, 1), Point(10, 7))
seg2 = Segment(Point(1, 2), Point(10, 2))
print("seg1, seg2:", seg1, seg2)
print("Intersects?", seg1.intersects(seg2))
graphing()   # initialize matplotlib
seg1.draw("seg1")
seg2.draw("seg2")
plt.show()
```

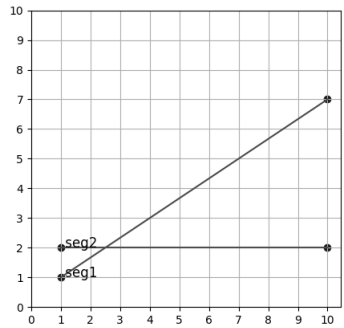


Figure 10.9. Segment intersection

intersects() only returns True or False; to obtain the intersection point, intersectPt() should be called. It treats the two segments as lines of the form $Ax + By + C = 0$ and solves the two equations:

```
def intersectPt(self, seg):
  # where does seg intersect with this segment?

  # Segment AB represented as a1x + b1y = c1
  a1 = self.p2.y - self.p1.y
  b1 = self.p1.x - self.p2.x
  c1 = self.p1.y * b1 + self.p1.x * a1

  # Segment CD represented as a2x + b2y = c2
  a2 = seg.p2.y - seg.p1.y
```

```
    b2 = seg.p1.x - seg.p2.x
    c2 = seg.p1.y * b2 + seg.p1.x * a2

    det = a1 * b2 - a2 * b1
    if det != 0:
      x = (b2 * c1 - b1 * c2) / det
      y = (a1 * c2 - a2 * c1) / det
      return Point(x,y)

    return None
```

Listing 10.3. intersectPt() in the segment class

10.5.1 The Intersection of Multiple Line Segments. The brute-force approach for finding all the intersections of n line segments is to examine every pair. This will involve $n(n-1)/2$ comparisons in the worst case, and so have a running time of $0(n^2)$. However, if the segments are sparsely distributed, then a lot of that work will be to no purpose; a faster algorithm is needed.

The *Bentley-Ottmann* algorithm utilizes a vertical sweep line that moves from left to right over the segments (which are assumed to be non-vertical themselves). As the line moves it keeps a tally of *event points* which consist of segment endpoints and intersections. The points are managed in an queue ordered by their coordinates which means that far fewer comparisons need to be carried out, ultimately reducing the search time to $0(n \log n)$. The drawback is the need for more complex point and segment classes, and a queue which supports the efficient reordering of points as the sweep line progresses. Code for Bentley-Ottmann can be found in `sweep.py` and `treeset.py`, but will not be described here.

10.6 Finding the Convex Hull

A planar region R is convex when all the pairs of points p, q in R, form line segments $\overline{pq}$ that lie completely inside R (see the examples in Fig. 10.10).

The convex hull of a set of points P is the smallest convex region that contains all of them. From a physical perspective, the hull is the shape taken by a rubber band stretched around all the points (Fig. 10.11). (Incidentally, this 'band' technique appears to be a constant time operation, easily beating other approaches, but remember that initialization costs must be factored into the running time.)

The *Graham Scan* utilizes a rotating sweep line with running time $O(n \log n)$. Before the sweep begins there are two set-up stages:

(1) The point with the smallest y value is located, and thereafter called the *anchor*. If there are several smallest points, then the one with the largest x value is chosen. This will take linear time.

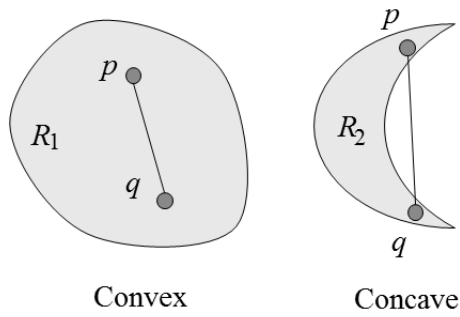


Figure 10.10. Convex and concave regions

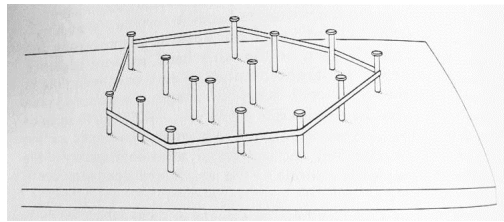


Figure 10.11. A convex hull

(2) The points are sorted based on the angle they make with the anchor. If two points have the same angle, then they're sorted based on their distance from the anchor. This will take $O(n \log n)$ time if merge sort is employed. The angle calculation is assumed to take constant time (see Point.polarAngle() which uses math.atan2()).

The scan starts from the anchor (p_0 in Fig. 10.12), and rotates counter-clockwise through the sorted points.

The sweep line applies a turning test to the last three points encountered (they are labeled as A, B, C in Fig. 10.13). Is the current point (C) to the left or right of the line segment formed by the other two points ($\overline{AB}$)?

If the current point requires a left turn then it's added to the hull and the sweep line moves on to the next point. This occurs in Fig. 10.13 (1) and the new set of points becomes B, C, and D. The question now becomes whether D is to the left or right of $\overline{BC}$?

A right turn means that the point at the end of the line segment can not be part of the convex hull and should be removed. This occurs in Fig. 10.13 (2) when C is discarded. D remains the current point but the line segment changes to $\overline{AB}$.

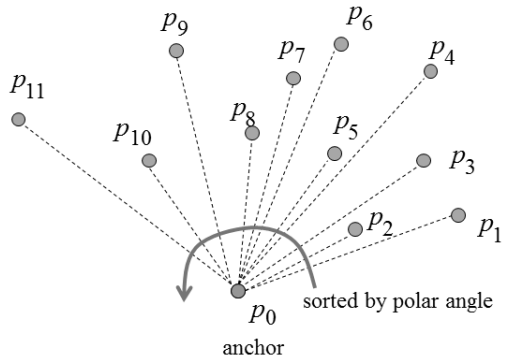


Figure 10.12. The Graham scan

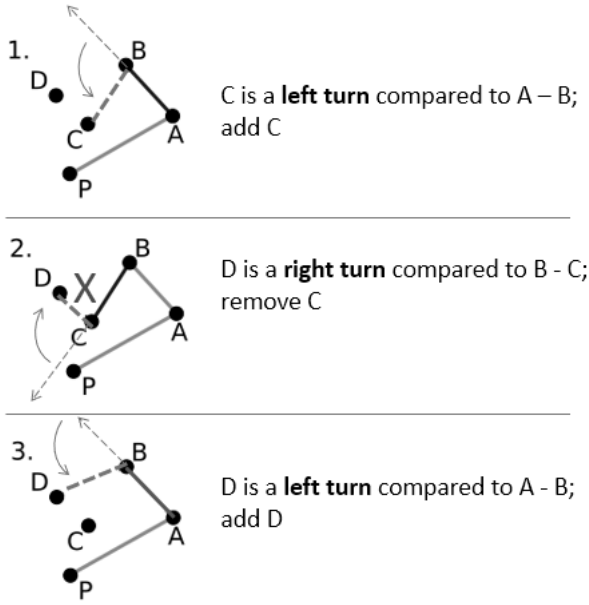


Figure 10.13. Turn testing in the Graham scan

Point removal will continue for as long as the current point and the preceding line segment form a right turn. This will require earlier points in the hull to be stored so they can be used to build new line segments when points are deleted.

The convexHull() code in Listing 10.4 (convexHull.py) utilizes ccw() to determine the turn type.

```
def convexHull(pts):
  # Find a point with the lowest y value,
  anchorPt = pts[0]
  for _, pt in enumerate(pts):
    if pt.y < anchorPt.y:
      anchorPt = pt
    elif pt.y == anchorPt.y and pt.x > anchorPt.x:
      anchorPt = pt
  # print("Anchor:", anchorPt)

  # Sort all the points based on the angle they make with the anchor
      pt.
  ptAngles = []
  for pt in pts:
    ptAngles.append([pt, anchorPt.polarAngle(pt)])
  ptAngles.sort( key=lambda x:x[1])
  sortedPts =  [ pt for [pt,_] in ptAngles]

  hull = [anchorPt, sortedPts[0]]
  # test turning if points
  for pt in sortedPts[1:]:
    while Vector.ccw(hull[-2], hull[-1], pt) <= 0:
      del hull[-1] # removal
    hull.append(pt)

  return hull
```

Listing 10.4. Implementing the convex hull

A series of ccw() tests are applied to the $n-1$ points spread around the anchor. Since ccw() is a constant time operation, the sweep has running time $O(n)$. When this is combined with the earlier set-up costs, Graham's scan takes $O(n \log n)$.

The test code in convexHull.py calls convexHull() with 100 random points. The returned hull is used to draw line segments around the points as shown in Fig. 10.14.

The ∘ point in Fig. 10.14 is the result of a call to isInside() (Listing 10.5) (convexHull.py) which returns True or False depending on whether the point is inside the hull. The code uses ccw() to check if the point is on the left of every segment making up the hull.

```
def isInside(poly, p):
  n = len(poly)
  if n < 3:
    return False
  for i in range(n):
    if Vector.ccw(poly[i], poly[(i+1)%n], p) == -1:  # on right
      return False
```

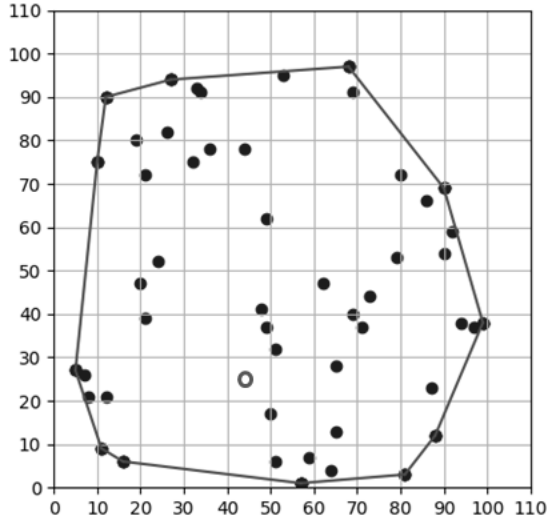


Figure 10.14. The convex hull of random points

```
# p always on left or collinear
return True
```

Listing 10.5. isInside() in convexHull.py

10.7 Finding the Closest Pair of Points

Another classic CG problem is finding a pair of points p, q such that $|p-q|$ is the minimum of all $|p_i - p_j|$ in a set of n points $P = \{p_1, \cdots p_n\}$.

As with the convex hull, there's an $O(n^2)$ algorithm which computes the absolute distances between all pairs of points. There's also a sweep line technique which has running time $O(n \log n)$; we won't discuss either of these, although their Python code can be found in `nearestPts.py` and `closestSweep.py` respectively. Instead, we'll utilize a divide-and-conquer approach.

The divide phase determines the median x-coordinate of the points, which is used to split the search into two halves. This requires the points to be sorted beforehand, which can be done in $O(n \log n)$ time.

The division process recursively progresses until a single distance remains. As the function calls return, the smallest distance is selected and passed up. At the top-level, the search produces two distances (labeled as d_1 and d_2 in Fig. 10.15).

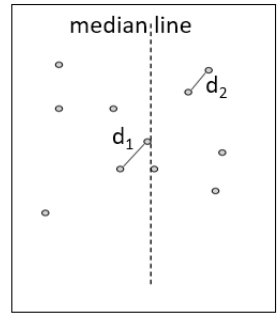


Figure 10.15. Shortest distances in two halved spaces

Result generation is not just a matter of selecting the smaller of the two distances. The search may have missed close points across the median line – occurring in a narrow vertical strip of width $2d$, where d is the minimum of d_1 and d_2 (see Fig. 10.16).

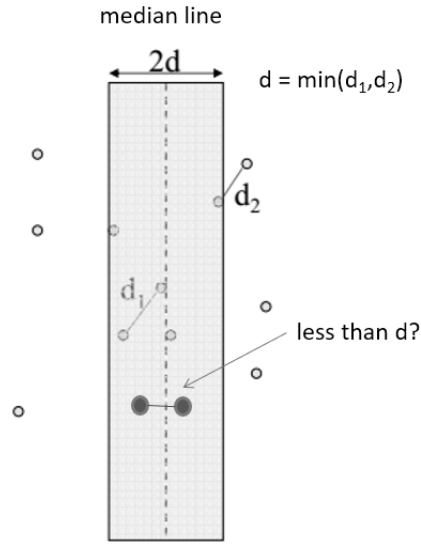


Figure 10.16. The unchecked strip along the median

An efficient way to scan this strip is to sort all of its points into increasing y-order, so that the search needs only move up. The current point being examined

(the one intersecting by dotted box in Fig. 10.17) only has to look at the points in a $d \times 2d$ region above it (the dotted area in the figure).

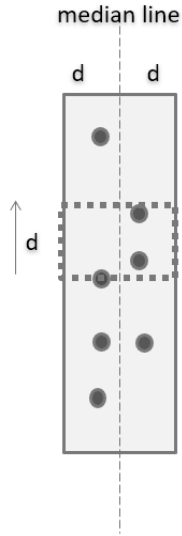


Figure 10.17. Scanning the strip

Searching this area takes a constant amount of time since there can only be a maximum of seven points in the region (see Fig. 10.18). The points cannot be more than a d distance apart, which means that there can be at most one point in each $d/2 \times d/2$ sub-square making up the $d \times 2d$ rectangle, and one of those points is currently being tested.

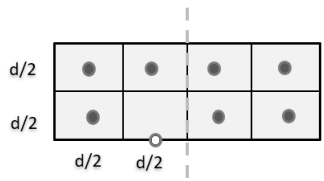


Figure 10.18. Points in the rectangle

The recurrence relation for the running time of the entire algorithm is:

$$T(n) = 2T(n/2) + O(7n)$$

The recursive calls to T() represent the recursive division of the space while the last term is the cost of examining all n point in the median strip. This relation simplifies to $O(n \log n)$.

The code for this algorithm is in Listing 10.6 (closestDC.py).

```
def closestPair(pts, n):
  if n <= 3:
    return pairsSearch(pts, n)

  mid = n//2
  midPoint = pts[mid]
  # divide and conquer
  dl, mpl = closestPair(pts, mid)  # left
  dr, mpr = closestPair(pts[mid:], n-mid) # right

  # use the minimum distance (and its pair)
  if dl < dr:
    d = dl
    mp = mpl
  else:
    d = dr
    mp = mpr

  strip = []
  # build a 2d-wide strip of points
  for i in range(n):
    if abs(pts[i].x - midPoint.x) < d:
      strip.append(pts[i])
  return checkStrip(strip, d, mp)


def checkStrip(strip, minDist, minPair):
  size = len(strip)

  # sort so can move upwards through the strip
  strip = sorted(strip, key=lambda pt: pt.y)

  for i in range(size):
    for j in range(i+1, size):
      # only examine minDist upwards
      if (strip[j].y - strip[i].y) >= minDist:
          break
      d = strip[i].distTo(strip[j])
      if d < minDist:
        minDist = d
        minPair = (strip[i], strip[j])

  return minDist, minPair
```

Listing 10.6. Finding the closest pair

One complication is that the function not only calculates the minimum distance between the points, but also returns the minimum points as a tuple. Another optimization is that when the number of points being considered is small (≤ 3) then the pairs are examined iteratively by pairsSearch().

The test-rig in closestDC.py generates 100 random points, calls closestDC() and uses matplotlib to plot the result, as shown in Fig. 10.19. The two ∘ points are the closest pair, and their coordinates are also printed to standard output.

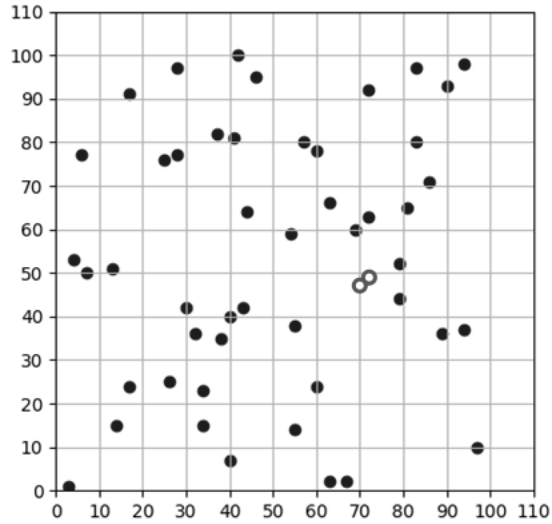


Figure 10.19. Finding the closest points by divide-and-conquer

Exercises

(1) Find an expression that relates the number of vertices of a regular polygon to the number of its diagonals.

(2) Given an arbitrary set of points, find the coordinates of the vertices of a rectangular box that just encloses it.

(3) A polygon can be formed from a sequence of line segments. One complication is that the result may not be *simple*, meaning that some of the shape's line segments cross. Write an algorithm which determines whether an n-vertex polygon is simple. The straightforward approach has running time $O(n^2)$; can you reduce this to $O(n \log n)$?

(4) Given a point $p_0 = (x_0, y_0)$, a right horizontal *ray* from p_0 is the set of points to the right of p_0: $\{p_i = (x_i, y_i) : x_i \geq x_0 \text{ and } y_i = y_0\}$. Show how to determine whether a given right horizontal ray from p_0 intersects a line segment $\overline{rs}$ in constant time. *Hint*: reduce the problem to one of determining if two segments intersect.

(5) One way to determine whether a point p_0 is inside a polygon P is to generate a ray from p_0 and check that it intersects the edge of P an *odd* number of times. Show how to compute this in linear time. *Warning*: make sure your algorithm is correct when the ray intersects the polygon boundary at a vertex and when the ray overlaps a side of the polygon.

(6) *Picking up sticks.* There are n sticks piled up in some random order. Each stick is specified by its endpoints defined as (x, y, z) coordinates. You can assume that no stick is standing vertical. Your task is to pick up all the sticks, one at a time, subject to the condition that you can only pick up a stick if there isn't another stick on top of it.

a) Write code that takes two sticks a and b and reports whether a is above, below, or unrelated to b.

b) Describe an algorithm that determines whether it is possible to pick up all the sticks, and if so, provides a legal order for achieving this.

(7) Let S be a set of n circles in the plane. Describe a sweep algorithm to compute all of the intersection points between the circles. (Note that two circles do not intersect if one lies entirely inside the other.)

11

Continued Fractions

Why Study Continued Fractions?

- **They offer an alternative representations for numbers**. Every real number has a unique simple continued fraction expansion, that's often more revealing than decimal or binary.
- **They underlie efficient algorithms**. The Euclidean algorithm can be viewed as the computation of a continued fraction. This links continued fractions to number theory, modular arithmetic, and greatest common divisors.
- **They solve Pell-type and quadratic Diophantine equations**. Periodic continued fractions for $\sqrt{d}$ can be employed to solve Pell's equation $x^2 - dy^2 = 1$. This connects continued fractions to algebraic number theory and even to conics.
- **They give the best rational approximations to real numbers.** If you want to approximate a real number by a fraction with a small numerator and denominator, its continued fraction convergents are provably the best.

11.1 Introduction

If x is a real number (and not an integer) then we can write

$$x = a_0 + \frac{1}{x_1}, \quad a_0 = \lfloor x \rfloor, \quad x_1 = \frac{1}{x - a_0} > 1.$$

If x_1 is also not an integer, then we can apply the same transformation again

$$x_1 = a_1 + \frac{1}{x_2}, \quad a_1 = \lfloor x_1 \rfloor, \quad x_2 = \frac{1}{x_1 - a_1} > 1.$$

For rational numbers we'll eventually get

$$\begin{aligned} x &= a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \ddots + \cfrac{1}{a_n}}}} \\ &= a_0 + 1/(a_1 + 1(a_2 + 1/(a_3 + \cdots + 1/a_n))\cdots) \end{aligned}$$

This expression is called a *simple continued fraction*, and often written as $[a_0; a_1, a_2, ..., a_n]$; the a_i's are *partial quotients*.

Example. Suppose we start with the rational number

$$\frac{57}{33}$$

Start converting this to a continued fraction:

$$\frac{57}{33} = 1 + \frac{24}{33}.$$

Now invert the remainder:

$$\frac{33}{24} = 1 + \frac{9}{24}.$$

Again:

$$\frac{24}{9} = 2 + \frac{6}{9},$$

and again:

$$\frac{9}{6} = 1 + \frac{3}{6},$$

and finally:

$$\frac{6}{3} = 2$$

Thus

$$\frac{57}{33} = [1; 1, 2, 1, 2]$$

This procedure is implemented by Listing 11.1 (cfLib.py).

```
def cf(numer, denom, maxTerms=10):
  # Return the terms of the continued fraction for
  # the numerator and denominator as a list
  terms = []
  quotient = numer//denom
  rem = numer - quotient*denom
  count = 0
```

```
  while (rem != 0) and (count < maxTerms):
    terms.append(quotient)
    count += 1
    prev_r = rem
    quotient = denom//prev_r
    rem = denom - quotient*prev_r
    denom = prev_r
  terms.append(quotient)
  return terms
```

Listing 11.1. From rational to continued fraction

Example:

```
>>> from cfLib import *
>>> cf(57,33)
[1, 1, 2, 1, 2]
```

Conversely, the continued fraction

$$[2;1,3,2] = 2 + \cfrac{1}{1 + \cfrac{1}{3 + \cfrac{1}{2}}}$$

represents the rational

$$\begin{aligned} 2 + \cfrac{1}{1 + \cfrac{2}{7}} &= 2 + \frac{7}{9} \\ &= \frac{25}{9} \end{aligned}$$

This conversion is implemented by Listing 11.2 (cfLib.py).

```
def cf2rat(cfList):
  # finite continued fraction to rational
  n = len(cfList)
  numer = 1
  denom = 0
  for i in range(n-1, -1, -1):
    n1 = numer*cfList[i] + denom
    denom = numer
    numer = n1
  return (numer, denom)
```

Listing 11.2. From continued fraction to rational

Example

```
>>> from cfLib import *
>>> cf2rat([2,1,3,2])
(25, 9)
```

When we evaluate $[a_0; a_1, a_2, ..., a_n]$, we may choose to stop at a_i, obtaining the i-th *convergent* $r_i = p_i/q_i$.

Returning to $\frac{57}{33}$, the successive convergents for

$$\frac{57}{33} = [1; 1, 2, 1, 2]$$

are

$$\begin{aligned}
p_0/q_0 &= [1] = \frac{1}{1} \\
p_1/q_1 &= [1; 1] = 1 + \frac{1}{1} = \frac{2}{1} \\
p_2/q_2 &= [1; 1, 2] = [1; 3/2] = 1 + \frac{2}{3} = \frac{5}{3} \\
p_3/q_3 &= [1; 1, 2, 1] = [1; 1, 3] = [1; 4/3] = 1 + \frac{3}{4} = \frac{7}{4} \\
p_4/q_4 &= [1; 1, 2, 1, 2] = [1; 1, 2, 3/2] = [1; 1, 8/3] = [1; 11/8] = \frac{19}{11}
\end{aligned}$$

The production of convergents as a list is implemented by Listing 11.3 (cfLib.py).

```
def convergents(cfList, numRats=10):
  p0 = 1; q0 = 0
  p = cfList[0]; q = 1
  i = 1
  err = 1
  cs = [(1,1)]
  while i < len(cfList) and (i < numRats):
    a = cfList[i]
    if a < 1:
      break
    # update the numerator of the convergent
    p1 = a*p + p0
    p0 = p
    p = p1
    # update the denominator
    q1 = a*q + q0
    q0 = q
    q = q1
    cs.append((p,q))
    i += 1
  return cs
```

Listing 11.3. Generate the convergents for a continued fraction

Example:

```
>>> from cfLib import *
>>> convergents([1,1,2,1,2])
[(1, 1), (2, 1), (5, 3), (7, 4), (19, 11)]
```

We can define p_i and q_i (the numerator and denominator of the r_i convergent) recursively as:

$$\begin{array}{lcllcll} p_0 & = & a_0, & & q_0 & = & 1 \\ p_1 & = & a_0a_1 + 1, & & q_1 & = & a_1 \\ & \dots & & & & \dots & \\ p_k & = & p_{k-1}a_k + p_{k-2} & & q_k & = & q_{k-1}a_k + q_{k-2}, \quad k = 2, ..., n. \end{array} \tag{11.1}$$

Indeed,

$$\begin{array}{lcl} p_0/q_0 & = & r_0, \\ p_1/q_1 & = & a_0 + 1/a_1 = r_1, \\ p_2/q_2 & = & r_2 \end{array}$$

and induction shows that

$$p_k/q_k = r_k \text{ for } k = 0, 1, 2, ..., n.$$

11.1.1 From Rational to Irrational. Rational numbers are guaranteed to produce *finite length* continued fraction, but the conversion of irrationals (such as π, $\sqrt{2}$, and e) never terminates. For that reason, their conversion is implemented by cfList() using a generator function cfY() (see Listing 11.4; cfLib.py).

```
def cfList(x, maxTerms=10):
  cfs = []
  i = 0
  for val in cfY(x):
    # print(val)
    cfs.append(val)
    i += 1
    if i > maxTerms:
      break
  return cfs


def cfY(x):
  while x != int(x):
    a = int(x)
    x = 1/(x-a)
    yield a
```

Listing 11.4. From irrational to continued fraction

Examples:

```
>>> r2 = math.sqrt(2)
>>> cfList(r2)
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

```
>>> cfList(r2,30)
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
1, 1, 1, 3, 3, 1, 3, 1, 1, 2]

>>> cfList(math.pi)
[3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3]

>>> cfList(math.e, 18)
[2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12, 1]

>>> phi = (math.sqrt(5)+1)/2
>>> cfList(phi, 40)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1]
```

The problem with this approach is that rounding errors associated with Python's floats eventually start to overwhelm the calculations in cfY() and produce incorrect quotients. This can be seen in the long continued fractions for $\sqrt{2}$ and ϕ (the golden ratio), which should be $[1;\overline{2}]$ and $[1;\overline{1}]$ respectively (a bar over a digit (or digits) means that it is repeated indefinitely).

One partial solution is to switch to the Python decimal version of cfY() shown in Listing 11.5 (cfLib.py).

```
def cfDecList(x, maxTerms=20):
  cfs = []
  i = 0
  for val in cfDecY(x):
    # print(val)
    cfs.append(val)
    i += 1
    if i > maxTerms:
      break
  return cfs


def cfDecY(x):
  decimal.getcontext().prec = 100
  d1 = D(1)
  dx = D(x)
  while dx != math.trunc(dx):
    a = math.trunc(dx)
    dx = d1/(dx-a)
    yield a
```

Listing 11.5. From irrational to continued fraction using decimals

Examples:

```
>>> phi = (math.sqrt(5)+1)/2
>>> cfDecList(phi, 40)
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 7, 1]

>>> cfDecList('1.6180339887498948482045868343656381177203091798057628621
3544862270526046281890', 100)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1]
```

One issue illustrated by the first example above is that supplying an irrational encoded as a float may not actually improve the continued fraction result. This is due to the inherent inaccuracy of the float, and the way that the decimal module converts a float to a decimal. It's much better to supply the irrational as a string (as in the second example), making sure to include enough digits for the continued fraction's calculation.

Many surprising patterns are revealed in the irrationals when they're converted to continued fractions. For example, the continued fraction expansion of e is:

$$e = [2; 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12, 1, 1, 14, 1, 1, 16, \cdots]$$

But the continued fraction of π doesn't appear to have any regularity:

$$\pi = [3; 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2, 2, 2, 1, 84, 2, 1, 1, 15, \\ 3, 13, 1, 4, 2, 6, 6, 99, 1, 2, 2, 6, 3, 5, 1, 1, 6, \cdots]$$

One objection to continued fractions is that they seem to make arithmetic more difficult. For example, it's not at all obvious how to add or multiply two continued fractions. This has been addressed by Bill Gosper who devised algorithms for computing with continued fractions term-by-term. His unpublished paper can be found at `https://web.archive.org/web/20030202011209/http://www.tweedledum.com/rwg/cfup.htm`, and is very well explained in a online article by Ross Dempsey at `https://srossd.com/posts/2020-09-18-gosper-1/`.

By far the best beginners book on continued fractions is by Olds [**Old63**], which we recommend highly.

11.2 Continued Fractions and Euclid's Algorithm

We apply Euclid's GCD algorithm (see 2.1.1) to two positive integers n_0 and n_1, where $n_0 \geq n_1 > 0$:

$$\begin{aligned} n_0 &= q_1 n_1 + n_2 \\ n_1 &= q_2 n_2 + n_3 \\ &\cdots \\ n_{k-2} &= q_{k-1} n_{k-1} + n_k \\ n_{k-1} &= q_k n_k \end{aligned}$$

From these equations we can obtain

$$\frac{n_0}{n_1} = [q_1; q_2, \cdots, q_k].$$

Example:

$$\begin{aligned} 271828 &= 2 * 100000 + 71828 \\ 100000 &= 1 * 71828 + 28172 \\ 71828 &= 2 * 28172 + 15484 \\ 28172 &= 1 * 15484 + 12688 \\ 15484 &= 1 * 12688 + 2796 \\ 12688 &= 4 * 2796 + 1504 \\ 2796 &= 1 * 1504 + 1292 \end{aligned} \qquad \begin{aligned} 1504 &= 1 * 1292 + 212 \\ 1292 &= 6 * 212 + 20 \\ 212 &= 10 * 20 + 12 \\ 20 &= 1 * 12 + 8 \\ 12 &= 1 * 8 + 4 \\ 8 &= 2 * 4 \end{aligned}$$

$$\text{And so } \frac{271828}{100000} = [2; 1, 2, 1, 1, 4, 1, 1, 6, 10, 1, 1, 2].$$

We can double check by using cf() (Listing 11.1):

```
>>> cf(271828, 100000, 20)
[2, 1, 2, 1, 1, 4, 1, 1, 6, 10, 1, 1, 2]
```

11.3 Continued Fractions for Square Roots

A continued fraction is eventually periodic if and only if it represents a *quadratic* irrational number (i.e. a solution to a quadratic equation with rational coefficients). The proof is a bit technical, but the idea is to establish a recurrence among the terms of the continued fraction. A simple example is $\sqrt{2}$ whose continued fraction is

$$[1; 2, 2, 2, \cdots]$$

We know that $\sqrt{2}$ is a solution to $x^2 = 2$, which we can write as $x^2 - 1 = 1$, or $(x-1)(x+1) = 1$, so

$$x = 1 + \frac{1}{1+x}$$

Now let $x = [1; a1, a2, \cdots]$, and substitute it into the equation above to get

$$\begin{aligned}[1; a_1, a_2, \cdots] &= 1 + \frac{1}{[2; a_1, a_2, \cdots]} \\ &= [1; 2, a_1, a_2, \cdots] \quad \text{or} \quad [1; \overline{2}]\end{aligned}$$

Comparing the continued fractions term by term, we find that $a_i = 2$ for all $i = 1, 2, \cdots$.

Let's move along to the continued fraction for $\sqrt{3}$. By choosing a starting value carefully, we can convert $\sqrt{3}$ in a similar way to $\sqrt{2}$:

$$\begin{aligned}\sqrt{3} &= 1 + (\sqrt{3} - 1), \\ \frac{1}{\sqrt{3}-1} &= \frac{\sqrt{3}+1}{2} = 1 + \frac{\sqrt{3}-1}{2}, \\ \frac{2}{\sqrt{3}-1} &= \sqrt{3} + 1 = 2 + (\sqrt{3} - 1), \\ \frac{1}{\sqrt{3}-1} &= 1 + \frac{\sqrt{3}-1}{2}, \\ &\cdots\end{aligned}$$

Thus

$$\sqrt{3} = [1; \overline{1, 2}]$$

We can check that $\sqrt{d}$ always has a periodic expansion, by using cfDecList() (Listing 11.5) and decimal.sqrt()

```
>>> decimal.getcontext().prec = 100
>>> cfDecList( D(7).sqrt(), 30)
[2, 1, 1, 1, 4, 1, 1, 1, 4, 1, 1, 1, 4, 1, 1, 1, 4, 1, 1, 1, 4, 1, 1, 1,
4, 1, 1, 1, 4, 1, 1]

>>> cfDecList( D(43).sqrt(), 30)
[6, 1, 1, 3, 1, 5, 1, 3, 1, 1, 12, 1, 1, 3, 1, 5, 1, 3, 1, 1, 12, 1, 1,
3, 1, 5, 1, 3, 1, 1, 12]

>>> cfDecList( D(94).sqrt(), 40)
[9, 1, 2, 3, 1, 1, 5, 1, 8, 1, 5, 1, 1, 3, 2, 1, 18, 1, 2, 3, 1, 1, 5, 1,
8, 1, 5, 1, 1, 3, 2, 1, 18, 1, 2, 3, 1, 1, 5, 1, 8]

>>> cfDecList( D(919).sqrt(), 100)
[30, 3, 5, 1, 2, 1, 2, 1, 1, 1, 2, 3, 1, 19, 2, 3, 1, 1, 4, 9, 1, 7, 1, 3,
6, 2, 11, 1, 1, 1, 29, 1, 1, 1, 11, 2, 6, 3, 1, 7, 1, 9, 4, 1, 1, 3, 2,
19, 1, 3, 2, 1, 1, 1, 2, 1, 2, 1, 5, 3, 60, 3, 5, 1, 2, 1, 2, 1, 1, 1, 2,
3, 1, 19, 2, 3, 1, 1, 4, 9, 1, 7, 1, 3, 6, 2, 11, 1, 1, 1, 29, 1, 1, 1, 11,
2, 6, 3, 1, 7, 1]
```

```
>>> cfDecList( D(991).sqrt(), 100)
[31, 2, 12, 10, 2, 2, 2, 1, 1, 2, 6, 1, 1, 1, 1, 3, 1, 8, 4, 1, 2, 1, 2, 3,
1, 4, 1, 20, 6, 4, 31, 4, 6, 20, 1, 4, 1, 3, 2, 1, 2, 1, 4, 8, 1, 3, 1, 1,
1, 1, 6, 2, 1, 1, 2, 2, 2, 10, 12, 2, 62, 2, 12, 10, 2, 2, 2, 1, 1, 2, 6,
1, 1, 1, 1, 3, 1, 8, 4, 1, 2, 1, 2, 3, 1, 4, 1, 20, 6, 4, 31, 4, 6, 20, 1,
4, 1, 3, 1, 1, 9]
```

In summary:

$$\begin{aligned}
\sqrt{7} &= [2;\overline{1,1,1,4}] \\
\sqrt{43} &= [6;\overline{1,1,3,1,5,1,3,1,1,12}] \\
\sqrt{94} &= [9;\overline{1,2,3,1,1,5,1,3,1,5,1,1,3,2,1,18}] \\
\sqrt{919} &= [30;\overline{3,5,1,2,1,2,1,1,1,2,\cdots,2,1,5,3,60}] \\
\sqrt{991} &= [31;\overline{2,12,10,2,2,2,1,1,2,6,\cdots,2,10,12,2,62}]
\end{aligned}$$

The periods for $\sqrt{919}$ and $\sqrt{991}$ have 62 and 60 terms respectively. Each period always ends with $2\lfloor\sqrt{d}\rfloor$ and, excluding the last term, is a palindrome (it reads the same forwards and backwards).

11.4 The Law of Best Approximation

If the rational r/s is closer to an irrational x than the n-th convergent $r_n = p_n/q_n$ of x, then $s > q_n$. In other words, the n-th convergent is the 'best' approximation to the irrational for that denominator size. If you want a better approximation, then you'll need a larger denominator, which can be found in subsequent convergents in the continued fraction. We could prove this, but instead refer to Sec. 2.12, where the topic is dealt with in a more general way.

The following examples generate 100-term continued fractions and convergents for π, e, and ϕ.

```
>>> cfPi = cfDecList("3.14159265358979323846264338327950288419716939
9937510582097494459230781640628620899862803482534211706 79", 100)
>>> convergents(cfPi, 5)
[(1, 1), (22, 7), (333, 106), (355, 113), (103993, 33102)]

>>> cfE = cfDecList("2.71828182845904523536028747135266249775724709
369995957496696762772407663035354759457138217852516642 74", 100)
>>> convergents(cfE)
[(1, 1), (3, 1), (8, 3), (11, 4), (19, 7), (87, 32), (106, 39),
(193, 71), (1264, 465), (1457, 536)]

>>> cfPhi = cfDecList("1.61803398874989484820458683436563811772030917
980576286213544862270526046281890244970720720418939113 74", 100)
>>> convergents(cfPhi, 20)
[(1, 1), (2, 1), (3, 2), (5, 3), (8, 5), (13, 8), (21, 13), (34, 21),
```

```
(55, 34), (89, 55), (144, 89), (233, 144), (377, 233), (610, 377),
(987, 610), (1597, 987), (2584, 1597), (4181, 2584), (6765, 4181),
(10946, 6765)]
```

We can say just how close the convergents get to the irrational. Let $\alpha = [a_0; a_1, a_2, ...]$, and let the n-th convergent be p_n/q_n, with $p_0 = a_0$ and $q_0 = 1$. Then

$$\frac{1}{(a_{n+1}+2)q_n^2} < \left|\alpha - \frac{p_n}{q_n}\right| < \frac{1}{a_{n+1}\, q_n^2}$$

This implies that larger denominators in a convergent is a sign that the convergent is getting closer to the irrational's actual value. For instance, consider the fourth convergent for π:

$$[3; 7, 15, 1, 292] = \frac{355}{113} = 3.14159\underline{292035}\cdots$$

It differs from π (3.141592 65359...) by no more than $1/(292 \cdot 113^2) \approx 2.2e-7$.

This also explains why $\phi = [1; 1, 1, 1, \cdots]$ is considered the most difficult number to approximate by rationals, since $a_{n+1}\, q_n^2$, grows as slowly as possible.

11.4.1 Convergents for ϕ. All of the numerators and denominators of the convergents for ϕ above are Fibonacci numbers. Indeed, each numerator is the successor of its denominator in the Fibonacci series.

This is easily proved: ϕ can be expressed as the equation $\phi^2 - \phi - 1 = 0$, or equivalently

$$\phi = 1 + \frac{1}{\phi}$$

This can be expanded into the continued fraction $[1; 1, 1, 1, \cdots]$, or written as the sequence:

$$\phi_{n+1} = 1 + \frac{1}{\phi_n}, \quad \phi_0 = 1.$$

Suppose

$$\phi_n = \frac{r_n}{t_n}$$

Then

$$\phi_{n+1} = \frac{r_{n+1}}{t_{n+1}} = 1 + \frac{t_n}{r_n} = \frac{t_n + r_n}{r_n}$$

Equate the corresponding numerators and denominators:

$$\begin{aligned} r_{n+1} &= t_n + r_n \\ t_{n+1} &= r_n \end{aligned}$$

which can be rewritten as a Fibonacci series for generating r's or a series to produce t's.

Another way to obtain the series is by utilizing the general recurrence relation for the numerators and denominators of convergents in Equ. (11.1). Recall:

$$\begin{aligned} p_k &= a_k p_{k-1} + p_{k-2}, \quad k = 2, ..., n \\ q_k &= a_k q_{k-1} + q_{k-2} \end{aligned}$$

In the case of ϕ, all of the a_k quotients = 1 and so the p_k and q_k relations simplify to Fibonacci series.

11.4.2 Convergent Relative Errors. convergentsErr() is a slight variant of Listing 11.3 in cfLib.py which prints the relative error of each convergent in terms of the previous one. It shows a rapid decrease in the error, and also an oscillation around the 'true' value. For example, for π:

```
>>> convergentsErr(cfPi)
22/7 = 3.142857142857143 ; rel error: 0.047619047619047596
333/106 = 3.141509433962264 ; rel error: -0.00042881646655234076
355/113 = 3.1415929203539825 ; rel error: 2.6575247814301264e-05
103993/33102 = 3.1415926530119025 ; rel error: -8.50976198402835e-08
[(1, 1), (22, 7), (333, 106), (355, 113), (103993, 33102)]
```

Exercises

(1) There is a deep result which says that for rationals (and irrationals) the proportion of 1's, 2's, 3's, and 4's that occur in the quotients of the continued fraction will be in the approximate proportions $log_2(4/3) \approx 0.41504$, $log_2(9/8) \approx 0.16992$, $log_2(16/15) \approx 0.09311$, and $log_2(25/24) \approx 0.05890$.

These constants can be estimated by calculating the average number of 1's, 2's, 3's, and 4's in the quotient list, excluding the first integer term. For example, Listing 11.6 (quotFreqs.py) averages the number of 1's:

```
r = random.random()
print(r)
cs = cfList(r)
print(cs)
num1s = cs.count(1)
quotLen = len(cs)-1   # don't count leading 0
print(f"num 1s = {num1s}, est p1 = {num1s/quotLen:.5f}")
```

Listing 11.6. Counting quotients

Here are two runs of the program:

```
> python quotFreqs.py
0.2795543177484361
[0, 3, 1, 1, 2, 1, 2, 1, 6, 1, 2]
num 1s = 5, est p1 = 0.50000
```

```
> python quotFreqs.py
0.041800404500606736
[0, 23, 1, 12, 43, 3, 1, 7, 1, 3, 3]
num 1s = 3, est p1 = 0.30000
```

Modify Listing 11.6 to gather more empirical information over $n = 10000$ trials to see if the 1's, 2's, 3's, and 4's averages really do approach these constants.

(2) We can use the continued fraction for $\sqrt{d}$ to find the integer solutions of Pell's equations $x^2 - dy^2 = \pm 1$. It has been shown that the (x, y) solutions (if they exist at all) will be one of the convergent pairs (p_i, q_i).

Some outputs from our pell.py:

```
> python pell.py
d? 61
Success: 10 (29718, 3805) -1
Success: 21 (1766319049, 226153980) 1

> python pell.py
d? 109
Success: 14 (8890182, 851525) -1
Success: 29 (158070671986249, 15140424455100) 1
```

The two 'Success' lines indicate that solutions were found for both Pell forms (± 1). The bracketed values are the (x, y) solutions, and the initial integer is the index position of the matching convergent. This index is very suggestive of a general pattern for Pell equation solutions. What is that pattern?

12

Discrete-Time Markov Chains

Why Study Discrete-Time Markov Chains (DTMCs)?

- **They model random processes that can change over time**. Many real systems evolve in discrete steps, including queues, populations, networks, games, and genetic sequences.
- **They support "memoryless" behaviour**. This Markov property captures situations where the next state depends only on the current state, not the full past.
- **They can predict long-run behaviour**. DTMCs let you determine what will happen "in the long run", via stationary distributions, recurrence/absorption probabilities, mixing times, and limiting behaviour.
- **They connect numerous areas of mathematics**. DTMCs link probability, linear algebra, graph theory, and dynamic systems.

12.1 Introduction

The state X_n of a discrete-time Markov chain changes at each time step n, but is limited to a finite set S of possible states, called the state space. When the state happens to be S_i, there is probability p_{ij} that the next state is equal to S_j. In other words,

$$p_{ij} = \mathbf{P}(X_{n+1} = S_j \mid X_n = S_i). \quad S_i, S_j \in S.$$

A key assumption is that the same transition probability p_{ij} applies whenever state S_i is visited, no matter what happened in the past and how state S_i was

reached. Put another way, the probability of the next state X_{n+1} depends *only* on the value of the present state X_n.

The most convenient way to encode a Markov chain is as a transition probability matrix whose element in the S_ith row and S_jth column is p_{ij}:

$$\begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1m} \\ p_{21} & p_{22} & \cdots & p_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mm} \end{bmatrix}$$

Markov chains also lend themselves to a visual representation as directed graphs whose nodes are the states and the arcs are labeled with the transition probabilities.

Markov chains can be used to model a wide range of behaviors, such as random walks, a Gambler's ruin (e.g. section 3.4), queuing, and many types of dynamic system involving probabilistic change (e.g. in signal processing, communications, and economics).

An important example is Google's original PageRank algorithm, which was based on calculating the importance of a page in terms of the importance of the pages linking to it. PageRank can be thought of as a model of a Web user's behavior. The 'surfer' begins on a random page and keeps clicking on links, never hitting the 'back' button, but, after getting bored, will start over on a new random page. The Markov chain representing this behavior (i.e. Web pages are states, and hyperlinks are probabilistic arcs) is executed until it achieves a steady state. The importance of a page is the long-term fraction of the time spent on that page in the steady state. Details can be found in Chapter 10 of Lay's *Linear Algebra and its Applications* [**LLM21**], and it's also explained in video lecture 33 of Harvard's Statistics 110 course on YouTube (`https://www.youtube.com/watch?v=Q-pCzTpwPBU&list=PL2SOU6wwxB0uwwH80KTQ6ht66KWxbzTIo&index=33`, starting at around the 17 minute mark).

Most textbooks on probability will have at least one chapter dedicated to Markov chains. For example, Chapter 11 of Grinstead and Snell's *Introduction to Probability*, 2nd edition [**GS97**], which is also online at `https://stats.libretexts.org/Bookshelves/Probability_Theory/Introductory_Probability_(Grinstead_and_Snell)`. Another is Chapter 7 of Bertsekas and Tsitsiklis' *Introduction to Probability*, 2nd edition [**BT08**]. The book has been used in several courses at MIT, which are on YouTube at 'MIT RES.6-012 Introduction to Probability, Spring 2018' (`https://www.youtube.com/playlist?list=PLUl4u3cNGP60hI9ATjSFgLZpbNJ7myAg6`) and 'MIT 6.041SC Probabilistic Systems Analysis and Applied Probability' (`https://www.youtube.com/playlist?list=PLUl4u3cNGP60A3XMwZ5sep719_nh95qOe`).

12.1.1 Example: Weather in Oz. According to Kemeny, Snell, and Thompson [**KST74**], the Land of Oz never has two nice days in a row. If they do have a nice day, then they're just as likely to have snow or rain the next day. If they have snow or rain, then there's an even chance of the same weather tomorrow. If there's a change then its only to nice weather half of the time.

If we label the states as R (Rain), N (Nice), and S (Snow), then the transition matrix becomes:

$$\mathbf{P} = \begin{array}{c} \\ R \\ N \\ S \end{array} \begin{array}{c} \begin{array}{ccc} R & N & S \end{array} \\ \begin{pmatrix} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/4 & 1/4 & 1/2 \end{pmatrix} \end{array}$$

This is stored in the file `OzWeather.txt` as:

```
labels: R N S

0.5  0.25 0.25
0.5  0    0.5
0.25 0.25 0.5
```

The directed graph generated by `display.py` for this matrix is shown in Fig. 12.1.

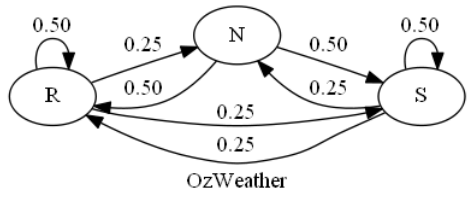


Figure 12.1. The Oz Weather Markov Chain as a Directed Graph

12.1.2 Example: Two Urns. Suppose we have two urns, labeled A and B, with N balls distributed between them. At any given moment, there are n balls in urn A and (obviously) $N - n$ balls in urn B. At each time step, we apply the procedure:

(1) Randomly choose one of the N balls (with equal probability).

(2) With probability q, move the chosen ball from its current urn into the other one. Otherwise, with probability $1 - q$, leave the ball where it is.

If there are n balls in urn A, we have probability n/N of choosing a ball from that urn, followed by a probability of q of moving it to urn B. Using similar reasoning, the three possible outcomes are:

- Move a ball from A to B: probability nq/N

- Move a ball from B to A: probability $(N-n)q/N$
- Leave the urns unchanged: probability $1-q$

We'll label the states of the system using $n \in 0, 1, ..., N$, corresponding to the number of balls in urn A. The graph for the Markov chain is shown in Fig. 12.2.

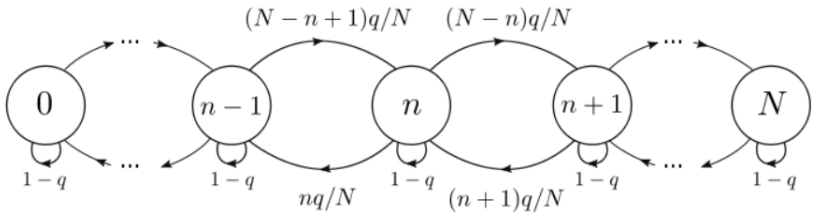


Figure 12.2. Two Urns with N balls and probability q of moving

Focusing on the n balls state in Fig. 12.2, the three possible changes are represented by the three arcs leaving that state.

urnsN.py generates a transition matrix for the urns problem given suitable values for N and q:

```
> python urnsN.py
N? 5
q=? 0.5
urnsN:
        0       1       2       3       4       5
0        0.50    0.50    0.00    0.00    0.00    0.00
1        0.10    0.50    0.40    0.00    0.00    0.00
2        0.00    0.20    0.50    0.30    0.00    0.00
3        0.00    0.00    0.30    0.50    0.20    0.00
4        0.00    0.00    0.00    0.40    0.50    0.10
5        0.00    0.00    0.00    0.00    0.50    0.50
```

The graph is shown in Fig. 12.3.

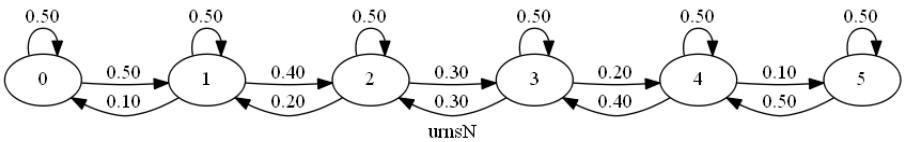


Figure 12.3. Two Urns with 5 balls and probability 1/2 of moving

The urns problem is a special case of the Ehrenfest model (`https://en.wikipedia.org/wiki/Ehrenfest_model`), which has been used to model gas diffusion.

12.2 n-Step Transitions

Most Markov chain problems require the calculation of state probability at some future time based on the current state. These n-step transition probabilities can be expressed as:

$$r_{ij}(n) = \mathbf{P}(X_n = S_j | X_0 = S_i).$$

$r_{ij}(n)$ is the probability that the state after n time steps will be S_j, given that the current state is S_i. It can be calculated recursively:

$$r_{ij}(n) = \sum_{k=1}^{m} r_{ik}(n-1)p_{kj}, \quad \text{for } n > 1, \text{ and all } S_i, S_j$$

starting with

$$r_{ij}(1) = p_{ij}$$

Fig. 12.4 is a visual interpretation of this equation.

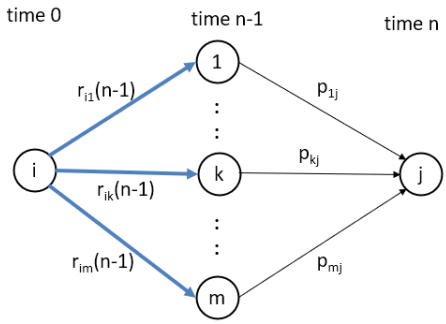


Figure 12.4. The $r_{ij}(n)$ Equation

The probability of being in state S_j at time n is the sum of the probabilities $r_{ik}(n-1)p_{kj}$ of the different ways of reaching S_j.

An equivalent way of representing $r_{ij}(n)$ is as a matrix obtained by multiplying the matrix for $r_{ik}(n-1)$ (which holds the $(n-1)$-step transition probabilities) to the one-step transition probability matrix P. This recursive definition can be rewritten as n powers of P, P^n, such that the ijth entry p_{ij}^n of P^n is the probability that the Markov chain, starting in state S_i, will be in state S_j after n steps.

We can use `powersTest.py` on the Land of Oz example to compute the successive powers of its matrix from 1 to 6. The results are shown below.

```
> python powersTest.py
fnm: OzWeather
Reading: OzWeather.txt
n: 6
```

```
mat^1:
        R       N       S
R       0.5000  0.2500  0.2500
N       0.5000  0.0000  0.5000
S       0.2500  0.2500  0.5000

mat^2:
        R       N       S
R       0.4375  0.1875  0.3750
N       0.3750  0.2500  0.3750
S       0.3750  0.1875  0.4375

mat^3:
        R       N       S
R       0.4062  0.2031  0.3906
N       0.4062  0.1875  0.4062
S       0.3906  0.2031  0.4062

mat^4:
        R       N       S
R       0.4023  0.1992  0.3984
N       0.3984  0.2031  0.3984
S       0.3984  0.1992  0.4023

mat^5:
        R       N       S
R       0.4004  0.2002  0.3994
N       0.4004  0.1992  0.4004
S       0.3994  0.2002  0.4004

mat^6:
        R       N       S
R       0.4001  0.2000  0.3999
N       0.3999  0.2002  0.3999
S       0.3999  0.2000  0.4001
```

After six time steps, the weather probabilities on each row are the same, to 2 dp accuracy, indicating that this chain has reached a *steady state*. The probabilities in any row can be interpreted as the long run fraction of time that the system will spend in each state, and the fact that each row is identical means that the probability of X_{n+1} no longer depends on X_n. For example, the probability of a nice day tomorrow is approximately $\frac{1}{5}$ no matter what the weather is today.

The following results are obtained when calling powers() on the urns matrix generated by urnsN.py, with $n = 40$:

```
> python urnsN.py
N? 5
q=? 0.5
Power n: 40
```

```
mat^1:
        0       1       2       3       4       5
0       0.5000  0.5000  0.0000  0.0000  0.0000  0.0000
1       0.1000  0.5000  0.4000  0.0000  0.0000  0.0000
2       0.0000  0.2000  0.5000  0.3000  0.0000  0.0000
3       0.0000  0.0000  0.3000  0.5000  0.2000  0.0000
4       0.0000  0.0000  0.0000  0.4000  0.5000  0.1000
5       0.0000  0.0000  0.0000  0.0000  0.5000  0.5000

  : many steps not shown

mat^39:
        0       1       2       3       4       5
0       0.0313  0.1563  0.3126  0.3124  0.1562  0.0312
1       0.0313  0.1563  0.3125  0.3125  0.1562  0.0312
2       0.0313  0.1563  0.3125  0.3125  0.1562  0.0312
3       0.0312  0.1562  0.3125  0.3125  0.1563  0.0313
4       0.0312  0.1562  0.3125  0.3125  0.1563  0.0313
5       0.0312  0.1562  0.3124  0.3126  0.1563  0.0313

mat^40:
        0       1       2       3       4       5
0       0.0313  0.1563  0.3125  0.3125  0.1562  0.0312
1       0.0313  0.1563  0.3125  0.3125  0.1562  0.0312
2       0.0313  0.1563  0.3125  0.3125  0.1562  0.0312
3       0.0312  0.1562  0.3125  0.3125  0.1563  0.0313
4       0.0312  0.1562  0.3125  0.3125  0.1563  0.0313
5       0.0312  0.1562  0.3125  0.3125  0.1563  0.0313
```

This chain also reaches a steady state, although it takes more time steps to do so.

12.3 Simulation

A good way to obtain a feel for the behavior of a Markov chain is to start it in a random state and execute the chain over multiple time steps. This should be repeated several times to see how different start states affect the long-term behavior.

Listing 12.1 (simMarkov.py) implements this approach after loading a Markov chain from a text file.

```
RUN_LEN = 20

fnm = input("fnm? ")
labels, P = readMarkov(fnm)
print("P:")
matLabels(P, labels)
# graph(fnm, labels, P)
```

```
p = P.data
n = len(labels)
count = [0] * n   # count of states visited
for _ in range(NUM_RUNS):
  print("> ", end=' ')
  currState = random.choice(range(n)) # random start
  for _ in range(RUN_LEN):
    r = random.random()
    for j in range(n):
      # decrement prob from r until it reaches 0 (or less)
      r -= p[currState][j]
      if r < 0:  # what state are we in?
        currState = j
        print(labels[currState], end=' ')
        break
    count[currState] += 1
  print()

print("\n\nLabel  Counts  Fraction")
totSteps = NUM_RUNS*RUN_LEN
for i in range(n):
  print(f"{labels[i]:<7}{count[i]:<8}{(count[i] / totSteps):.3f}")
```

Listing 12.1. Simulate the Behavior of a Markov Chain

For the Markov Chain in OzWeather.txt, its behavior over multiple runs is remarkably consistent, suggesting that the starting state has little effect on the chain's long-term behavior:

```
> python simMarkov.py
fnm? OzWeather
Reading: OzWeather.txt
P:
        R       N       S
R        0.50    0.25    0.25
N        0.50    0.00    0.50
S        0.25    0.25    0.50

>  S S S R R R R R R N R S R S S S S R R
>  S N S N S S R R N R N R N S S S R S N S
>  S S S R N S S S S S R R R R S R R R R N
>  S S S S S S S S R N S S S S S N R S R N S
>  S N S N S N R R R R R R R N S S R R R N
>  S N R R R R S S S S R N R R N R R R N S
     : many lines not shown
>  S N S N S S N S S S S N R R N S N R N S
>  N R N R R R N S S S S R N R S R R R R S
>  R R R R R N R N S N S S S N R R S S R R N
>  S R R R S R R R S N R R N S R R S S R R
```

```
Label  Counts  Fraction
R      398     0.398
N      189     0.189
S      413     0.413
```

The final output is a tally of the states at the end of the runs. After 50 runs, each one involving 20 time-steps, the fraction of times that the final state is R, N, or S are close to the steady-state values in P^n.

The behavior is quite different for drunk6.txt, an example that we'll discuss in Section 12.7. It appears to get 'stuck' in either state 0 or 5 depending on which one is nearer to the start state:

```
> python simMarkov.py
fnm? drunk6
Reading: drunk6.txt
P:
        0       1       2       3       4       5
0        1.00    0.00    0.00    0.00    0.00    0.00
1        0.50    0.00    0.50    0.00    0.00    0.00
2        0.00    0.50    0.00    0.50    0.00    0.00
3        0.00    0.00    0.50    0.00    0.50    0.00
4        0.00    0.00    0.00    0.50    0.00    0.50
5        0.00    0.00    0.00    0.00    0.00    1.00

>  5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
>  2 3 4 3 2 3 4 3 4 3 4 3 4 3 2 1 0 0 0 0
>  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
>  4 3 2 1 2 3 4 3 2 3 4 5 5 5 5 5 5 5 5 5
>  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
>  2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      : many lines not shown
>  4 3 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
>  5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
>  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
>  2 3 4 3 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

Label  Counts  Fraction
0      463     0.463
1      17      0.017
2      39      0.039
3      53      0.053
4      36      0.036
5      392     0.392
```

This example shows the utility of running a Markov chain with different start states.

12.4 Classification of States

Before we proceed, it's useful to define a few types of state found in Markov chains.

12.4.1 Accessible States. A state S_j is *accessible* from a state S_i if for some n, the n-step transition probability $r_{ij}(n)$ is positive, i.e., there is a chance of reaching S_j from S_i after some number of time steps.

12.4.2 Recurrent States. Let $A(i)$ be the set of states that are accessible from S_i. We say that S_i is *recurrent* if for every S_j that's accessible from S_i, then S_i is also accessible from S_j.

When we start in a recurrent state, S_i, there's always some probability of returning to S_i and, given enough time, that's certain to happen. Also, if a recurrent state is reached once, then it's certain to be revisited an infinite number of times.

12.4.3 Transient States. A state is *transient* if it's not recurrent. Thus, a state S_i is transient if there's a state $S_j \in A(i)$ such that S_i is *not* accessible from S_j. After each visit to state S_i, there's a positive probability that we'll visit state S_j, and once that occurs then state S_i can no longer be visited. As a consequence, a transient state can only be visited a finite number of times.

Fig. 12.5 shows a transition graph labeled with its recurrent and transient states. State 2 is transient since the system can eventually move to state 1 or 3, and state 2 is thereafter inaccessible. States 3 and 4 are recurrent since they're always accessible from each other.

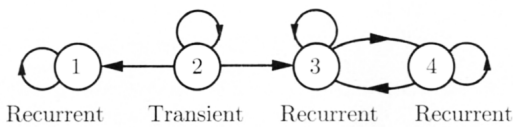


Figure 12.5. Recurrent and Transient States

12.4.4 Recurrent Classes. If S_i is a recurrent state, then the set of states $A(i)$ that are accessible from S_i form a *recurrent class*, meaning that the states in $A(i)$ are all accessible from each other, and no state outside $A(i)$ is accessible from them.

In Fig. 12.5, states 3 and 4 form a single recurrent class, and state 1 a second class.

12.4.5 Periodicity. A recurrent class is said to be *periodic* if its states can be grouped in $d > 1$ disjoint subsets $Y_1, \cdots, Y_d$ so that all the transitions in one subset point to the next subset. An example, where $d = 3$, is shown in Fig. 12.6.

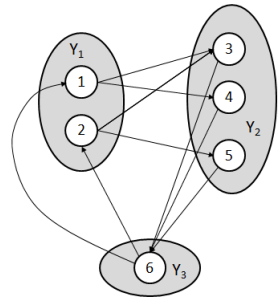


Figure 12.6. A Periodic Recurrent Class

More precisely:

$$\text{if } i \in Y_k \text{ and } p_{ij} > 0 \quad \text{then} \begin{cases} j \in Y_{k+1} & \text{if } k = 1, \cdots, d-1, \\ j \in Y_1 & \text{if } k = d. \end{cases}$$

We cycle through the subsets, and after a multiple of d steps return to the subset where we started.

A recurrent class that isn't periodic, is said to be *aperiodic*.

12.5 Steady-State Behavior

We can now be more precise about the meaning of steady-state behavior when the n-step transitions $(r_{ij}(n))$ converge (as occurs in the Oz Weather and urns examples in section 12.3).

Clearly, if a chain contains two or more recurrent classes then the convergence to steady-state values will depend on the initial state – the possibility of visiting S_j depends on whether it's in the same recurrent class as the current state. We'll therefore restrict our attention to chains with a single recurrent class, possibly with additional transient states. This isn't as restrictive as it may seem, since the behavior of chains with multiple recurrent classes can be analyzed by considering each class separately.

We also want to avoid periodicity since it makes the possibility of visiting a state S_j depend on whether it's located some multiple of the chain's periodicity d away from the current state.

If we assume that our Markov chain has a single recurrent class, and is also aperiodic, then its states S_j have steady-state probabilities π_j with the following properties:

(a) For each S_j, we can express the convergence of the n-step transitions as

$$\lim_{n\to\infty} r_{ij}(n) = \pi_j, \text{ for all } i.$$

(b) The π_j values are unique solutions to the following system of equations:

$$\begin{aligned} \pi_j &= \sum_{k=1}^{m} \pi_k p_{kj}, \quad j = 1, ..., m, \\ 1 &= \sum_{k=1}^{m} \pi_k. \end{aligned}$$

The equations defined by the first line are called *balance equations*, and the last equality is a *normalization equation*.

(c) We have:

$$\begin{aligned} \pi_j &= 0 \quad \text{for all transient states } j, \\ \pi_j &> 0 \quad \text{for all recurrent states } j. \end{aligned}$$

The balance and normalization equations can be solved to obtain the π_j steady-state probabilities for a chain, as illustrated in the following examples.

12.5.1 Weather in Oz. The OzWeather chain (see Fig. 12.1) has a single aperiodic recurrent class, so converges to a steady state. The balance equations are obtained by defining π_j using each row of the transition probability matrix. The result:

$$\pi_R = \frac{1}{2}\pi_N + \frac{1}{4}\pi_S + \frac{1}{2}\pi_R \tag{12.1}$$

$$\pi_N = \frac{1}{4}\pi_R + \frac{1}{4}\pi_S \tag{12.2}$$

$$\pi_S = \frac{1}{2}\pi_N + \frac{1}{4}\pi_R + \frac{1}{2}\pi_S \tag{12.3}$$

Simplifying π_R in 12.1:

$$\pi_R = \pi_N + \frac{1}{2}\pi_S \tag{12.4}$$

Substitute 12.2 into 12.4 to remove π_N:

$$\begin{aligned} \pi_R &= \frac{1}{4}\pi_R + \frac{3}{4}\pi_S \\ \pi_R &= \pi_S \end{aligned} \tag{12.5}$$

Substitute 12.5 into 12.2 to remove π_R:

$$\begin{aligned} \pi_N &= \frac{1}{4}\pi_S + \frac{1}{4}\pi_S \\ &= \frac{1}{2}\pi_S \end{aligned} \tag{12.6}$$

We now substitute equations 12.5 and 12.6 into the Normalization equation ($\pi_S + \pi_R + \pi_N = 1$) to obtain $\pi_S = 2/5$, and so $\pi_R = 2/5$ and $\pi_N = 1/5$. Note that these steady-state probabilities correspond to the results of calling `powersTest.py` on the OzWeather matrix in section 12.2.

12.6 The Two Urns

In the general case when there are N balls in the two urns, and a probability of q of moving a ball from urn A to B, then the balance equation for n balls in A is:

$$\begin{aligned}\pi_n &= \frac{N-(n-1)}{N}q\pi_{n-1} + (1-q)\pi_n + \frac{n+1}{N}q\pi_{n+1} \\ &= \frac{N-(n-1)}{N}\pi_{n-1} + \frac{n+1}{N}\pi_{n+1}\end{aligned}$$

The easiest way to see this is by considering the three arcs entering state S_n (the n node) in Fig. 12.2 from states S_{n-1}, S_n, and S_{n+1}. The probabilities on these arcs are assigned to π_{n-1}, π_n, and π_{n+1}.

Note that after simplification, π_n is independent of q. Intuitively, q governs how "quickly" we transfer balls from one urn to the other. Therefore, it should affect the speed at which the system reaches a steady state, but not the values in that state.

The balance equations becomes a little clearer if we let $p_n = n/N$ and $q_n = 1 - n/N$ (which is $(N-n)/N$). This allows us to write:

$$\pi_n = q_{n-1}\pi_{n-1} + p_{n+1}\pi_{n+1}$$

The π_i equations become:

$$\begin{aligned}\pi_0 &= p_1\pi_1 \\ \pi_1 &= q_0\pi_0 + p_2\pi_2 \\ \pi_2 &= q_1\pi_1 + p_3\pi_3 \\ &\vdots \\ \pi_N &= q_{N-1}\pi_{N-1}\end{aligned}$$

Induction shows that π_n can be rewritten nonrecursively as:

$$\pi_n = \frac{q_{n-1}q_{n-2}\cdots q_0}{p_n p_{n-1}\cdots p_1}\pi_0$$

So:

$$\begin{aligned}\pi_n &= \frac{(N-(n-1))(N-(n-2))\cdots N}{n(n-1)\cdots 1}\pi_0 \\ &= \frac{N!}{n!(N-n)!}\pi_0 \\ &= \binom{N}{n}\pi_0\end{aligned}$$

Combining this and the Normalization equation ($\pi_0 + \pi_1 + \cdots + \pi_N = 1$) implies that $\pi_0 = \frac{1}{2^N}$, and so:

$$\pi_n = \binom{N}{n}2^{-N}$$

We can check this against specific cases of the urns problem, such as the $N = 5$ run of `urnsN.py`. The steady state vector Π is:

$$\begin{matrix} & 0 & 1 & 2 & 3 & 4 & 5 \end{matrix}$$
$$\mathbf{\Pi} = \begin{bmatrix} 1/32 & 5/32 & 10/32 & 10/32 & 5/32 & 1/32 \end{bmatrix}$$

This tallies with the output from PowersTest.py at the end of section 12.2.

12.7 Absorption

Another common use of Markov chains is to determine short-term behavior, such as the first recurrent state to be entered, and when that occurs. This information is frequently required when a state is *absorbing*, which means that the state can never be left once entered. An equivalent definition of an absorbing state is one which has a transition back to itself with probability 1; i.e.,

$$p_{kk} = 1, \qquad p_{kj} = 0 \text{ for all } j \neq k.$$

Such a transition may occur naturally in a Markov chain (as in the examples below). Another situation is when we rewrite a recurrent class consisting of several states as a single absorbing state. This captures the fact that a recurrent class is never left once entered, and has the side-benefit of simplifying the analysis of the chain.

Typical questions for such chains include:

(a) What is the probability that the system will eventually reach an absorbing state?

(b) What is the probability that the system will end in a particular absorbing state?

(c) On average, how many time steps will it take for the system to reach an absorbed state?

(d) On average, how long will the system be in each transient state before being absorbed?

12.7.1 Example: The Drunkard's Walk. A man walks along a stretch of a city street with six 'stops' marked as '0' to '5' (see Fig. 12.7).

If the drunk is at stop 1, 2, 3, or 4, then he walks to the left or right with equal probability. He continues until he reaches stop 5, which is a bar, or stop 0, which is his home. If he reaches either home or the bar, he stays there. In other words, 0 and 5 are absorbing states and 1, 2, 3, and 4 are transient. The transition matrix

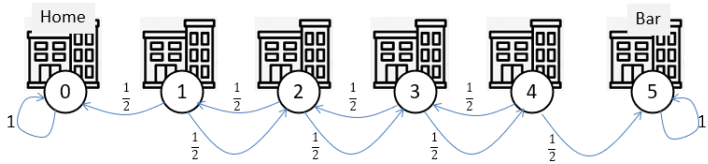


Figure 12.7. The Drunkard's Walk with 6 Stops

in drunk6.txt is:

$$\mathbf{P} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

12.7.2 Example: Searching a Maze. The maze consist of nine rooms with connections as indicated in Fig. 12.8.

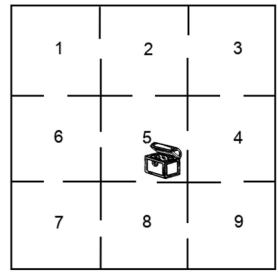


Figure 12.8. Searching a Maze for Treasure

The explorer is searching for room 5 which holds a prize, and chooses a doorway in the current room with equal probability. The transition matrix is in maze.txt:

```
> python display.py
fnm: maze
Reading: maze.txt
P:
        1       2       3       4       5       6       7       8       9
1        0.00    0.50    0.00    0.00    0.00    0.50    0.00    0.00    0.00
2        0.33    0.00    0.33    0.00    0.33    0.00    0.00    0.00    0.00
```

```
3       0.00    0.50    0.00    0.50    0.00    0.00    0.00    0.00    0.00
4       0.00    0.00    0.33    0.00    0.33    0.00    0.00    0.00    0.33
5       0.00    0.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00
6       0.33    0.00    0.00    0.00    0.33    0.00    0.33    0.00    0.00
7       0.00    0.00    0.00    0.00    0.00    0.50    0.00    0.50    0.00
8       0.00    0.00    0.00    0.00    0.33    0.00    0.33    0.00    0.33
9       0.00    0.00    0.00    0.50    0.00    0.00    0.00    0.50    0.00
```

Room 5 is the absorbing state, as confirmed by its self-transition probability of 1.

12.8 Finding Absorbing Probabilities

It's possible to calculate the probability a_i of eventually reaching an absorbing state s, starting from state i, by formulating recursive equations:

$$\begin{aligned} a_s &= 1, \\ a_i &= 0 \quad \text{for all absorbing states } i \neq s, \\ a_i &= \textstyle\sum_{j=1}^{m} p_{ij} a_j, \quad \text{for all transient states } i. \end{aligned}$$

In a similar way, the expected times to absorption, $\mu_1, ..., \mu_m$, are:

$$\begin{aligned} \mu_i &= 0 \quad \text{for all recurrent states } i \\ \mu_i &= 1 + \textstyle\sum_{j=1}^{m} p_{ij}\mu_j \quad \text{for all transient states } i. \end{aligned}$$

An alternative approach involves reorganizing the transition matrix into a *canonical* form which can be manipulated using matrix operations to find these probabilities and times.

The canonical form is obtained by rearranging the transition matrix's states so that the transient states come first. If there are r absorbing states and t transient states, the matrix becomes:

$$\mathbf{P} = \begin{array}{c} \\ \text{TR} \\ \text{ABS.} \end{array} \begin{array}{c} \begin{array}{cc} \text{TR.} & \text{ABS.} \end{array} \\ \left(\begin{array}{c|c} Q & R \\ \hline \mathbf{0} & \mathbf{I} \end{array}\right) \end{array}$$

$\mathbf{I}$ is an r-by-r identity matrix, $\mathbf{0}$ is an r-by-t zero matrix, R is a nonzero t-by-r matrix, and Q is a t-by-t matrix. The first t states are transient and the last r states are absorbing.

The entry p_{ij}^n of the nth power of the rearranged matrix P is the probability of being in the state j after n steps, when the chain starts in state i. A standard matrix algebra argument shows that P^n will have the form:

$$\mathbf{P^n} = \begin{array}{c} \\ \text{TR.} \\ \text{ABS.} \end{array} \begin{array}{c} \begin{array}{cc} \text{TR.} & \text{ABS.} \end{array} \\ \left(\begin{array}{c|c} Q^n & * \\ \hline \mathbf{0} & \mathbf{I} \end{array}\right) \end{array}$$

The asterisk * stands for the t-by-r matrix in the upper right-hand corner which can be written in terms of Q and R, but isn't needed here.

The entries in Q^n give the probabilities for being in each of the transient states after n steps for each possible transient starting state. Since the probability of being in a transient state after n steps approaches zero, every entry in Q^n will tend to zero as n approaches infinity.

For an absorbing Markov chain, the inverse of the matrix $I - Q$ is

$$\begin{aligned} N &= (I-Q)^{-1} \\ &= I + Q + Q^2 + \cdots . \end{aligned}$$

This states that the ij-entry n_{ij} of N is the expected number of times the chain is in state j, given that it starts in state i. This can be proved by letting $(I-Q)x = 0$; that is $x = Qx$. By iterating we see that $x = Q^n x$. However, $Q^n \to 0$, so $Q^n x \to 0$, and so $x = 0$ since $x = Qx$. A fundamental result of linear algebra is that a matrix M has an inverse if and only if $Mx = 0$ implies $x = 0$, which means that $(I-Q)^{-1}$ exists. Next, note that:

$$(I—Q)(I + Q + Q^2 + \cdots + Q^n) = I - Q^{n+1}.$$

Multiplying both sides by N gives

$$I + Q + Q^2 + \cdots + Q^n = N(I - Q^{n+1}).$$

Letting n go to infinity produces:

$$N = I + Q + Q^2 + \cdots .$$

N is called the *Fundamental Matrix.*

12.8.1 A Canonical Drunkard's Walk. For the Drunkard's walk example (Fig. 12.7), the transition matrix in canonical form is:

$$\mathbf{P} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ \hline 0 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \left(\begin{array}{cccc|cc} 0 & 1/2 & 0 & 0 & 1/2 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 0 & 1/2 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) \end{array} \begin{array}{c} \\ \\ \\ \\ \\ \begin{array}{cc} 0 & 5 \end{array} \end{array}$$

So

$$\mathbf{Q} = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 1/2 & 0 \end{bmatrix}$$

and

$$\mathbf{I} - \mathbf{Q} = \begin{bmatrix} 1 & -1/2 & 0 & 0 \\ -1/2 & 1 & -1/2 & 0 \\ 0 & -1/2 & 1 & -1/2 \\ 0 & 0 & -1/2 & 1 \end{bmatrix}$$

Computing $(I - Q)^{-1}$ with Mat.py:

```
from Mat import *

Q = Mat([[0,   0.5, 0,   0],
         [0.5, 0,   0.5, 0],
         [0,   0.5, 0,   0.5],
         [0,   0,   0.5, 0] ])
N = (Mat.identity(4)-Q).inverse()   # 4-by-4 identity
print(N)
```

The result:

$$\mathbf{N} = (\mathbf{I} - \mathbf{Q})^{-1} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1.60 & 1.20 & 0.80 & 0.40 \\ 1.20 & 2.40 & 1.60 & 0.80 \\ 0.80 & 1.60 & 2.40 & 1.20 \\ 0.40 & 0.80 & 1.20 & 1.60 \end{pmatrix} \end{array}$$

From the second row of N, we see that if the system starts in state 2, then the expected number of times it returns to states 1, 2, 3, and 4 before being absorbed are 1.2, 2.4, 1.6, and 0.8.

12.9 Absorption Time and Probabilities

Given that a chain starts in state i, what is the expected number of steps t before the chain is absorbed? It can be expressed as:

$$t = Nc,$$

where c is a column vector whose entries are all 1's. The effect of the matrix multiplication is to sum the entries in the ith row of N, to get the total expected number of times state i visits the transient states.

Let b_{ij} be the probability that the system reaches absorbing state j if it starts in the transient state i. B, containing the b_{ij} entries, is generated using:

$$B = NR,$$

R is the nonzero t-by-r matrix from the canonical matrix. Each b_{ij} will be the sum of all the transitions through the transient states in Q ending in transient state k, followed by a transition to the absorbing state j with probability r_{kj}. This is:

$$\begin{aligned} b_{ij} &= \sum_n \sum_k q_{ik}^n r_{kj} \\ &= \sum_k \sum_n q_{ik}^n r_{kj} \\ &= \sum_k n_{ik} r_{kj} \\ &= (NR)_{ij}. \end{aligned}$$

12.9.1 Analyzing the Drunkard's Walk. In the six-stop Drunkard's walk,

$$\mathbf{N} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1.60 & 1.20 & 0.80 & 0.40 \\ 1.20 & 2.40 & 1.60 & 0.80 \\ 0.80 & 1.60 & 2.40 & 1.20 \\ 0.40 & 0.80 & 1.20 & 1.60 \end{pmatrix} \end{array}$$

The expected number of time steps, t, before absorption:

$$\mathbf{t} = \mathbf{Nc} = \begin{bmatrix} 1.60 & 1.20 & 0.80 & 0.40 \\ 1.20 & 2.40 & 1.60 & 0.80 \\ 0.80 & 1.60 & 2.40 & 1.20 \\ 0.40 & 0.80 & 1.20 & 1.60 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 4.00 \\ 6.00 \\ 6.00 \\ 4.00 \end{bmatrix}.$$

Thus, starting in states 1, 2, 3, and 4, the expected times to absorption are 4, 6, 6, and 4, respectively. This reflects the fact that the closer the transient state is to an absorbing state, the less time it takes to be absorbed.

The probabilities of absorption:

$$\begin{aligned} \mathbf{B} = \mathbf{NR} &= \begin{bmatrix} 1.60 & 1.20 & 0.80 & 0.40 \\ 1.20 & 2.40 & 1.60 & 0.80 \\ 0.80 & 1.60 & 2.40 & 1.20 \\ 0.40 & 0.80 & 1.20 & 1.60 \end{bmatrix} \begin{bmatrix} 0.5 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0.5 \end{bmatrix} \\ &= \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cc} 0 & 5 \end{array} \\ \begin{pmatrix} 0.80 & 0.20 \\ 0.60 & 0.40 \\ 0.40 & 0.60 \\ 0.20 & 0.80 \end{pmatrix} \end{array} \end{aligned}$$

The shows that the probability of being absorbed into states 0 or 5 depends on how close the transient state is to one of them. The spread of probabilities in each column mirror each other since the drunk has equal chance of walking to the left or right.

12.10 Implementing Absorption

The absorb() function in MarkovLib.py carries out all of the calculations detailed in the previous section. It is passed the Q and R subarrays of the canonical matrix, and returns the fundamental matrix N, the absorption probabilities matrix B, and a vector t holding the expected times to absorption.

```
def absorb(Q, R):
  n = Q.nRows
  N = (Mat.identity(n)- Q).inverse()
  t = N*Mat.fill(n, 1, 1)  # column vector of 1's
```

```
  B = N*R
  return N, B, t
```

The values reported in the previous section for the Drunkard's Walk were obtained by passing drunk6.txt to absorbTests.py. It reads in the matrix, rearranges it to obtain the Q and R sub-matrices, calls absorb(), and prints the results:

```
fnm = input("fnm? ")
pLabels, P = readMarkov(fnm)
Q, qLabels, R, rLabels = getQR(P, pLabels)
N, B, t = absorb(Q, R)
```

getQR() makes the assumption that the matrix passed to it has absorbing states in the first and last row.

12.11 Implementing Steady State Using Matrices

Although section 12.5 solved balance equations with a normalization equation, it's also possible to obtain solutions using a matrix approach similar to that used in absorption problems.

The idea of a fundamental matrix (Z^*) for steady-state problems was first proposed by Kemeny and Snell in the first edition of their text *Finite Markov Chains* [**KS83**], where they defined:

$$Z^* = (I - P + e\pi)^{-1}$$

P is the transition probability matrix, e a column vector of ones, and π a row vector holding the steady state distribution.

Many properties of the Markov chain, such as the mean passage time and its variance, can be found using this fundamental matrix. The drawback is the cost of calculating π by solving n equations. This was addressed in a subsequent paper by Kemeny, which appeared as an appendix in later editions of the textbook. The matrix became:

$$Z_\beta = (I - P + e\beta)^{-1},$$

where β is any row vector such that $\beta \cdot e = 1$. Also, the steady-state vector π becomes much faster to calculate since $\pi = \beta \cdot Z_\beta$.

This approach is implemented in the getZPi() function in MarkovLib.py:

```
def getZPi(P):
  n = P.nRows
  e = Mat.fill(n, 1, 1)  # column vector, all 1's
  beta = Mat.fill(1, n, 0)  # row vector
  beta[0][0] = 1  # so that beta * e == 1

  X = Mat.identity(n) - P + (e*beta)
  Z = X.inverse()
  pi = beta*Z
  return Z, pi.row(0)  # return pi as a list
```

12.11.1 Using the Steady-State Matrix. If a recurrent Markov chain begins in state i, the expected number of steps to reach state j for the first time is called the *mean first passage time* from states i to j. It's denoted by m_{ij}, with the convention that $m_{ii} = 0$.

A quantity closely related to the mean first passage time is the *mean recurrence time*, r_i. Assume that we start in state i, and calculate the length of time before we return to i for the first time. We either stay in i or go to some other state j, and from there we'll eventually return to i because the chain is recurrent.

The mean first passage time from states i to j can be calculated recursively. If the first step is to state j, the expected number of steps is 1; if it goes to some other state k, the expected number of steps is m_{kj}, plus 1 for the step just taken. In addition, these steps must be multiplied by their probabilities of occurring. Thus,

$$m_{ij} = p_{ij} + \sum_{k \neq j} p_{ik}(m_{kj} + 1)$$

Since $\sum_k p_{ik} = 1$,

$$m_{ij} = 1 + \sum_{k \neq j} p_{ik} m_{kj}.$$

The mean recurrence times are obtained in a similar way but by considering the steps needed to return to state i:

$$\begin{aligned} r_i &= \textstyle\sum_k p_{ik}(m_{ki} + 1) \\ &= 1 + \textstyle\sum_k p_{ik} m_{ki}. \end{aligned}$$

If we have the fundamental matrix Z_β and steady-state vector π (which are both returned by getZPi() in MarkovLib.py), then a more convenient way to determine these values is by using:

$$\begin{aligned} m_{ij} &= \frac{z_{jj} - z_{ij}}{\pi_j} \\ r_i &= 1/\pi_i \end{aligned}$$

This approach is utilized in steady() in MarkovLib.py:

```
def steady(P):
  Z, pi = getZPi(P)
  n = P.nRows
  M = Mat.fillSq(n, 0)     # M = mean first passage times
  r = [0]*n     # r = mean recurrence times
  for i in range(n):
    for j in range(n):
      if pi[j] < EPS:   # ignore transient states
        M[i][j] = 0
      else:
        M[i][j] = (Z[j][j] - Z[i][j]) / pi[j]
    if pi[i] < EPS:  # ignore transient states
```

```
      r[i] = 0
    else:
      r[i] = 1 / pi[i]
  return Z, pi, r, M
```

12.11.2 Oz Weather Again. steadyTests.py reads in a Markov chain, calls steady(), and prints the various generated matrices:

```
> python steadyTests.py
fnm: OzWeather
Reading: OzWeather.txt
P:
        R       N       S
R        0.50    0.25    0.25
N        0.50    0.00    0.50
S        0.25    0.25    0.50

Fundamental matrix Z:
        R       N       S
R        0.93   -0.20   -0.40
N       -0.13    0.60   -0.13
S       -0.40   -0.20    0.93

Stationary Distribution pi:
        R       N       S
         0.4000  0.2000  0.4000

Mean Recurrence Times r:
        R       N       S
         2.50    5.00    2.50

Mean First Passage Times M:
        R       N       S
R        0.00    4.00    3.33
N        2.67    0.00    2.67
S        3.33    4.00    0.00
```

The M and r data reveal that the first nice day is a long time coming (4 days) and reoccurs infrequently (every 5 days).

Exercises

(1) A car insurance company groups its customers into three categories: 'poor', 'satisfactory', and 'preferred'. No one moves from 'poor' to 'preferred', or from 'preferred' to 'poor' in a single year. However, 40% of the customers in the 'poor' category become 'satisfactory', 30% of those in the 'satisfactory' category

moves to 'preferred', while 10% become 'poor'; 20% of those in the 'preferred' category are downgraded to 'satisfactory'.

(a) Write the transition matrix for the model.

(b) What is the steady-state fraction of car owners in each of these categories?

(2) In his book, *Wahrscheinlichkeitsrechnung und Statistik* [**Eng76**], the first author proposed an algorithm for finding the steady state for a Markov chain when the transition probabilities are rationals.

For each state i, let a_i be the least common multiple of the denominators of the non-zero entries in the ith row. The algorithm operates in terms of moving chips around on the states – indeed, for small examples, the algorithm can be implemented this way.

Start by putting a_i chips on state i for all i. Then, at each state, redistribute the a_i, chips, sending $a_i p_{ij}$ to state j. For each state i, add just enough chips to bring the number of chips at state i up to a multiple of a_i. Then redistribute the chips in the same manner.

This process will eventually reach a point where the number of chips at each state, after the redistribution, is the same as before redistribution. At this point, we have found a steady state. Here's an example:

$$\mathbf{P} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/2 & 1/4 & 1/4 \end{pmatrix} \end{array}$$

We start with $a_1 = (4, 2, 4)$. The chips after successive redistributions are:

```
(4 2 4)  (5 2 3)   (8 2 4)   (7 3 4)   (8 4 4)  (8 3 5) (8 4 8)
(10 4 6) (12 4 8)  (12 5 7)  (12 6 8)  (13 5 8) (16 6 8)
(15 6 9) (16 6 12) (17 7 10) (20 8 12) (20 8 12).
```

$a = (20, 8, 12)$ is a steady state.

Write a program to implement this algorithm.

(3) A highly simplified game of "Monopoly" is played on a board with four squares shown in Fig. 12.9. You start at "GO". You roll a die and move clockwise around the board a number of squares equal to the number that turns up on the die. You collect or pay an amount indicated on the square on which you land. You then roll the die again and move around the board in the same manner from your last position.

Calculate the amount you should expect to win in the long run playing this version of Monopoly.

-5 B	20 C
-30 A	15 GO

Figure 12.9. Simplified "Monopoly"

(4) A professor gives tests that are hard, medium, or easy. If she gives a hard test, her next test will be either medium or easy, with equal probability. However, if she gives a medium or easy test, there is a 0.5 probability that her next test will be of the same difficulty, and a 0.25 probability for each of the two other difficulty levels. Construct an appropriate Markov chain, and find the steady-state probabilities.

(5) SQSnakes and Ladders is played on a board of nine squares shown in Fig. 12.10.

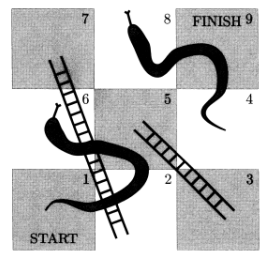


Figure 12.10. Small "Snakes and Ladders"

On each turn, a player tosses a fair coin and advances one or two places according to whether the coin lands heads or tails. If you land at the foot of a ladder you climb to the top, but if you land on the head of a snake you slide down its tail.

(a) How many turns on average does it take to complete the game?
(b) What is the probability that a player who has reached the middle square will complete the game without slipping back to square 1?

(6) A fair coin is tossed repeatedly and independently. Find the expected number of tosses required until the pattern HTHH appears.

(7) In tennis the winner of a game is the first player to win four points, unless the score is 4-3, in which case the game must continue until one player wins by having two more points than the other player.

Suppose that the game has reached the stage where one player is trying to get two points ahead to win, and that the server will independently win the point with probability 0.6.

What is the probability that the server will win the game if the score is currently tied at 3-3?; if she is ahead by one point?; behind by one point?

(8) Mary and John have a three-card deck marked with the numbers 1, 2, and 3, and a spinner with the numbers 1, 2, and 3 written on it. The game begins by dealing the cards out so that the dealer gets one card and the other person gets two. A move in the game consists of a spin of the spinner. The person having the card with the number that comes up on the spinner gives that card to the other person. The game ends when someone has all the cards.

(a) Set up the transition matrix for this absorbing Markov chain. Have the states correspond to the number of cards that Mary has.

(b) Calculate the fundamental matrix.

(c) On average, how many moves will the game last for?

(d) If Mary deals, what is the probability that John will win the game?

(9) A deck of cards has 3 red and 3 blue cards. At each stage, a card is selected at random. If it's red, it's taken from the deck and discarded. If it's blue then the card stays in the deck, and the game continues.

Find the average number of steps till there are no red cards left in the deck.

13

Navigating the Earth

Why Study Spherical Trigonometry?

- **It is the natural geometry of the Earth**. Distances, bearings, angles, and areas on a sphere follow spherical, not planar, relations. Any problem involving global position, long distances, or directions across Earth must use spherical trigonometry for accuracy. Modern GPS and mapping still rely on spherical or ellipsoidal formulae.

- **It is central to astronomy**. The positions of stars, planets, and satellites are mapped onto a celestial sphere. Converting between coordinate systems (equatorial, ecliptic, galactic), computing angular separations, and predicting rising/setting all use spherical trigonometry.

13.1 Introduction

Spherical trigonometry is the branch of geometry dealing with triangles drawn on the surface of a sphere, a topic of great importance in astronomy and navigation, but often ignored in modern textbooks on trigonometry. This is a shame since it lends itself to practical examples especially if you're an amateur star-gazer or sailor, and offers a fresh perspective that reveals hidden commonalities with Euclidean geometry.

We'll be focusing on the use of spherical trigonometry for navigating over the Earth, but if you'd like a wider discussion that includes its astronomical origins, we strongly recommend *Heavenly Mathematics* by Glen Van Brummelen [**VB13**].

If you'd prefer something a bit shorter, try chapter 7 of the excellent *Mathematics for the Million* by Lancelot Hogben [**Hog93**] and/or chapter 13 of Eli Maor's *Trigonometric Delights* [**Mao20**].

13.2 Properties of the Sphere

The location of a point P on a sphere is specified in terms of *latitude* ϕ and *longitude* λ (see Fig. 13.1). These are defined with respect to a reference point M on the equatorial *great circle*. A great circle is any circular intersection of a sphere and a plane passing through the sphere's center O.

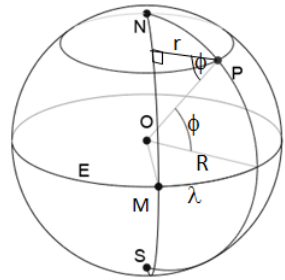


Figure 13.1. Locating a Point P

Given any point P, different from the poles (N and S), there's a unique great circle through P and the poles, and the half containing P is termed its *meridian*. The angle ϕ at the sphere's center between P and the equator is its latitude, and considered to be positive in the northern hemisphere and negative in the south. The meridian through M is the *prime meridian*.

A set of points with the same latitude is a *parallel*, and forms a small spherical circle over the surface centered on one of the poles. If the radius of the latitude circle is r and the equator's radius is R then $r = R\cos\phi$.

The angle λ around the equator from M to P's meridian is its longitude, which is positive to the east of M and negative to the west.

On the Earth, the prime meridian passes through Greenwich, England (see Fig. 13.2), placing New Orleans at 29° 57' 16.74" N and -90° 4' 30.252"' E, or (29.95465, -90.07507) as a decimal tuple. Of course, the Earth isn't a sphere, it's an ellipsoid or oblate spheroid, but we'll be ignoring that distinction since it simplifies the maths (at the expense of some accuracy).

The meridian located at 180°E is the same as that at 180°W, and runs almost entirely through the Pacific Ocean, coinciding for the most part with the international date line.

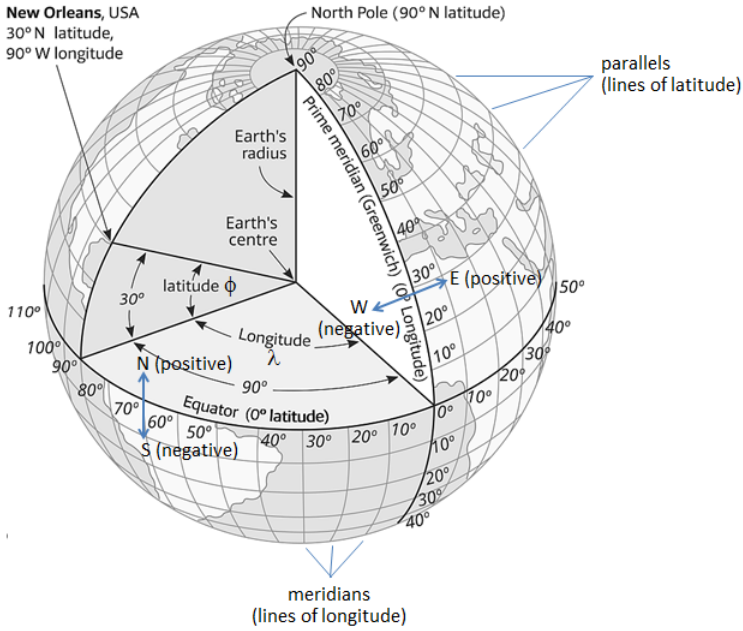


Figure 13.2. Locating a Point on Earth

13.3 Curves on a Sphere

There are two notable types of curves running over a sphere. We've already mentioned the great circle (Fig. 13.3(a)) which forms the shortest arc between any two points on its circle. This is obviously useful in navigation when we need to get from one point to another in the shortest time.

The second curve is the *rhumb line*, or loxodrome, which intersects all the meridians at the same angle (see Fig. 13.3(b)). Compass navigation is particularly simple along a rhumb line since the bearing never changes (e.g. it's θ at A, B, C, and Z). By contrast, the bearing for a great circle usually keeps changing as a journey progresses.

13.4 Mapping a Sphere

Euler proved in 1775 that it wasn't possible to map even a part of a sphere onto the plane without introducing some distortion. Nevertheless, cartographers had to try, and one way to classify their (many) attempts is based on the spherical properties that are preserved. Some of the more common ones are:

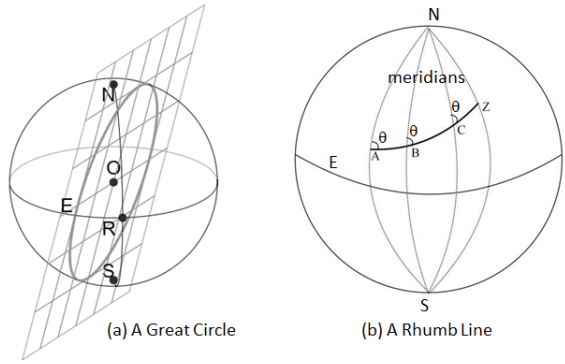


Figure 13.3. Curves Over a Sphere

- Preserving direction. This is only possible with respect to one or two chosen points going to every other point on the sphere, and the resulting map will vary significantly depending on the choice. One familiar example is the orthographic projection in Fig. 13.4.

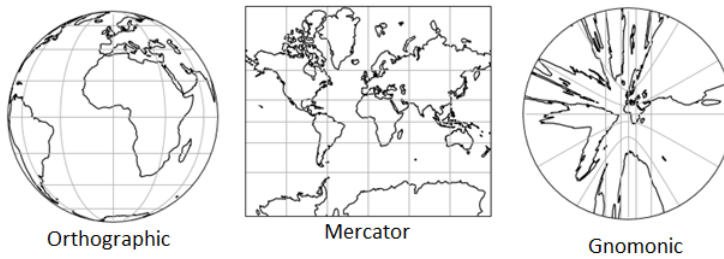


Figure 13.4. Orthographic, Mercator, and Gnomonic Projections

- Preserving Angle. *Conformal* projections preserve angles but not necessarily lengths, even though over small distances the shapes are fairly similar to those on the sphere. The Mercator map (Fig. 13.4) is the most popular, and has the significant benefit of allowing rhumb lines to be drawn as straight lines.
- Preserving area. An example is the Lambert cylindrical equal-area projection (see Fig. 13.5).

  Each non-polar point on the sphere is mapped to a point on a cylinder wrapped around its equator. As depicted in Fig. 13.6, a line is drawn horizontally through a given point on the sphere until it intersects the cylinder.

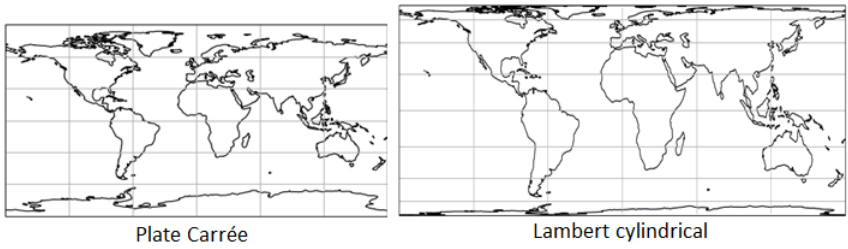


Figure 13.5. Plate Carrée and Lambert Cylindrical Projections

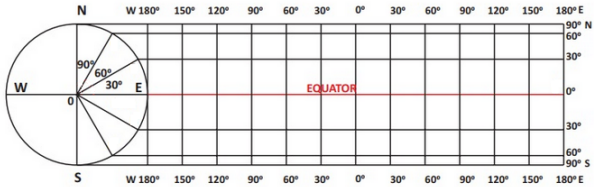


Figure 13.6. Lambert Cylindrical Equal Area Projection

- Preserving distance (equidistant). As with direction preserving projections, the mapping only works for a chosen point (or two) going to every other point. The most common map of this type is the Plate Carrée projection (see Fig. 13.5).
- Preserving the shortest route. The gnomonic projection (Fig. 13.4) allows great circles to be drawn as straight lines, but significantly distorts the surrounding map.

The examples in Figs. 13.4 and 13.5 were drawn with Cartopy (`https://scitools.org.uk/cartopy/docs/latest/`), a Python module that extends Matplotlib. We'll be using it throughout this section to illustrate the maths behind great circles and rhumb lines.

13.5 Spherical Triangles

Fig. 13.7 shows a spherical triangle ABC formed by the intersection of three great circles with O at the sphere's center. (We've borrowed this figure and the associated maths from Hogben [**Hog93**].)

The angles at A, B, and C are between the triangles's edges on the surface of the sphere, while the lengths (a, b, and c) can be viewed as the angles between the radial lines OA, OB, and OC. If the sphere has unit radius (as assumed in the right hand image in Fig. 13.7), then the side lengths are equal to the angles at O.

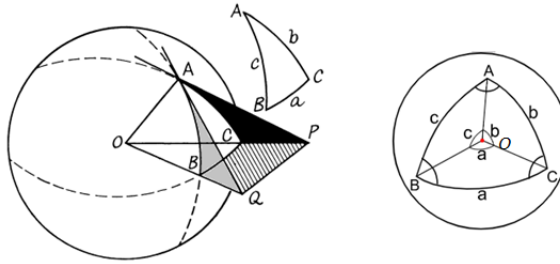


Figure 13.7. A Spherical Triangle ABC

In the left-hand image, we've extended OB and OC to OBQ and OCP such that AQ and AP are tangents to the great circles with arcs AB and AC. This means that OAQ and OAP are right angles, although it's hard to draw them as such.

The result is a triangular pyramid $OAPQ$, which is unfolded in Fig. 13.8 around the point A. Fig. 13.8 is a view of the resulting four triangles from above with OQP as the base of the pyramid.

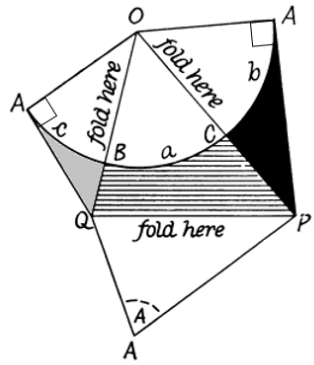


Figure 13.8. The Unfolded Pyramid $OAPQ$

The triangles can be related using Euclidean trigonometry, specifically its sine law:

$$PQ^2 = PO^2 + QO^2 - 2PO \cdot QO \cos a$$
$$PQ^2 = PA^2 + QA^2 - 2PA \cdot QA \cos A$$

Subtracting these equations:

$$(PO^2 - PA^2) + (QO^2 - QA^2) - 2PO \cdot QO \cos a + 2PA \cdot QA \cos A = 0$$

Utilizing Pythagoras, and some rearranging:

$$2PO \cdot QO \cos a = 2AO^2 + 2PA \cdot QA \cos A$$

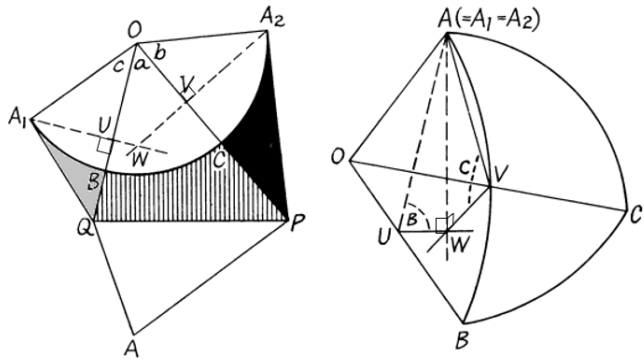
Divide through by $2PO \cdot QO$:

$$\begin{aligned} \cos a &= \frac{AO}{PO} \cdot \frac{AO}{QO} + \frac{PA}{PO} \cdot \frac{QA}{QO} \cos A \\ &= \cos POA \cos QOA + \sin POA \sin QOA \cos A \\ &= \cos b \cos c + \sin b \sin c \cos A \end{aligned}$$

This formula (the *spherical* cosine law) allows us to calculate a third side a, when we know the other two (b and c) and the included angle A. Similar reasoning can be used to provide expressions for calculating the other two sides.

13.6 The Spherical Sine Law

We can employ Fig. 13.8 to demonstrate another important spherical trigonometric formula. The four triangles are drawn in Fig. 13.9(a) with the versions of A named separately, and with a few new coordinates.



(a) Unfolded Pyramid OAPQ (b) Refolded Pyramid

Figure 13.9. The Unfolded and Refolded Pyramid

As the pyramid model is unfolded, A_1 rotates around OQ, and A_2 describes a similar arc about OP. Upon reaching the plane,

$$A_1UO = 90° = A_2VO$$

When the triangles are folded back into a pyramid in Fig. 13.9(b), A will be directly above W, and from Fig. 13.9(a) we see that:

$$\frac{A_1U}{A_1O} = \sin c, \quad \text{and} \quad \frac{A_2V}{A_2O} = \sin b$$

and since $A_1O = A_2O = AO$:

$$A_1U = AO \sin c \quad \text{and} \quad A_2V = AO \sin b$$

In Fig. 13.9(b), AUW is the angle between the planes that include the AB and BC arcs, which means that AUW equals the angle B of the spherical triangle. And so:

$$\frac{AW}{A_1U} = \sin B, \quad \text{and} \quad \frac{AW}{A_2V} = \sin C$$
$$AW = A_1U \sin B \quad \text{and} \quad AW = A_2V \sin C$$

A combination of these equalities produces:

$$AO \sin c \sin B = AW = AO \sin b \sin C$$

and so:

$$\frac{\sin c}{\sin C} = \frac{\sin b}{\sin B}$$

It's not difficult to show that these ratios are also equal to $\frac{\sin a}{\sin A}$.

13.7 Triangles made from Lunes

The lune in Fig. 13.10 is a slice of the surface of a sphere of radius R cut by two planes through meridians separated by an angle θ. The lune's surface area, $2R^2\theta$, is the total area of the sphere ($4\pi R^2$) multiplied by the angular fraction of the circumference $\theta/2\pi$. The volume of the wedge is $\frac{2}{3}R^3\theta$, which is obtained by multiplying the sphere's volume ($\frac{4}{3}\pi R^3$) by the same fraction.

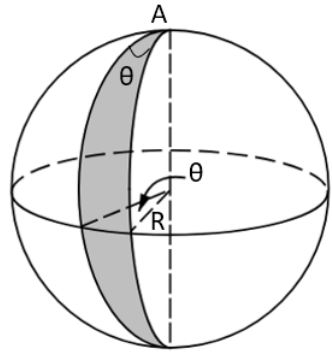


Figure 13.10. A Lune

The two planes that cut the meridians can be extended through the sphere, resulting in the double lune shown in Fig. 13.11. A and A' are located on opposite

sides of the sphere (A' is the *antipodal* point of A), and the shaded area is double that of the original lune, so equals $4R^2\theta$.

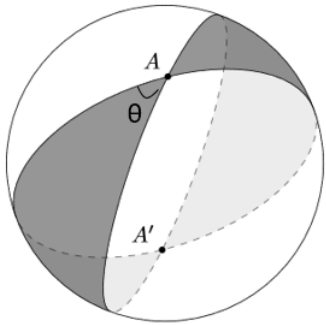


Figure 13.11. A Double Lune

The spherical triangle ABC in Fig. 13.12 is constructed by overlapping three double lunes (D_A, D_B, and D_C) located at A, B, and C.

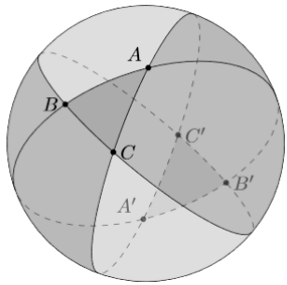


Figure 13.12. A Spherical Triangle made from three Double Lunes

The sum of the areas of the double lunes, denoted by area(D_A) + area(D_B) + area(D_C), is the area of the entire sphere and, due to overlapping, two extra copies of the areas of ABC and $A'B'C'$. The areas of these two triangles are the same, and so:

$$\text{area}(D_A) + \text{area}(D_B) + \text{area}(D_C) = 4\pi R^2 + 4T$$

where T is the area of the spherical triangle ABC. So:

$$4R^2A + 4R^2B + 4R^2C = 4\pi R^2 + 4T$$

Therefore, the area of the spherical triangle is

$$T = R^2(A + B + C - \pi)$$

It is a small step from this to determine that the the sum of the angles $A+B+C$ of *any* spherical triangle is always greater than 180° (or π radians). Specifically: any spherical triangle must occupy some portion of the sphere's surface, and so its area must be positive: $T > 0$. This implies that $A + B + C - \pi > 0$, and so $A + B + C > \pi$.

This property can be utilized to argue for Euclid's theorem – that it's impossible to map any portion of a sphere onto the plane without introducing some distortion. If we decompose the shape into one of more spherical triangles then it's impossible for their angular sum to equal those of the mapped flat triangles.

13.8 Are Great Circle Arcs the Shortest Distance?

Our intuition tells us that a great circle arc between two points is the shortest path, but let's prove it. We'll choose two arbitrary points and show that the great circle connecting them is shorter than every possible other path. This approach is based on one presented in Feeman's *Portraits of the Earth* [**Fee00**].

We'll assume the sphere (see Fig. 13.13) has unit radius and employs an (x, y, z) coordinate system centered around O, and the polar coordinates are (ρ, λ, ψ). ρ is the length of the point's vector, λ corresponds to its longitude, while ψ is its inclination, which is equivalent to $\pi/2 - \phi$ (ϕ is the point's latitude).

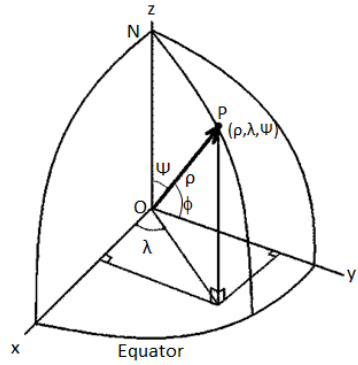


Figure 13.13. 3D Polar and Cartesian Coordinates

The Cartesian coordinates can be expressed as:

$$x = \rho \sin\psi \cos\lambda, \quad y = \rho \sin\psi \sin\lambda \quad z = \rho \cos\psi$$

It's more convenient for our needs to replace ψ by the latitude ϕ:

$$\begin{array}{rclcl} x & = & \rho\sin(\pi/2-\phi)\cos\lambda & = & \rho\cos\phi\cos\lambda \\ y & = & \rho\sin(\pi/2-\phi)\sin\lambda & = & \rho\cos\phi\sin\lambda \\ z & = & \rho\cos(\pi/2-\phi) & = & \rho\sin\phi \end{array}$$

We'll use the north pole as the starting point, which simplifies the calculations. P will have latitude λ and longitude ϕ, and the latitude at the north pole is $\pi/2$ and its longitude is undefined.

The great circle arc from the north pole to P will follow P's meridian at longitude λ. The arc will travel an angle of $\pi/2-\phi$, which is also the meridian arc's length since the sphere has unit radius.

Now its time to consider some other path connecting these same points and show that its length is greater than (or equal to) $\pi/2-\phi$. During the trip over the sphere from the north pole to P, the latitude and longitude keep changing. This can be expressed by two functions, $\phi(t)$ and $\lambda(t)$, that return the current latitude and longitude at a given time t.

The total path length will be computed using the integral of a speed function with respect to the trip duration. This function can be obtained via the derivative of the distance vector.

At time t the latitude is $\phi(t)$ and longitude is $\lambda(t)$. In 3D rectangular coordinates, this means that the position at time t is

$$\begin{array}{rcl} p(t) & = & (x,y,z) \\ & = & (\cos\phi(t)\cos\lambda(t),\ \cos\phi(t)\sin\lambda(t),\ \sin\phi(t)). \end{array}$$

The velocity vector at time t is the coordinate-by-coordinate derivative of the position vector:

$$\begin{array}{rcl} p'(t) & = & (-\sin\phi(t)\dfrac{d\phi}{dt}\cos\lambda(t)-\cos\phi(t)\sin\lambda(t)\dfrac{d\lambda}{dt}, \\ & & -\sin\phi(t)\dfrac{d\phi}{dt}\sin\lambda(t)+\cos\phi(t)\cos\lambda(t)\dfrac{d\lambda}{dt}, \\ & & \cos\phi(t)\dfrac{d\phi}{dt}). \end{array}$$

The speed at time t is the magnitude of the velocity vector, which is the square root of the sum of the squares of the coordinates of the velocity vector. After some refactoring, this simplifies to

$$\|p'(t)\| = \sqrt{\left(\frac{d\phi}{dt}\right)^2 + \cos^2\phi(t)\left(\frac{d\lambda}{dt}\right)^2}$$

We'll assume that the trip from the north pole began at time $t=0$ and concluded at time t_1. Thus, $\phi(t_1)=\phi$, $\lambda(t_1)=\lambda$, and $\phi(0)=\pi/2$. (The value of $\lambda(0)$ isn't

defined, but it doesn't matter.) The total distance travelled is:

$$L = \int_{t=0}^{t_1} \|p'(t)\|\, dt.$$

But:

$$\|p'(t)\| \geq \sqrt{\left(\frac{d\phi}{dt}\right)^2} = \left|\frac{d\phi}{dt}\right| \geq -\frac{d\phi}{dt}.$$

It follows that:

$$L \geq \int_{t=0}^{t_1} -\frac{d\phi}{dt} dt = \phi(0) - \phi(t_1) = \pi/2 - \phi$$

The right-hand side of this inequality is the length of the great circle connecting the north pole and P. Thus, we can conclude that the great circle route is indeed the shortest possible.

13.9 Spherical Triangles Summarized

Table 13.1 summarizes the three main results for spherical triangles, and compares them to their equivalents for Euclidean triangles. The triangles are shown in Fig. 13.14.

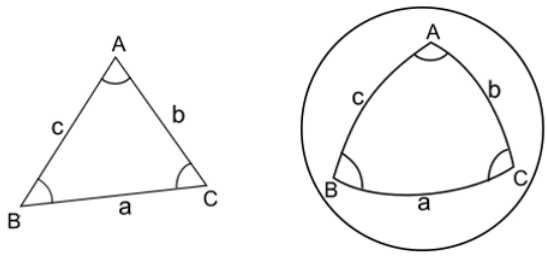


Figure 13.14. Euclidean and Spherical Triangle

13.10 Calculating with Great Circles

Calculations involving great circles typically fall into three areas: finding the great circle distance between two points, determining the initial bearing to travel from one point to another, and obtaining the destination coordinate if only the start point, bearing, and travel distance are supplied.

The best source for code for these calculations is the Movable Type Scripts web page "Calculate distance, bearing and more between Latitude/Longitude

Property	Euclidean Triangle	Spherical Triangle
Law of Cosines	$c^2 = a^2 + b^2 - 2ab\cos C$	$\cos c = \cos a \cos b + \sin a \sin b \cos C$
Law of Sines	$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$	$\frac{\sin a}{\sin A} = \frac{\sin b}{\sin B} = \frac{\sin c}{\sin C}$
Sum of Triangle Angles	$A + B + C = \pi$	$A + B + C > \pi$

Table 13.1. Comparison of Triangle Laws in Euclidean and Spherical Geometries

points" (`https://www.movable-type.co.uk/scripts/latlong.html`), which offers both formulae and JavaScript functions. Although the page assumes the Earth is spherical (as we've done), formulae for calculating distances on ellipsoids is available separately at `https://www.movable-type.co.uk/scripts/latlong-vincenty.html`.

Our Python functions in `nav.py` are closely based on Movable Type Scripts' JavaScript, so the emphasis here will be on explaining how the spherical cosine and sine laws are manipulated to get the formulae used by those functions.

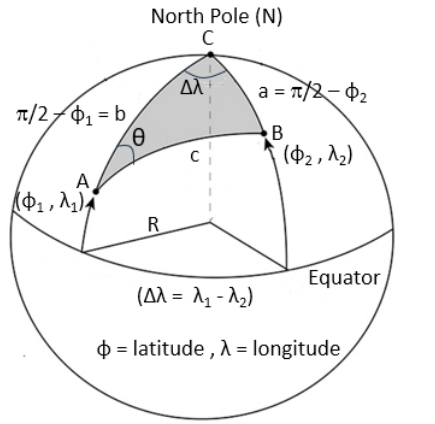


Figure 13.15. A Spherical Triangle for Calculations

Most great circle equations can be derived from the spherical triangle in Fig. 13.15. A is the trip's starting point at latitude ϕ_1, longitude λ_1, with destination B at latitude ϕ_2, longitude λ_2. The third point, C, is located at the North pole

(latitude $\pi/2$) making AC and BC meridian arcs, and the side angles b and a *co-latitudes*, $\pi/2-\phi_1$ and $\pi/2-\phi_2$. We'll treat the sphere as having a unit radius, but when it comes time to convert sides to distances, we'll assume the Earth's radius is $R = 6371$ km.

Since C is positioned at the North pole, its angle is the difference between A and B's longitudes ($\Delta\lambda = \lambda_1 - \lambda_2$). We'll occasionally refer to the bearing of someone leaving A to travel to B as θ, which is equivalent to the angle A, at least initially. As mentioned before, a drawback of great circle navigation is that this bearing will change as the traveler moves towards B.

The required distance is c, which suggest the use of the cosine law:

$$\cos c = \cos a \cdot \cos b + \sin a \cdot \sin b \cdot \cos C$$

It can be rephrased to use latitudes:

$$\begin{aligned} \cos c &= \cos(\pi/2 - \phi_2) \cdot \cos(\pi/2 - \phi_1) + \sin(\pi/2 - \phi_2) \cdot \sin(\pi/2 - \phi_1) \cdot \cos C \\ &= \sin\phi_2 \cdot \sin\phi_1 + \cos\phi_2 \cdot \cos\phi_1 \cdot \cos C \end{aligned}$$

13.10.1 Distance. Given the coordinates of two points, the great circle distance between them is obtained using nav.distTo() from nav.py.

```
def distTo(lat1, lon1, lat2, lon2):
  phi1 = math.radians(lat1)
  phi2 = math.radians(lat2)
  dLambda = math.radians(lon2 - lon1)

  # Spherical law of cosines
  angle = math.acos(
    math.sin(phi1) * math.sin(phi2) + \
    math.cos(phi1) * math.cos(phi2) * math.cos(dLambda)
  )
  return EARTH_RADIUS * angle
```

The latitude and longitudes are supplied as decimal degrees, and the distance is returned in kilometers. In the code, `angle` corresponds to the spherical triangle side c.

13.10.2 The Haversine Distance. The *haversine* formula was a once-popular alternative for finding the distance since it's less susceptible to rounding errors when the distance is small. Our nav.havDistTo() is:

```
def havDistTo(lat1, lon1, lat2, lon2):
  phi1 = math.radians(lat1)
  phi2 = math.radians(lat2)
  deltaPhi = math.radians(lat2 - lat1)
  dLambda = math.radians(lon2 - lon1)

  h = math.sin(deltaPhi/2)**2 + \
```

```
        math.cos(phi1) * math.cos(phi2) * \
        math.sin(dLambda/2)**2
    c = 2 * math.atan2(math.sqrt(h), math.sqrt(1 - h))
    return EARTH_RADIUS * c
```

In distTo(), math.acos() returns the angular distance c (see Fig. 13.15), but if the distance is small then accuracy may be lost, especially if we're relying on cosine tables rather than a computer.

In havDistTo(), the corresponding operation is the job of math.atan2(), and the inverse tangent of a number can be nicely approximated by the number itself, resulting in much less loss of precision.

That being said, modern programming languages (including Python) offer roughly 15 significant figures of precision for floats, which means that distTo() and havDistTo() produce equivalent distances down to a few meters. We checked this with `havDiff.py` which calculates distances using both approaches in the range 1 to 1000 meters in steps of 5. A graph showing the absolute differences between their values is shown in Fig. 13.16.

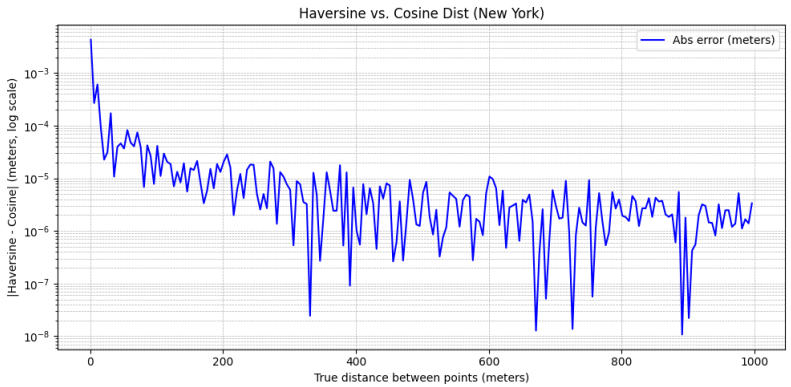


Figure 13.16. Haversine vs. Cosine Distance

The small errors are acceptable especially when our assumption that the Earth is a sphere introduces much greater errors. For example, the Earth's radius varies from 6,336 km (equatorial meridian) to 6,399 km (at the poles), whereas we're fixing it at 6,371 km. This means that calculated distance can be in error by up to 0.3%, or roughly 22 km, depending on the points in question.

The basic haversine formula is:

$$\text{haver}(x) = \sin^2(x/2) = \frac{1 - \cos x}{2}$$

The '(re)versed sine' is $1 - \cos x$ and the 'half-versed-sine' is $(1 - \cos x)/2$. After a little manipulation based around a spherical triangle such as the one in Fig.

13.14, the *law of haversine* is:

$$\text{haver}(c) = \text{haver}(a-b) + \sin a \sin b \, \text{haver}(C)$$

Utilizing the co-latitudes for a and b (see Fig. 13.15), this becomes:

$$\text{haver}(c) = \text{haver}(\phi_1 - \phi_2) + \cos\phi_2 \cos\phi_1 \, \text{haver}(\Delta\lambda)$$

The two uses of `haver(x)` on the right-hand side can be replaced by the $\sin^2(x/2)$ definition:

$$\text{haver}(c) = \sin^2 \Delta\phi + \cos\phi_2 \cos\phi_1 \sin^2 \Delta\lambda$$

where $\Delta\phi = \phi_1 - \phi_2$.

Let $h = \sin^2 \Delta\phi + \cos\phi_2 \cos\phi_1 \sin^2 \Delta\lambda$, and we want $c = \text{haver}^{-1}(h)$. Instead of directly inverting the haversine, we use the numerically stable identity involving the arc-tangent:

$$\begin{aligned} c &= \text{haver}^{-1}(h) \\ &= 2 \cdot \arcsin(\sqrt{h}) \\ &= 2 \cdot \arctan(\tfrac{\sqrt{h}}{\sqrt{1-h}}) \\ &= 2 \cdot \operatorname{arctan2}(\sqrt{h}, \sqrt{1-h}) \end{aligned}$$

13.10.3 Bearing. Given the coordinates of two points, A and B, the initial bearing from A to B is obtained by calling nav.bearingTo() from `nav.py`:

```
def bearingTo(lat1, lon1, lat2, lon2):
  # return degree bearing from first to second coord
  phi1 = math.radians(lat1)
  phi2 = math.radians(lat2)
  dLambda = math.radians(lon2 - lon1)

  y = math.sin(dLambda) * math.cos(phi2)
  x = math.cos(phi1) * math.sin(phi2) - \
      math.sin(phi1) * math.cos(phi2) * math.cos(dLambda)
  theta = math.atan2(y, x)
  return (math.degrees(theta) + 360) % 360
```

The bearing is returned in degrees in the range 0°...360°, with negative values mapped to the range 180° – 360°.

The function returns the angle at A (see Fig. 13.15), by taking the arc-tangent of y/x ($\tan A$). This division is best thought of as $\sin A/\cos A$ so y can be equated with $\sin A$ and x with $\cos A$.

y ($\sin A$) is obtained via the sine law:

$$\frac{\sin a}{\sin A} = \frac{\sin b}{\sin B} = \frac{\sin c}{\sin C}$$

a is rephrased as a co-latitude, and C as the longitude delta:

$$\frac{\sin(\pi/2 - \phi_2)}{\sin A} = \frac{\sin c}{\sin \Delta\lambda}$$

so:

$$\sin A = \frac{\sin \Delta\lambda \cdot \cos \phi_2}{\sin c}$$

For the determination of x ($\cos A$), start with the cosine law for a:

$$\cos a = \cos b \cdot \cos c + \sin b \cdot \sin c \cdot \cos A$$

Rearranging:

$$\begin{aligned}
\sin b \cdot \sin c \cdot \cos A &= \cos a - \cos b \cdot \cos c \\
&= \cos a - \cos b \cdot (\cos a \cdot \cos b + \sin a \cdot \sin b \cdot \cos C) \\
&= \cos a - \cos^2 b \cdot \cos a - \sin a \cdot \sin b \cdot \cos b \cdot \cos C \\
&= \cos a - (1 - \sin^2 b) \cdot \cos a - \sin a \cdot \sin b \cdot \cos b \cdot \cos C \\
&= \sin^2 b \cdot \cos a - \sin a \cdot \sin b \cdot \cos b \cdot \cos C
\end{aligned}$$

Divide through by $\sin b \cdot \sin c$:

$$\cos A = \frac{\sin b \cdot \cos a - \sin a \cdot \cos b \cdot \cos C}{\sin c}$$

Rewrite a and b as co-latitudes:

$$\begin{aligned}
\cos A &= \frac{\sin(\pi/2 - \phi_1) \cdot \cos(\pi/2 - \phi_2) - \sin(\pi/2 - \phi_2) \cdot \cos(\pi/2 - \phi_1) \cdot \cos C}{\sin c} \\
&= \frac{\cos \phi_1 \cdot \sin \phi_2 - \cos \phi_2 \cdot \sin \phi_1 \cdot \cos \Delta\lambda}{\sin c}
\end{aligned}$$

Combine these $\sin A$ and $\cos A$ equations:

$$\frac{y}{x} = \tan A = \frac{\sin A}{\cos A} = \frac{\sin \Delta\lambda \cdot \cos \phi_2}{\cos \phi_1 \cdot \sin \phi_2 - \cos \phi_2 \cdot \sin \phi_1 \cdot \cos \Delta\lambda}$$

This numerator and denominator are implemented in the code for `bearingTo()`.

13.10.4 Calculating a Destination.

nav.destCoord() is supplied with a starting point, an initial bearing, a distance to travel, and returns the destination.

```
def destCoord(lat1, lon1, dist, bearing):
  theta = math.radians(bearing)
  phi1 = math.radians(lat1)
  lambda1 = math.radians(lon1)
  delta = dist/EARTH_RADIUS

  phi2 = math.asin(math.sin(phi1) * math.cos(delta) + \
             math.cos(phi1) * math.sin(delta) * \
             math.cos(theta)  )

  lambda2 = lambda1 + \
       math.atan2(math.sin(theta) * math.sin(delta) * \
```

```
                math.cos(phi1),
                math.cos(delta) - math.sin(phi1) *  \
                math.sin(phi2)  )

    lambda2 = (lambda2 + math.pi) % (2 * math.pi) - math.pi
    return math.degrees(phi2), math.degrees(lambda2)  # lat, lon
```

With reference to Fig. 13.15, the code is calculating ϕ_2 and λ_2 of point B. The latitude, ϕ_2, is obtained via the cosine law for a:

$$\cos a = \cos b \cdot \cos c + \sin b \cdot \sin c \cdot \cos A$$

Substitute the co-latitudes for a and b:

$$\begin{aligned} \cos(\pi/2 - \phi_2) &= \cos(\pi/2 - \phi_1) \cdot \cos c + \sin(\pi/2 - \phi_1) \cdot \sin c \cdot \cos A \\ \sin \phi_2 &= \sin \phi_1 \cdot \cos c + \cos \phi_1 \cdot \sin c \cdot \cos A \end{aligned}$$

The longitude is:

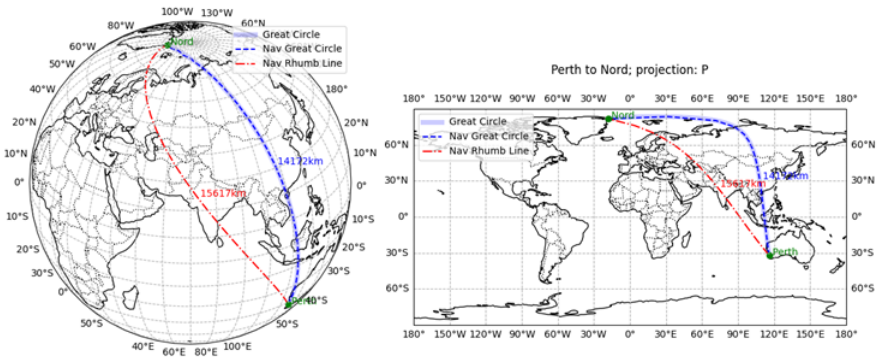
$$\lambda_2 = \lambda_1 + \arctan(\tan \Delta\lambda)$$

or

$$\lambda_2 = \lambda_1 + \arctan(\tan C)$$

The tangent can be written as $\sin C / \cos C$, which splits the problem into two parts. The next steps are quite similar to those used in the bearing formula above, so won't be explained again.

13.10.5 Drawing the Great Circle. showMap.py utilizes the great circle functions in nav.py to offer the user a choice of four map projections for displaying two great circles and a rhumb line connecting Perth, Australia to Nord, Greenland. Fig. 13.17 shows the Orthographic and Platte Carrée options.



(a) Orthographic Projection (b) Platte Carree Projection

Figure 13.17. Orthographic and Platte Carrée Trips from Perth to Nord

Although the curves appear differently on the two maps, a study of the countries they pass over reveals them to be the same. The distances – 14172 km for the great circles, and 15617 km for the rhumb line – are also the same. We'll discuss rhumb lines in the next section, and focus here on how the great circles are drawn.

The solid curve, labeled 'Great Circle' on the map, is drawn using the Cartopy plot() function:

```
plt.plot([lon1, lon2], [lat1, lat2],
     color='blue', lw=5, alpha=0.2,
     label="Great Circle", transform=ccrs.Geodetic())
```

It only requires the coordinates of the two end points, and uses a Geodetic() transform to interpolate great circle coordinates between those points for the plotting. We've included this code mainly as a check that the coordinates generated by our nav.greatCircleCoords() are correct. That function is called like so:

```
coords = nav.greatCircleCoords(lat1, lon1, lat2, lon2, GCStep)
print("\nGreat Circle (lat,lon)s and deg Bearing:")
for coord in coords:
  cLat, cLon, cBear = coord
  print(f"  ({cLat:.2f}, {cLon:.2f}); {cBear:.2f}")
```

The returned list contains triplets of the form (latitude, longitude, bearing); the most interesting aspect of the data is that the bearing changes at each point along the line (as you'd expect with great circles):

```
  :
(43.94, 105.62); 351.10
(50.23, 104.09); 349.97
(56.49, 102.08); 348.36
(62.71, 99.28); 345.94
(68.83, 95.00); 342.02
(74.77, 87.50); 334.89
(80.17, 71.48); 319.23
  :
```

The data is passed to Cartopy's plot():

```
lats, lons, _ = zip(*coords)
plt.plot(lons, lats, color='blue', linestyle='--',
         label="Nav Great Circle",
         transform=ccrs.PlateCarree())
```

The key difference here is that the PlateCarree() transform connects adjacent points in the list with straight lines without generating intermediate points.

nav.greatCircleCoords() produces its list by first calculating the total distance to the destination with nav.distTo(), from which it obtains a step distance between points. A loop calls nav.destCoord() to move forward by that step using

the current bearing, then nav.bearingTo() updates the bearing for the new location, and the loop repeats:

```
def greatCircleCoords(lat1, lon1, lat2, lon2, num):
  lat1 = parseCoord(lat1)
  lon1 = parseCoord(lon1)
  lat2 = parseCoord(lat2)
  lon2 = parseCoord(lon2)

  dist = distTo(lat1, lon1, lat2, lon2)
  distStep = dist/num

  bearing = bearingTo(lat1, lon1, lat2, lon2)
  nSteps = 1
  coords = [(lat1, lon1, bearing)]
  currLat, currLon = lat1, lon1
  while (nSteps < num) or \
        (not areClose(currLat, currLon, lat2, lon2)):
    destLat, destLon =  \
       destCoord(currLat, currLon, distStep, bearing)
    bearing = bearingTo(destLat, destLon, lat2, lon2)
    nSteps += 1
    coords.append((destLat, destLon, bearing))
    currLat, currLon = destLat, destLon
  return coords
```

The call to areClose() adds a second test to end coordinate generation if the current position is sufficient close to the destination.

13.11 The Mercator Projection

Sailors have plotted courses using rhumb lines and compass headings since the thirteenth century. Mathematicians call these lines loxodromes, from the Latin *loxos* - slant and *drome* - running, because they cut meridians at the same angle. One consequence is that as a rhumb line heads north (or south), and the meridians get closer together, it turns steadily toward the pole, spiraling around it as in Fig. 13.18.

The curve in Fig. 13.18 was drawn using nav.rhumbDestCoord() which is described in section 13.12.3 below. The complete code is available in NYCSpiral.py.

Sailors initially tried to plot rhumb lines on planar charts (i.e. maps where the lines of latitude are equally spaced) but the position errors gradually increase with the line's length, a problem finally solved by Gerhardus Mercator (1512–1594). His map was still laid out on an easy-to-understand rectangular grid: all the circles of latitude were represented by horizontal lines parallel to the equator

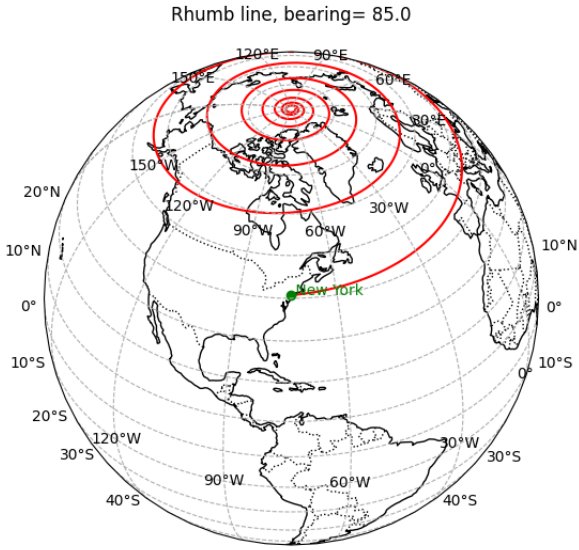


Figure 13.18. A Rhumb Line Bearing of 85° from New York City

and equal to it in length, and all the meridians were drawn as vertical lines perpendicular to the equator. But the important change was that the projection was *conformal* – the map preserved the angles between all the points on the globe.

On a globe, the parallels decrease in size at higher latitudes, until they shrink to a point at either pole. Mercator's map draws these circles as horizontal lines of equal length, which means that each parallel has to be stretched horizontally by a factor depending on its latitude.

Fig. 13.19 shows a parallel at latitude ϕ, which has a radius of $R\cos\phi$ and circumference $2\pi R\cos\phi$ on the globe.

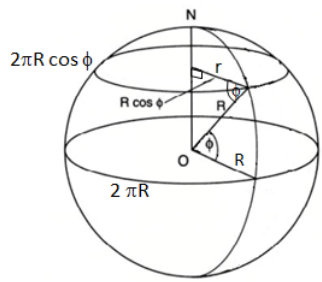


Figure 13.19. Lengths for Latitude ϕ

On Mercator's map this parallel must be stretched to be the same length as the equator, $2\pi R$, which requires it to be multiplied by the factor $\frac{2\pi R}{2\pi R \cos\phi} = \sec\phi$. Note that this stretching is a function of ϕ: the higher the latitude, the greater the ratio.

Mercator's big idea was that in order for the map to be angle preserving, the east-west stretch of the parallels had to be accompanied by an *equal* north-south stretch of the spacings between the parallels, implying that they had to progressively increase at higher latitudes. In other words, equally spaced latitudes along each meridian on a globe will gradually increase on the map (see Fig. 13.20).

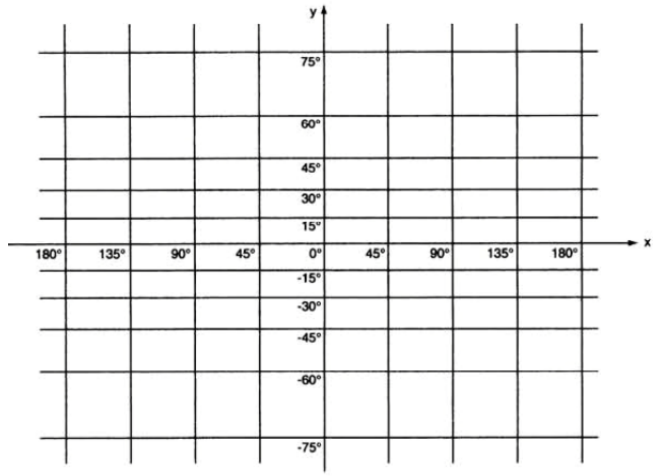


Figure 13.20. The Mercator Grid

Fig. 13.20 also highlights the map's main shortcoming: distances are hard to measure because regions far to the north and south are grossly exaggerated in size (e.g. compare Greenland to Africa on a globe and on a Mercator map).

Mercator first published his map in 1569 with the succinct title *New and Improved Description of the Lands of the World, Amended and Intended for the Use of Navigators*. It was massive, printed across 18 sheets, each measuring 33 by 40 cm (`https://en.wikipedia.org/wiki/Mercator_1569_world_map`).

Mercator never gave a full account of how he had progressively increased the distances between the parallels. It was left to Edward Wright in 1599 to present the mathematical details in *Certaine errors in navigation, arising either of the ordinarie erroneous making or vsing of the sea chart, compasse, crosse stafie, and tables of declination of the sunne, and fixed starres detected and corrected*. He's also devised the 'bladder' explanation of the projection, responsible for much mirth in school classrooms ever since.

Wright asks us to imagine a spherical bladder representing the Earth, with meridians and parallels drawn on its surface. It's placed inside a vertical cylinder representing the map projection, preferably made of glass so you can see the projection develop (see Fig. 13.21).

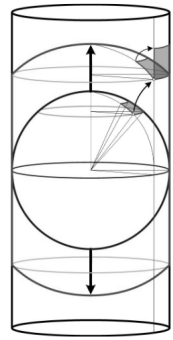


Figure 13.21. Wright's Expanding Bladder in a Cylinder

Inflate the bladder slowly so that its vertical axis aligns with the axis of the cylinder until its equator line touches the cylinder walls. This sets the equator on the bladder and cylinder to be the same. Then continue slowly inflating the bladder so that it stretches by the same amount in all directions – this enforces Mercator's angle-preserving condition. As a consequence, the parallels on the bladder grow farther and farther apart, but because the cylinder is open ended, the bladder's poles never touch the cylinder's sides.

A mathematical analysis of the process is outlined by Jim Stevenson in *Mercator Projection Balloon*, online at `https://josmfs.net/wordpress/wp-content/uploads/2018/12/Mercator-Balloon-170306.pdf`.

Fig. 13.22 shows the projection problem: the spherical rectangle on the globe, with width $(R\cos\phi)\Delta\lambda$ and height $R\Delta\phi$, must be mapped to a planar rectangle of width Δx and height Δy.

Fig. 13.23 depicts the two rectangles (spherical and planar) in more detail. The spherical rectangle is defined by meridians of longitudes λ_1 and $\lambda_1 + \Delta\lambda$, and parallels of latitudes ϕ_1 and $\phi_1 + \Delta\phi$. The sides of this rectangle have length $(R\cos\phi_1)\Delta\lambda$ and $R\Delta\phi$ respectively. The planar rectangle is delimited by the lines x_1, $x_1 + \Delta x$, y_1, and $y_1 + \Delta y$, with sides $R\Delta x$ and $R\Delta y$.

The compass bearing from P on the globe is at the angle ψ, and since the projection is angle preserving this is also true for the bearing from P' in the projection.

Since the meridians in the Mercator projection are vertical straight lines, this implies that $\Delta x = \Delta\lambda$, the angular distance (or delta) between the two longitudes at the equator.

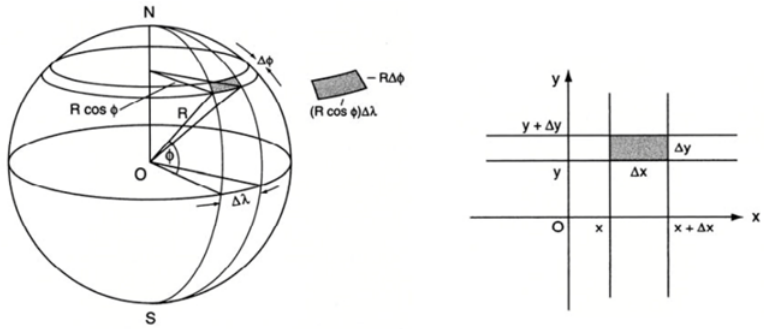


Figure 13.22. The Mercator Projection of a Rectangle

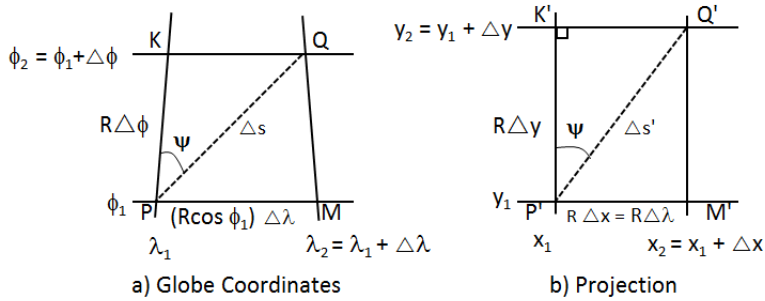


Figure 13.23. The Spherical and Planar Rectangles

The scaling in the x-direction is therefore:

$$\frac{P'M'}{PM} = \frac{R\Delta\lambda}{(R\cos\phi_1)\Delta\lambda} = \frac{1}{\cos\phi_1} = \sec\phi_1$$

The scaling in the y-direction must be the same, and so:

$$\begin{aligned} \frac{K'P'}{KP} &= \frac{R\Delta y}{R\Delta\phi} = \sec\phi_1 \\ \Delta y &= (\sec\phi_1)\Delta\phi \end{aligned}$$

This is a finite difference equation, which Wright solved with numerical integration, a time-consuming procedure before computers. An alternative available to us is to write it as a differential equation: let Δy and $\Delta\phi$ become infinitely small, and in the limit we get

$$\frac{dy}{d\phi} = \sec\phi$$

whose solution is

$$y = \int_0^{\phi} \sec\phi \, d\phi,$$

which must be multiplied by R if the sphere doesn't have a unit radius.

Geographers of the time didn't have calculus yet, so searched for a formula via other techniques. In 1620, Edmund Gunter published a table of $\log(\tan\theta)$ values, and in 1645 Henry Bond discovered that the latitude lines on the Mercator map were analogous to a scaling of those values. Essentially, he realized that

$$\int_0^{\phi} \sec\phi \, d\phi = -\ln\tan\left(\frac{1}{2}(\frac{\pi}{2} - \phi)\right)$$

Newton and Leibnitz developed the calculus during the 1660s, and in 1668, via a complicated geometric argument, James Gregory proved the equation. During the next decade or two, simpler proofs of $\int \sec\phi \, d\phi$ appeared, and a modern notation version of Isaac Barrow's solution from 1670 is given below. It was also the first to employ partial fractions decomposition.

$$\begin{aligned}
\sec\phi &= \frac{1}{\cos\phi} = \frac{\cos\phi}{\cos^2\phi} \\
&= \frac{\cos\phi}{1-\sin^2\phi} = \frac{\cos\phi}{(1+\sin\phi)(1-\sin\phi)} \\
&= \frac{1}{2}\left(\frac{\cos\phi}{1+\sin\phi} + \frac{\cos\phi}{1-\sin\phi}\right)
\end{aligned}$$

So the task becomes,

$$\int \sec\phi \, d\phi = \frac{1}{2}\int \left(\frac{\cos\phi}{1+\sin\phi} + \frac{\cos\phi}{1-\sin\phi}\right) d\phi$$

The first term inside the right-hand integral is of the form $\frac{u'}{u}$, where u is a function of ϕ, and the second term is of the form $-\frac{u'}{u}$. So, using

$$\int \frac{u'}{u} \, d\phi = \ln|u(\phi)| + C,$$

we get

$$\begin{aligned}
\int \sec\phi \, d\phi &= \frac{1}{2}\left[\ln|1+\sin\phi| - \ln|1-\sin\phi|\right] + C \\
&= \frac{1}{2}\ln\left|\frac{1+\sin\phi}{1-\sin\phi}\right| + C.
\end{aligned}$$

Inside the logarithm, multiply top and bottom by $(1+\sin\phi)$. The denominator becomes

$$(1-\sin\phi)(1+\sin\phi) = 1 - \sin^2\phi = \cos^2\phi,$$

so the expression inside the logarithm is now:

$$\int \sec\phi \, d\phi = \frac{1}{2}\ln\left|\frac{(1+\sin\phi)^2}{\cos^2\phi}\right| + C$$

or, using the power property of logarithms:

$$= \ln \left| \frac{1 + \sin \phi}{\cos \phi} \right| + C.$$

Now, write $\phi = 2(\phi/2)$, and use the double-angle formulas:

$$\sin \phi = 2 \sin \phi/2 \cos \phi/2, \quad \cos \phi = \cos^2 \phi/2 - \sin^2 \phi/2$$

and

$$1 = (\cos \pi/2)^2 + (\sin \phi/2)^2$$

to rewrite the expression inside the logarithm:

$$\begin{aligned} \frac{1+\sin \phi}{\cos \phi} &= \frac{1+2 \sin \phi/2 \cos \phi/2}{\cos^2 \phi/2 - \sin^2 \phi/2} \\ &= \frac{(\cos \pi/2)^2 + (\sin \phi/2)^2 + 2 \sin \phi/2 \cos \phi/2}{\cos^2 \phi/2 - \sin^2 \phi/2} \\ &= \frac{(\cos \phi/2 + \sin \phi/2)^2}{(\cos \phi/2 + \sin \phi/2)(\cos \phi/2 - \sin \phi/2)} \\ &= \frac{\cos \phi/2 + \sin \phi/2}{\cos \phi/2 - \sin \phi/2} \end{aligned}$$

Divide the numerator and denominator by $\cos \phi/2$:

$$\frac{1 + \tan \phi/2}{1 - \tan \phi/2}$$

This can be simplified using:

$$\tan(A + B) = \frac{\tan A + \tan B}{1 - \tan A \cdot \tan B}$$

with $A = \pi/4$ and $B = \phi/2$. So:

$$\frac{1 + \tan \phi/2}{1 - \tan \phi/2} = \tan (\pi/4 + \phi/2)$$

Substitute this into the logarithmic expression:

$$\int \sec \phi \, d\phi = \ln \left| \tan \left(\frac{\pi}{4} + \frac{\phi}{2} \right) \right| + C.$$

Turning now to the definite integral, we have

$$\int_0^{\phi} \sec t \, dt = \ln \left| \tan \left(\frac{\pi}{4} + \frac{\phi}{2} \right) \right| - \ln \tan \frac{\pi}{4}$$

But $\ln(\tan \frac{\pi}{4}) = \ln 1 = 0$, so

$$\int_0^{\phi} \sec t \, dt = \ln \tan \left(\frac{\pi}{4} + \frac{\phi}{2} \right)$$

We've dropped the absolute value sign because in the relevant range for ϕ, — $\pi/2 < \phi < \pi/2$, $\tan(\pi/4 + \phi/2)$ is always positive.

We began with:

$$y = \int_0^{\phi} \sec t \, dt$$

It can now be expressed as:

$$y = \ln \tan\left(\frac{\pi}{4} + \frac{\phi}{2}\right),$$

where $R \cdot y$ is the height of a parallel at latitude ϕ on a Mercator map.

This form of the equation quickly became popular because tables of $\log(\tan\theta)$ values had already been created. However, John Wallis published a series form for the integral in 1685, offering an alternative way of computing the height. From the partial fractions approach,

$$\int_0^{\phi} \sec t \, dt = \int_0^{\phi} \frac{\cos t}{1 - \sin^2 t} \, dt,$$

The fraction can be expanded as:

$$= \int_0^{\theta} \cos t \left[1 + \sin^2 t + \sin^4 t + \sin^6 t + \ldots\right] dt.$$

This uses the geometric series:

$$\frac{1}{1-a} = 1 + a + a^2 + a^3 + a^4 + \ldots \quad \text{if} \quad |a| < 1,$$

with $a = \sin^2 t$, which satisfies $|a| < 1$ for $t \in (-\pi/2, \pi/2)$.

The next step is to convert this integral-of-an-infinite-sum into an infinite-sum-of-integrals, which can be calculated term by term. In general, such a transformation may *not* be valid, but is acceptable here because this series is convergent for all the values in the interval, and the functions are continuous.

$$\begin{aligned} \int_0^{\phi} \sec t \, dt &= \int_0^{\phi} \cos t \cdot 1 \, dt + \int_0^{\phi} \cos t \cdot \sin^2 t \, dt + \int_0^{\phi} \cos t \cdot \sin^4 t \, dt + \ldots \\ &= \sin\phi + \frac{\sin^3\phi}{3} + \frac{\sin^5\phi}{5} + \frac{\sin^7\phi}{7} + \ldots \\ &= y + \frac{y^3}{3} + \frac{y^5}{5} + \frac{y^7}{7} + \ldots \quad \text{with } y = \sin\phi. \end{aligned}$$

13.12 Calculating with Rhumb Lines

Calculations in nav.py involving rhumb lines follow the same pattern as those for great circles: finding the distance between two points, determining the bearing to travel from one point to another (which will stay the same throughout the trip), and obtaining the destination coordinate if only the start point, bearing, and travel distance are supplied.

13.12.1 Distance. Given the coordinates of two points, the rhumb line distance between them is obtained using nav.rhumbDistTo() in nav.py.

```
def rhumbDistTo(lat1, lon1, lat2, lon2):
  phi1 = math.radians(lat1)
  phi2 = math.radians(lat2)
  deltaPhi = phi2 - phi1
  dLambda = math.radians(lon2 - lon1)

  deltaY = math.log(math.tan(math.pi/4 + phi2/2) / \
                    math.tan(math.pi/4 + phi1/2))
  q = deltaPhi/deltaY if abs(deltaY) > 1e-12 \
                      else math.cos(phi1)
  ''' If on a constant latitude course (traveling east-west),
      use cos(phi)
  '''
  if abs(dLambda) > math.pi:
    # use shorter rhumb line across the date line
    dLambda = dLambda - 2*math.pi if dLambda > 0 \
                                  else dLambda + 2*math.pi
  return math.sqrt(deltaPhi**2 + (q * dLambda)**2) * EARTH_RADIUS
```

The latitude and longitude are supplied as decimal degrees, and the distance is returned in kilometers.

`deltaY` is Δy in the projection in Fig. 13.23(b), which is obtained by subtracting the two Mercator values (y_2 and y_1), albeit by utilizing division inside the logarithmic expression.

`q` is the ratio $\frac{\Delta\phi}{\Delta y}$ which defines the scaling from lengths in the projection to their globe values.

From Fig. 13.23(b):

$$\Delta s' = R\sqrt{\Delta y^2 + \Delta x^2}$$

Let

$$q = \frac{\Delta\phi}{\Delta y} \text{ if } |\Delta\phi| > 0, \text{ otherwise } \cos\phi_1$$

and so

$$\Delta y = \frac{\Delta\phi}{q}$$

The conditional associated with q deals with a trip that heads directly east or west from the starting point, P.

Replace the Δy term in the square root:

$$\begin{aligned} \Delta s' &= R\sqrt{\Delta y^2 + \Delta\lambda^2} \\ &= R\sqrt{\left(\frac{\Delta\phi}{q}\right)^2 + \Delta\lambda^2} \\ &= \frac{R}{q}\sqrt{\Delta\phi^2 + (q\Delta\lambda)^2} \end{aligned}$$

The q ratio links other lengths in the spherical and planar rectangles; in particular, it can be used to relate the travel distances

$$\begin{aligned} q &= \frac{\Delta s}{\Delta s'} \\ \Delta s &= q\Delta s' \end{aligned}$$

$\Delta s'$ can be replaced by the square root from above, producing:

$$\Delta s = q\Delta s' = R\sqrt{\Delta\phi^2 + (q\Delta\lambda)^2}$$

This is the expression calculated by nav.rhumbDistTo().

13.12.2 Bearing. Given the coordinates of two points, the bearing along the rhumb line is obtained using nav.rhumbBearingTo():

```
def rhumbBearingTo(lat1, lon1, lat2, lon2):
  # return degree bearing from first to second coord
  phi1 = math.radians(lat1)
  phi2 = math.radians(lat2)
  dLambda = math.radians(lon2 - lon1)

  deltaY = math.log(math.tan(math.pi/4 + phi2/2)/ \
                    math.tan(math.pi/4 + phi1/2))

  if abs(dLambda) > math.pi:
    # use shorter rhumb line across the date line
    dLambda = dLambda - 2*math.pi if dLambda > 0  \
                                  else dLambda + 2*math.pi

  psi = math.atan2(dLambda, deltaY)
  return (math.degrees(psi) + 360) % 360
```

`psi` in the code is ψ in Fig. 13.23(b), which is obtained from the arc-tangent of $\frac{\Delta x}{\Delta y}$.

The bearing is returned in degrees in the range 0°..360°, with negative values mapped to the range 180°–360°.

13.12.3 Calculating a Destination. nav.rhumbDestCoord() is supplied with a starting point, the bearing, a distance to travel, and returns the destination coordinate.

```
def rhumbDestCoord(lat1, lon1, dist, bearing):
  phi1 = math.radians(lat1)
  lambda1 = math.radians(lon1)
  psi = math.radians(bearing)
  delta = dist/EARTH_RADIUS

  deltaPhi = delta * math.cos(psi)
```

```
  phi2 = phi1 + deltaPhi

  # normalize latitude if going past the pole
  if phi2 > math.pi/2:
    phi2 = math.pi - phi2
  elif phi2 < -math.pi/2:
    phi2 = -math.pi - phi2

  deltaY = math.log(math.tan(math.pi/4 + phi2/2)/ \
                    math.tan(math.pi/4 + phi1/2))
  q = deltaPhi/deltaY if abs(deltaY) > 1e-12 \
                      else math.cos(phi1)

  dLambda = delta * math.sin(psi)/q
  lambda2 = lambda1 + dLambda

  deglambda = ((math.degrees(lambda2) + 540) % 360) - 180
  return math.degrees(phi2), deglambda  # lat, lon
```

In order to calculate the latitude and longitude of the destination (point Q in Fig. 13.23(a)), we must obtain the two deltas $\Delta\phi$ and $\Delta\lambda$.

For $\Delta\phi$, note that the `bearing` in the code is ψ in Fig. 13.23(a), `dist` is Δs, and `delta` is $\frac{\Delta s}{R}$. Therefore:

$$\cos\psi = \frac{R\Delta\phi}{\Delta s} = \frac{\Delta\phi}{\text{delta}}$$

and so

$$\Delta\phi = \text{ delta } \cdot \cos\psi$$

For $\Delta\lambda$, we again make use of the q ratio (first seen in the distance calculation):

$$q = \frac{\Delta\phi}{\Delta y} = \frac{\Delta s}{\Delta s'}$$

or

$$\Delta s' = \frac{\Delta s}{q}$$

With reference to Fig. 13.23(b):

$$\sin\psi = \frac{R\Delta\lambda}{\Delta s'}$$

$$R\Delta\lambda = \sin\psi \cdot \Delta s' = \sin\psi \cdot \frac{\Delta s}{q}$$

Divide through by R, and replace $\frac{\Delta s}{R}$ by `delta`:

$$\Delta\lambda = \sin\psi \cdot \frac{\Delta s}{qR} = \sin\psi \frac{\text{delta}}{q}$$

nav.rhumbDestCoord() adds `deltaPhi` ($\Delta\phi$) to the latitude and `dLambda` ($\Delta\lambda$) to the longitude, normalizes the results, and returns.

13.12.4 Drawing a Rhumb Line. showMap.py utilizes the rhumb line functions in nav.py to offer the user a choice of four map projections for displaying a rhumb line connecting Perth, Australia to Nord, Greenland. Fig. 13.17 shows the Orthographic and Platte Carrée options, while Fig. 13.24 draws on Mercator and Gnomonic maps.

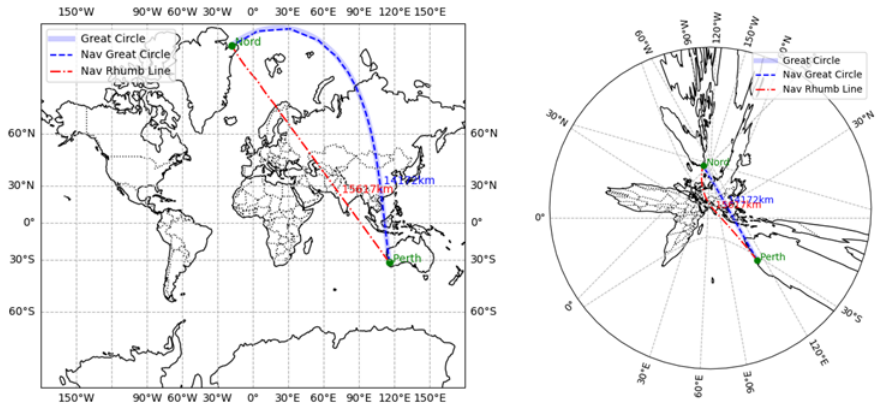


Figure 13.24. Mercator and Gnomonic Perth to Nord Trips

The obvious benefit of the Mercator projection is that the rhumb line is a straight line. The Gnomonic projection is included here since it has an analogous effect on a great circle, but with the side-effect of heavily distorting the map.

The rhumb line coordinates are generated by nav.rhumbCoords():

```
coords = nav.rhumbCoords(lat1, lon1, lat2, lon2, 20)
print("\nRhumb Line (lat,lon)s and deg Bearing:")
for coord in coords:
  cLat, cLon, cBear = coord
  print(f"  ({cLat:.2f}, {cLon:.2f}); {cBear:.2f}")
```

The returned list contains (latitude, longitude, and bearing) triplets, with the most interesting aspect being that the bearings do *not* change:

```
  :
(24.88, 72.70); 324.04
(30.56, 68.04); 324.04
(36.25, 63.09); 324.04
```

```
(41.93, 57.77); 324.04
(47.62, 51.96); 324.04
(53.30, 45.47); 324.04
(58.98, 38.05); 324.04
(64.67, 29.29); 324.04
  :
```

The data is passed to Cartopy's plot():

```
lats, lons, _ = zip(*coords)
plt.plot(lons, lats, color='red', ls='-.',
    label="Nav Rhumb Line", transform=ccrs.PlateCarree())
```

nav.rhumbCoords() generates its points in a similar way to nav.greatCircleCoords(), but by calling the rhumb line versions of the distance, bearing, and destination functions.

```
def rhumbCoords(lat1, lon1, lat2, lon2, num):
  lat1 = parseCoord(lat1)
  lon1 = parseCoord(lon1)
  lat2 = parseCoord(lat2)
  lon2 = parseCoord(lon2)

  dist = rhumbDistTo(lat1, lon1, lat2, lon2)
  distStep = dist/num

  bearing = rhumbBearingTo(lat1, lon1, lat2, lon2)
  nSteps = 1
  coords = [(lat1, lon1, bearing)]
  currLat, currLon = lat1, lon1
  while (nSteps < num) or \
        (not areClose(currLat, currLon, lat2, lon2)):
    destLat, destLon =  \
       rhumbDestCoord(currLat, currLon, distStep, bearing)
    bearing = rhumbBearingTo(destLat, destLon, lat2, lon2)
    nSteps += 1
    coords.append((destLat, destLon, bearing))
    currLat, currLon = destLat, destLon
  return coords
```

13.12.5 More Mercator Trips. mercatorTrip.py doesn't use any new navigation functions, but restricts itself to drawing a single great circle and rhumb line between any two cities on a Mercator map. The user supplies the city names and nav.readCoord() attempts to find their latitudes and longitudes in the world_cities.csv file. Fig. 13.25 shows two examples of its output.

The main visual change is that the map boundaries are limited by calls to nav.limitExtent() which calculates a rectangular region that is passed to Cartopy.

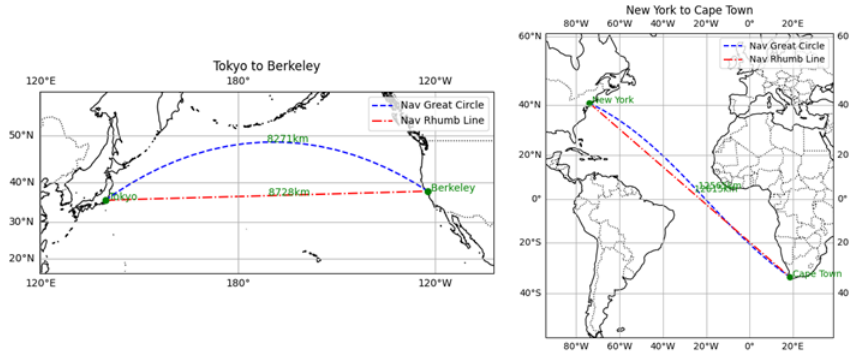


Figure 13.25. Two Mercator Map Trips

Exercises

(1) mercatorTrip.py can be repurposed to investigate a surprisingly tricky question – how much longer can a rhumb line be compared to the great circle between the same two points? The only detailed analysis that we know of was carried out by P.J. Heawood in "Comparison between Rhumb-Line and Great-Circle Courses" [**Hea26**].

(2) A common error when describing the Mercator map is to explain it not by visualizing Wright's gradually expanding bladder but as the result of wrapping a cylinder around the Earth, tangent to the equator, as in Fig. 13.26. Locations on the Earth's surface are projected onto the cylinder by a point-size light emitted from the Earth's core.

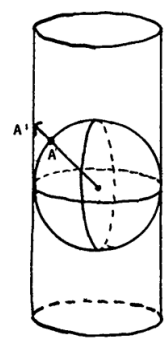


Figure 13.26. Cylindrical Projection: Point A is mapped to A'.

(a) What spacing of the parallels does this projection produce instead of the $R\int_0^{\phi}\sec\phi\,d\phi$ placement at latitude ϕ on the Mercator map?

(b) Are the parallels more spread out? This is difficult to decide as the cylindrical projection also stretches out the map at high latitudes (Fig. 13.27).

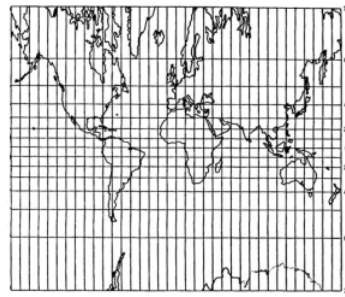


Figure 13.27. Cylindrical Projection World Map

(3) The rhumb line spiral in Fig. 13.18 is drawn by NYCSpiral.py. Experiment with different initial bearings to see how the spiral changes.

The code calls Cartopy plot() like so

```
plt.plot(lons, lats, color='red',
                     transform=ccrs.Geodetic())
      # to deal with curvature between points
```

which uses a `Geodetic` transform rather than the `PlateCarree` argument used elsewhere. Recall that `Geodetic` interpolates extra points between the supplied coordinates while `PlateCarree` merely connects them with a straight line.

If `transform=ccrs.Geodetic()` is replaced by `transform=ccrs.PlateCarree()` in the above code, what happens to the spiral and why?

(4) The Lambert cylindrical equal area projection in Figs. 13.5 and 13.6 of section 13.4 is quite different from the cylindrical mapping in Fig. 13.27. Why are they different?

Based on Fig. 13.28, show that the Lambert projection does indeed preserve areas.

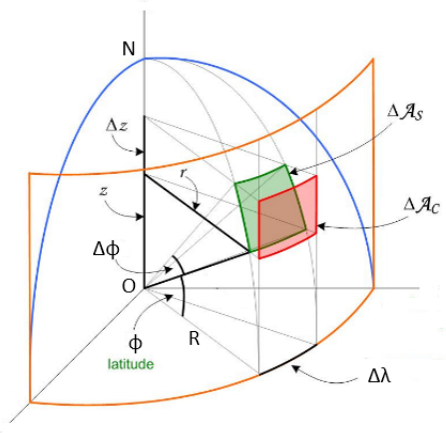


Figure 13.28. Lambert projection of area ΔA_s onto ΔA_c

(5) In 1685, Wallis devised a series form for the integral of $\tan x$ which begins with the following rewrites:

$$\begin{aligned}\int \tan x\,dx &= \int \frac{\sin x}{\cos x}\,dx \\ &= \int \frac{\sin x \cdot \cos x}{\cos^2 x}\,dx \\ &= \int \sin x \cdot \cos x \left(\frac{1}{1-\sin^2 x}\right) dx\end{aligned}$$

Continue the working, which will result in:

$$\int_0^x \tan x\,dx = \frac{1}{2}(s^2 + \frac{s^4}{2} + \frac{s^6}{3} + \frac{s^8}{4} + \cdots),$$

where $s = \sin x$.

(a) For what range of x values does this converge, and why?

(b) What goes wrong when we try to do the same for $\int_0^x \cot x\,dx$?

(6) Use Wallis' series for $\sec x$ to implement a partial sum approximations of the integral in Python. How do you decide when you have enough terms and can stop?

14

Quaternions and Rotation

Why Study Quaternions?

- **They avoid gimbal lock**. Euler angles lose a degree of freedom when two rotation axes align. Quaternions never suffer from this because they represent rotation as a single object in 4D space rather than a sequence of three rotations.
- **They interpolate rotations smoothly**. Smooth, constant-speed rotation between two orientations is hard with matrices or Euler angles. Quaternions make spherical linear interpolation (slerp) simple and numerically robust.
- **They compose rotations efficiently**. Combining two 3D rotations is just quaternion multiplication, which is both faster and more numerically stable than multiplying matrices, and less ambiguous than combining Euler angles.

14.1 Introduction

Quaternions, an extension of the complex numbers, were developed by the Irish mathematician William Rowan Hamilton in 1843 as a notation for modeling 3D mechanics. After Hamilton's death, they gradually fell from favor, displaced by vectors, which proved to be both simpler to use and more widely applicable. However, quaternions had a minor resurgence in the 1920s as a way to describe the spin of electrons and other particles in quantum mechanics, and have been a major element of 3D graphics since the 1980s as a alternative way to implement rotations instead of utilizing Euler angles or matrices [**Sho85**]. In particular, Showmake developed *Slerp* (Spherical Linear intERPolation), which allows

an object to be smoothly interpolated between two orientations. A good introduction to this use of quaternions can be found in [**Len11**] (`https://www.mathfor3dgameprogramming.com/`), and is also the subject of this chapter.

14.2 3D Rotation

Probably the most common way to express 3D rotations is through the use of Euler angles, which define an object's orientation as a sequence of rotations about the x, y, and z axes, called its roll, pitch and yaw, and depicted in Fig. 14.1.

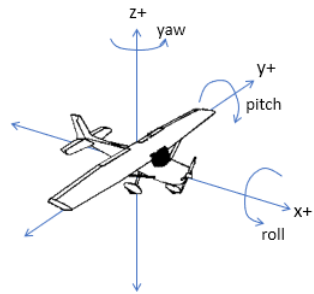


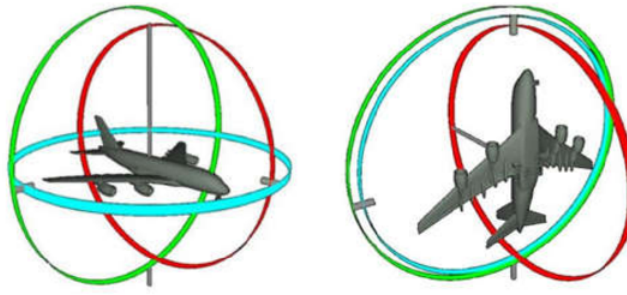
Figure 14.1. The Roll, Pitch, and Yaw of a Plane

A drawback of this approach is that a given orientation can often be defined by different sets of Euler angles. This makes it problematic to reliably undo a series of rotations and return an object to its original position. Also, applying the angle rotations in a different order usually results in different outcomes. Some of this ambiguity can be eliminated by adopting a fixed order for carrying out the rotations, typically around the x-axis, then y, then z, but it doesn't eliminate it altogether.

It's also possible to create a series of rotations with Euler angles that produce a *gimbal lock*, where a degree of freedom is (momentarily) lost. The problem was first observed with gyroscopes in the air and space industry which utilize three concentric rings corresponding to the rotation axes (see Fig. 14.2).

The rings are linked together in a way that reflects the ordering of their axial rotations: the innermost x-ring (red) is linked to the middle y-ring (blue), which connects to the outer z-ring (green). Crucially, turning a ring affects all the rings nested inside it.

When the plane is rotated upwards in Fig. 14.2(b), the y and the z- rings (blue and green) become aligned and a degree of freedom is lost because the x-ring (red) can no longer make the plane roll. Turning the x-ring will cause a yaw in a similar way to a turn to the z-ring (green).



(a) Original Orientation. (b) Pitch 90 Degrees up.

Figure 14.2. The Gimbal Lock Problem

Euler angles are difficult to interpolate: consider the case where you want to gradually turn between 359 and 0 degrees. Linear interpolation may trigger a large rotation, even though the two orientations are almost the same. Ensuring that the shortest path is chosen is easy for one axis, but non-trivial when considering all three angles (for instance, the shortest route between (240, 57, 145) and (35, -233, -270) isn't immediately clear).

These drawbacks of Euler angles are avoided by employing quaternions. As we'll see, a quaternion has a natural geometric interpretation as a rotation about a single axis, there's no possibility of gimbal lock, and interpolation and inverse operations are straight forward. Nevertheless, they do raise some difficulties; one is that quaternions are only really suited to modeling rotations. Other 3D transformations, such as translation, don't have natural quaternion representations. Another issue is the belief that quaternion maths is complicated, but if we restrict ourselves to quaternion-based rotations then they're no harder than complex numbers.

14.3 Vectors Again

We'll explain the maths behind quaternions by utilizing the vector dot and cross product, which we'll briefly describe here once again (you can also find them in section 10.3 of the Computational Geometry topic). These building blocks allow me to define Rodrigues' formula for rotation around an axis, and then it's a short step to quaternion axis-angle rotation.

14.3.1 The Dot Product. The dot product of two n-dimensional vectors $\mathbf{P}$ and $\mathbf{Q}$, written as $\mathbf{P} \cdot \mathbf{Q}$, is the scalar:

$$\mathbf{P} \cdot \mathbf{Q} = \sum_{i=1}^{n} \mathbf{P}_i \mathbf{Q}_i$$

It also satisfies the equation:

$$\mathbf{P} \cdot \mathbf{Q} = \|\mathbf{P}\| \, \|\mathbf{Q}\| \cos \alpha$$

where α is the planar angle between the lines connecting the origin to the points represented by $\mathbf{P}$ and $\mathbf{Q}$ (see Fig. 14.3). A key fact following on from this definition is that two vectors are perpendicular if and only if their dot product is zero.

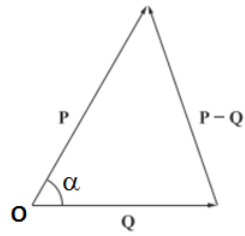


Figure 14.3. The Dot Product Angle

The situation often arises where we need to split a vector $\mathbf{P}$ into its parallel and perpendicular components relative to another vector $\mathbf{Q}$. As shown in Fig. 14.4, if we think of $\mathbf{P}$ as the hypotenuse of a right triangle, then its perpendicular projection onto $\mathbf{Q}$ is the adjacent side for the angle α.

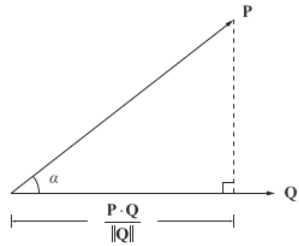


Figure 14.4. The Projection of P onto Q

Trigonometry tells us that the length of the side is $\|\mathbf{P}\| \cos \alpha$, but the dot product gives us a way to calculate this without knowing α:

$$\|\mathbf{P}\| \cos \alpha = \frac{\mathbf{P} \cdot \mathbf{Q}}{\|\mathbf{Q}\|}.$$

To obtain a vector that has this length and is parallel to $\mathbf{Q}$, we multiply by the unit vector $\mathbf{Q}/\|\mathbf{Q}\|$:

$$\mathbf{P}_{\parallel} = \frac{\mathbf{P}\cdot\mathbf{Q}}{\|\mathbf{Q}\|^2}\mathbf{Q}$$

The perpendicular component of $\mathbf{P}$ with respect to $\mathbf{Q}$, $\mathbf{P}_{\perp}$, is obtained by subtracting the parallel component from $\mathbf{P}$:

$$\mathbf{P}_{\perp} = \mathbf{P} - \mathbf{P}_{\parallel} = \mathbf{P} - \frac{\mathbf{P}\cdot\mathbf{Q}}{\|\mathbf{Q}\|^2}\mathbf{Q}$$

14.3.2 The Cross Product. The cross product of $\mathbf{P}$ and $\mathbf{Q}$ returns a vector perpendicular to both of them:

$$\mathbf{P}\times\mathbf{Q} = \langle\, \mathbf{P}_y\mathbf{Q}_z - \mathbf{P}_z\mathbf{Q}_y,\ \mathbf{P}_z\mathbf{Q}_x - \mathbf{P}_x\mathbf{Q}_z,\ \mathbf{P}_x\mathbf{Q}_y - \mathbf{P}_y\mathbf{Q}_x \,\rangle$$

This has many uses in computer graphics, such as the calculation of a surface normal given two tangents to a point.

A commonly used way to generate the product is by evaluating the *pseudo-determinant*:

$$\mathbf{P}\times\mathbf{Q} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \mathbf{P}_x & \mathbf{P}_y & \mathbf{P}_z \\ \mathbf{Q}_x & \mathbf{Q}_y & \mathbf{Q}_z \end{vmatrix}$$

where $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ are unit vectors parallel to the x, y, and z axes:

$$\mathbf{i} = \langle 1,0,0\rangle, \quad \mathbf{j} = \langle 0,1,0\rangle, \quad \mathbf{k} = \langle 0,0,1\rangle.$$

The top row of the determinant consists of vectors while the remaining entries are scalars. Nevertheless, the usual evaluation produces the cross product:

$$\begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \mathbf{P}_x & \mathbf{P}_y & \mathbf{P}_z \\ \mathbf{Q}_x & \mathbf{Q}_y & \mathbf{Q}_z \end{vmatrix} = \mathbf{i}(\mathbf{P}_y\mathbf{Q}_z - \mathbf{P}_z\mathbf{Q}_y) - \mathbf{j}(\mathbf{P}_x\mathbf{Q}_z - \mathbf{P}_z\mathbf{Q}_x) + \mathbf{k}(\mathbf{P}_x\mathbf{Q}_y - \mathbf{P}_y\mathbf{Q}_x).$$

The cross product's length satisfies the equation:

$$\|\mathbf{P}\times\mathbf{Q}\| = \|\mathbf{P}\|\|\mathbf{Q}\|\sin\alpha$$

where α is the angle between the lines connecting the origin to the points represented by $\mathbf{P}$ and $\mathbf{Q}$.

Any nonzero cross product result must be perpendicular to the two vectors multiplied together, but there are two directions that satisfy this requirement. The correct direction is specified by the 'right-hand rule' shown in Fig. 14.5.

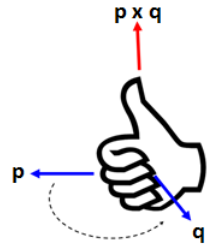


Figure 14.5. The Right-hand Rule for the Cross Product

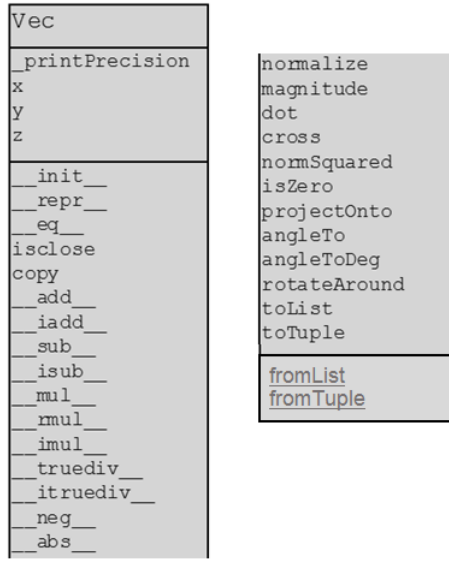


Figure 14.6. Vec Class Diagram

14.4 `Vec`: A vector class in Python

Implementing a 3D vector class in Python is fairly straight forward, so we won't describe much of it here. Its class diagram is given in Fig. 14.6. and the source is online at `Vec.py`.

A Vec object contains x, y, and z values, and supports a standard range of arithmetic operations, as well as dot and cross product. A sample of Vec's capabilities can be seen by calling Vec.py:

```
> python Vec.py
a = Vec(3.00, 4.00, 0.00)
-a = Vec(-3.00, -4.00, 0.00)
|a| = 5.0
b = Vec(0.00, 5.00, 0.00)
a + b = Vec(3.00, 9.00, 0.00)
a . b = 20
a × b = Vec(0.00, 0.00, 15.00)
Normalized 'a' = Vec(0.60, 0.80, 0.00)
Projection of a onto b = Vec(0.00, 4.00, 0.00)
Angle between a and b (deg) = 36.870
Rotate 'a' by 90° around Z axis = Vec(-4.00, 3.00, 0.00)
```

14.4.1 Rotation in 2D. Fig. 14.7 illustrates how a 90-degree counterclockwise rotation of a 2D vector $\mathbf{P}$ in the x-y plane can be accomplished by exchanging its x and y coordinates and negating the new x value. The rotated vector $\mathbf{Q} = \langle -\mathbf{P}_y, \mathbf{P}_x \rangle$ and $\mathbf{P}$ form an orthogonal *basis* for the plane, which means that any other vector can be expressed as a linear combination of these two.

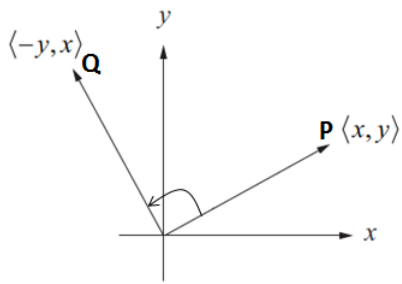


Figure 14.7. Rotation by 90 degrees in the x-y plane

Fig. 14.8 utilizes this basis to show how a vector $\mathbf{P}'$ can be expressed in terms of its components parallel to $\mathbf{P}$ and $\mathbf{Q}$.

Trigonometry lets us write:

$$\mathbf{P}' = \mathbf{P}\cos\theta + \mathbf{Q}\sin\theta$$

which allows us to define $\mathbf{P}'$s x and y components solely in terms of $\mathbf{P}$:

$$\begin{aligned} \mathbf{P}'_x &= \mathbf{P}_x\cos\theta - \mathbf{P}_y\sin\theta \\ \mathbf{P}'_y &= \mathbf{P}_y\cos\theta + \mathbf{P}_x\sin\theta \end{aligned}$$

Or in matrix form:

$$\mathbf{P}' = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \mathbf{P}$$

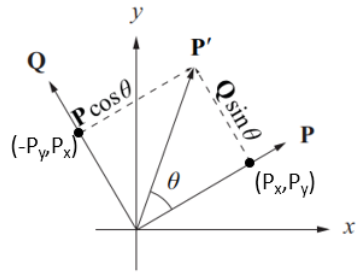


Figure 14.8. $\mathbf{P}'$ in terms of $\mathbf{P}$ and $\mathbf{Q}$

14.5 Rodrigues' Rotation Formula (RRF)

Rodrigues' rotation formula, named after Olinde Rodrigues, is an efficient way to rotate a vector given an axis and angle of rotation. Suppose we want to rotate $\mathbf{v}$ through an angle θ about an axis represented by the unit vector $\mathbf{k}$. We begin by separating $\mathbf{v}$ into its components parallel and perpendicular to $\mathbf{k}$, as in Fig. 14.9.

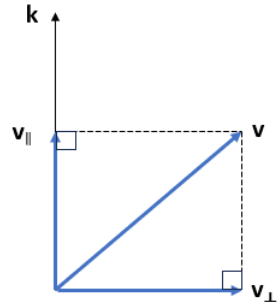


Figure 14.9. $\mathbf{v}$'s Projection ($\mathbf{v}_{\parallel}$) and Perpendicular ($\mathbf{v}_{\perp}$) relative to $\mathbf{k}$

Since $\mathbf{k}$ is a unit vector, the projection of $\mathbf{v}$ onto $\mathbf{k}$ simplifies to:

$$\mathbf{v}_{\parallel} = (\mathbf{k} \cdot \mathbf{v})\mathbf{k}$$

Then the component of $\mathbf{v}$ perpendicular to $\mathbf{k}$ is:

$$\mathbf{v}_{\perp} = \mathbf{v} - (\mathbf{k} \cdot \mathbf{v})\mathbf{k}$$

Since $\mathbf{v}_{\parallel}$ is unchanged during $\mathbf{v}$'s rotation around $\mathbf{k}$, the task reduces to $\mathbf{v}_{\perp}$'s reorientation. This is depicted in Fig. 14.10, which is a view of $\mathbf{v}_{\perp}$ from above, looking down the $\mathbf{k}$ axis.

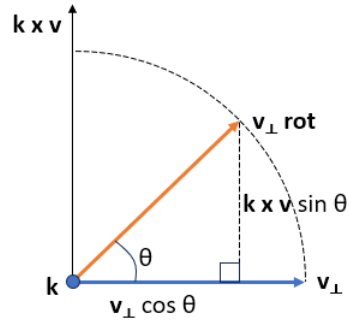


Figure 14.10. Rotating $\mathbf{v}_\perp$, seen from above

The rotation takes place in the plane perpendicular to $\mathbf{k}$. One axis of this plane is $\mathbf{v}_\perp$, and the other can be obtained with $\mathbf{k} \times \mathbf{v}$. Using these two vectors as a basis, the rotation through θ can be represented by:

$$\begin{aligned} \mathbf{v}_\perp \text{ rot} &= \mathbf{v}_\perp \cos\theta + (\mathbf{k} \times \mathbf{v}) \sin\theta \\ &= [\mathbf{v} - (\mathbf{k} \cdot \mathbf{v})\mathbf{k}] \cos\theta + (\mathbf{k} \times \mathbf{v}) \sin\theta \end{aligned}$$

After the rotation of $\mathbf{v}_\perp$ about $\mathbf{k}$, we add back $\mathbf{v}_\parallel$ to get the rotation for $\mathbf{v}$:

$$\begin{aligned} \mathbf{v}_{\text{rot}} &= \mathbf{v}_\parallel + \mathbf{v}_\perp \text{rot} \\ &= \mathbf{v}_\parallel + [\mathbf{v} - (\mathbf{k} \cdot \mathbf{v})\mathbf{k}] \cos\theta + (\mathbf{k} \times \mathbf{v}) \sin\theta \\ &= (\mathbf{k} \cdot \mathbf{v})\mathbf{k} + [\mathbf{v} - (\mathbf{k} \cdot \mathbf{v})\mathbf{k}] \cos\theta + (\mathbf{k} \times \mathbf{v}) \sin\theta \\ &= \mathbf{v} \cos\theta + (\mathbf{k} \times \mathbf{v}) \sin\theta + \mathbf{k}(\mathbf{k} \cdot \mathbf{v})(1 - \cos\theta) \end{aligned}$$

Fig. 14.11 gives a 3D view of the rotation of $\mathbf{v}$, resulting in $\mathbf{v}_{\text{rot}}$.

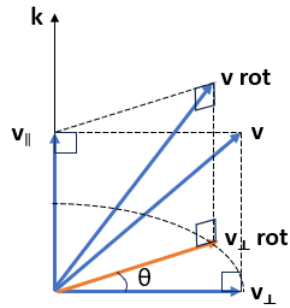


Figure 14.11. Rotating $\mathbf{v}$ by θ around $\mathbf{k}$

This rotation is implemented by rotateAround() in Vec.py:

```
def rotateAround(self, axis, theta):
  k = axis.normalize()
  cosA = math.cos(theta)
  sinA = math.sin(theta)
  vterm1 = self*cosA
  vterm2 = k.cross(self)*sinA
  vterm3 = k*(k.dot(self)*(1 - cosA))
  return vterm1 + vterm2 + vterm3  # rotated vector
```

There's a simple example of the effect of rotateAround() included in the output of the call to vec.py given above:

```
a = Vec(3.00, 4.00, 0.00)
   #  many more lines
Rotate 'a' by 90° around Z axis = Vec(-4.00, 3.00, 0.00)
```

The corresponding code is:

```
  a = Vec(3, 4, 0)
  zAxis = Vec(0, 0, 1)
  print("Rotate 'a' by 90° around Z axis =",
                a.rotateAround(zAxis, math.pi/2))
```

Apart from very basic examples, it can be difficult to decide if the vector calculated by rotateAround() is correct, which prompted me to implement VecPlot.py. It utilizes MatPlotlib to plot normalized vectors inside a 3D unit sphere. A visualization of the preceding example is achieved with the code:

```
a = Vec(3, 4, 5).normalize()
zAxis = Vec(0, 0, 1)
b = a.rotateAround(zAxis, math.pi/2)

plotter = VecPlot("Rotate 'a' by 90 degs around Z axis")
plotter.addVector(a, label='a', color='blue')
plotter.addVector(b, label='b', color='red')
plotter.show()
```

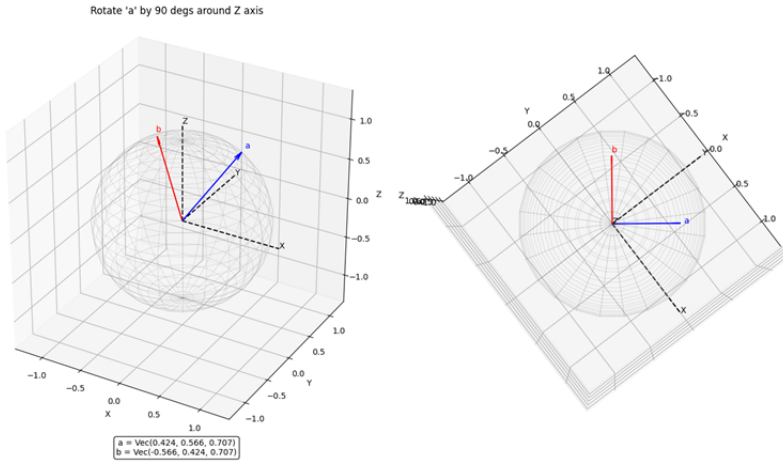
The resulting plot is shown twice in Fig. 14.12.

14.6 Quaternions

A quaternion is defined as

$$\mathbf{q} = (w, x, y, z) = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

but is often written as $\mathbf{q} = s + \mathbf{v}$, where s is a scalar corresponding to the w component, and $\mathbf{v}$ is a vector holding the x, y, z elements.



(a) Original Plot (b) View Looking Down the Z Axis

Figure 14.12. Rotate 'a' by 90° around the z-axis

Quaternion multiplication employs the following rules for $\mathbf{i}, \mathbf{j}$ and $\mathbf{k}$:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1,$$

$$\mathbf{ij} = -\mathbf{ji} = \mathbf{k}, \quad \mathbf{jk} = -\mathbf{kj} = \mathbf{i}, \quad \mathbf{ki} = -\mathbf{ik} = \mathbf{j}.$$

The presence of $\mathbf{i}, \mathbf{j}$ and $\mathbf{k}$ means that we can use operations like cross products and dot products which simplify quaternion multiplication quite a bit. For instance, for $\mathbf{q_1} = s_1 + \mathbf{v_1}$ and $\mathbf{q_2} = s_2 + \mathbf{v_2}$, we have:

$$\begin{aligned}
\mathbf{q_1 q_2} &= (s_1 s_2 - \mathbf{v_1} \cdot \mathbf{v_2}) + (s_1 \mathbf{v_2} + s_2 \mathbf{v_1} + \mathbf{v_1} \times \mathbf{v_2}) \\
\mathbf{q_2 q_1} &= (s_1 s_2 - \mathbf{v_1} \cdot \mathbf{v_2}) + (s_2 \mathbf{v_1} + s_1 \mathbf{v_2} + \mathbf{v_2} \times \mathbf{v_1}) \\
&= (s_1 s_2 - \mathbf{v_1} \cdot \mathbf{v_2}) + (s_1 \mathbf{v_2} + s_2 \mathbf{v_1} - \mathbf{v_1} \times \mathbf{v_2})
\end{aligned}$$

The second multiplication differs due to the non-commutativity of the cross product.

The crucial step in quaternion multiplication is how

$$\begin{aligned}
\mathbf{q_1 q_2} &= (s_1 + \mathbf{v_1})(s_2 + \mathbf{v_2}) \\
&= s_1 s_2 + s_1 \mathbf{v_2} + s_2 \mathbf{v_1} + \mathbf{v_1 v_2}
\end{aligned}$$

becomes

$$\mathbf{q_1 q_2} = s_1 s_2 + s_1 \mathbf{v_2} + s_2 \mathbf{v_1} - \mathbf{v_1} \cdot \mathbf{v_2} + \mathbf{v_1} \times \mathbf{v_2}$$

In other words, how can the quaternion multiplication of $\mathbf{v_1}$ and $\mathbf{v_2}$ be expressed in terms of the dot and cross products:

$$\mathbf{v_1 v_2} = -\mathbf{v_1} \cdot \mathbf{v_2} + \mathbf{v_1} \times \mathbf{v_2}$$

$\mathbf{v_1}$ and $\mathbf{v_2}$ can be written in full as:

$$\begin{aligned}\mathbf{v_1} &= a_1\mathbf{i} + b_1\mathbf{j} + c_1\mathbf{k}\\ \mathbf{v_2} &= a_2\mathbf{i} + b_2\mathbf{j} + c_2\mathbf{k}\end{aligned}$$

Their product is:

$$\begin{aligned}\mathbf{v_1}\mathbf{v_2} &= (a_1\,\mathbf{i} + b_1\,\mathbf{j} + c_1\,\mathbf{k})(a_2\,\mathbf{i} + b_2\,\mathbf{j} + c_2\,\mathbf{k})\\ &= a_1a_2\mathbf{ii} + a_1b_2\mathbf{ij} + a_1c_2\mathbf{ik} +\\ &\quad b_1a_2\mathbf{ji} + b_1b_2\mathbf{jj} + b_1c_2\mathbf{jk} +\\ &\quad c_1a_2\mathbf{ki} + c_1b_2\mathbf{kj} + c_1c_2\mathbf{kk}\end{aligned}$$

The quaternion multiplication rules simplify this to:

$$\begin{aligned}&= a_1a_2(-1) + a_1b_2\mathbf{k} + a_1c_2(-\mathbf{j}) +\\ &\quad b_1a_2(-\mathbf{k}) + b_1b_2(-1) + b_1c_2\mathbf{i} +\\ &\quad c_1a_2\mathbf{j} + c_1b_2(-\mathbf{i}) + c_1c_2(-1)\end{aligned}$$

The scalar and vector elements can be grouped together. The scalar part:

$$-a_1a_2 - b_1b_2 - c_1c_2 = -(a_1a_2 + b_1b_2 + c_1c_2) = -\mathbf{v_1}\cdot\mathbf{v_2}$$

The vector part can be organized into its $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ components:

$$(b_1c_2 - c_1b_2)\mathbf{i} + (c_1a_2 - a_1c_2)\mathbf{j} + (a_1b_2 - b_1a_2)\mathbf{k} = \mathbf{v_1}\times\mathbf{v_2}$$

The left side of this equality can be written as the pseudo-determinant:

$$\begin{vmatrix}\mathbf{i} & \mathbf{j} & \mathbf{k}\\ a_1 & a_2 & a_3\\ b_1 & b_2 & b_3\\ c_1 & c_2 & c_3\end{vmatrix}$$

which highlights its equivalence to the cross product of $\mathbf{v_1}$ and $\mathbf{v_2}$.

Combining the two parts:

$$\mathbf{v_1}\mathbf{v_2} = -\mathbf{v_1}\cdot\mathbf{v_2} + \mathbf{v_1}\times\mathbf{v_2}$$

A quaternion without a scalar is *pure*, and multiplication of pure quaternions can be simplified. When the pure quaternions $\mathbf{v}$ and $\mathbf{u}$ are multiplied together they become:

$$\begin{aligned}\mathbf{vu} &= -\mathbf{v}\cdot\mathbf{u} + \mathbf{v}\times\mathbf{u}\\ \mathbf{uv} &= -\mathbf{u}\cdot\mathbf{v} + \mathbf{u}\times\mathbf{v}\\ &= -\mathbf{v}\cdot\mathbf{u} - \mathbf{v}\times\mathbf{u}\end{aligned}$$

These expressions can be utilized to define the dot product and cross product in terms of the pure quaternion product:

$$\mathbf{v} \times \mathbf{u} = \frac{1}{2}(\mathbf{vu} - \mathbf{uv})$$
$$\mathbf{v} \cdot \mathbf{u} = -\frac{1}{2}(\mathbf{vu} + \mathbf{uv})$$

The *conjugate* of a quaternion $\mathbf{q} = s + \mathbf{v}$, denoted $\bar{\mathbf{q}}$, is $s - \mathbf{v}$. The product of a quaternion $\mathbf{q}$ with its conjugate equals the dot product of $\mathbf{q}$ with itself, and also the square of its magnitude:

$$\mathbf{q}\bar{\mathbf{q}} = \bar{\mathbf{q}}\mathbf{q} = \mathbf{q} \cdot \mathbf{q} = \|\mathbf{q}\|^2 = q^2.$$

This square, usually called the *norm*, can also be expressed as:

$$N\mathbf{q} = \mathbf{q}\bar{\mathbf{q}} = w^2 + x^2 + y^2 + z^2$$

The conjugate can be utilized to define the inverse of a nonzero quaternion:

$$\mathbf{q}^{-1} = \frac{\bar{\mathbf{q}}}{\|\mathbf{q}\|^2}.$$

This can be utilized with the magnitude to obtain:

$$\mathbf{q}\mathbf{q}^{-1} = \frac{\mathbf{q}\bar{\mathbf{q}}}{\|\mathbf{q}\|^2} = 1,$$

and similarly $\mathbf{q}^{-1}\mathbf{q} = 1$.

A *unit quaternion* $\hat{\mathbf{q}}$ is one where $N\mathbf{q} = 1$. If we substitute this into the formula for the inverse, we obtain $\mathbf{q}^{-1} = \hat{\mathbf{q}}$.

$\hat{\mathbf{q}}$ can also be represented by

$$\hat{\mathbf{q}} = \cos\theta + \hat{\mathbf{u}}\sin\theta$$

where $\hat{\mathbf{u}}$ is a unit vector. Also, Euler's formula for complex numbers

$$e^{ix} = \cos x + i \sin x$$

generalizes to unit quaternions:

$$e^{\hat{\mathbf{u}}\theta} = \cos\theta + \hat{\mathbf{u}}\sin\theta = \hat{\mathbf{q}}.$$

From this we can define the power of a unit quaternion:

$$\hat{\mathbf{q}}^t = (\cos\theta + \hat{\mathbf{u}}\sin\theta)^t = \exp(t\hat{\mathbf{u}}\theta) = \cos(t\theta) + \hat{\mathbf{u}}\sin(t\theta).$$

and the logarithm:

$$\log(\hat{\mathbf{q}}) = \log(\cos\theta + \hat{\mathbf{u}}\sin\theta) = \log(\exp(\hat{\mathbf{u}}\theta)) = \hat{\mathbf{u}}\theta.$$

14.7 Rotations with Quaternions

Suppose $\mathbf{k}$ is a unit vector and $\mathbf{v}$ some other vector, then rotating $\mathbf{v}$ counterclockwise around $\mathbf{k}$ by θ radians employs the unit quaternion $\hat{\mathbf{p}}$ (which we'll write as p in this section):

$$p = \cos(\theta/2) + \sin(\theta/2)\,\mathbf{k}$$

p is a unit quaternion because

$$p = \cos\left(\frac{\theta}{2}\right) + 0 \cdot \mathbf{i} + 0 \cdot \mathbf{j} + \sin\left(\frac{\theta}{2}\right)\mathbf{k}$$

So:

$$\|p\|^2 == \cos^2\left(\frac{\theta}{2}\right) + \sin^2\left(\frac{\theta}{2}\right) = 1$$

The rotation of $\mathbf{v}$ utilizes quaternion multiplication with p and its conjugate:

$$\mathbf{v}_{\text{Rot}} = p\mathbf{v}p^{-1} = p\mathbf{v}\bar{p}$$

Proving $\mathbf{v}_{\text{rot}} = p\mathbf{v}\bar{p}$ involves rewriting the right-hand side to produce Rodrigues' rotation formula, with the help of some trigonometry identities:

(a) $\cos(2x) = \cos^2 x - \sin^2 x$

(b) $\sin(2x) = 2\sin x \cos x$

(c) $2\sin^2 x = 1 - \cos(2x)$

(d) $\mathbf{u} \times \mathbf{v} = \frac{1}{2}(\mathbf{uv} - \mathbf{vu})$. (This expression was described in the previous section.)

(e) $\hat{\mathbf{u}}\mathbf{v}\hat{\mathbf{u}} = -2(\hat{\mathbf{u}} \cdot \mathbf{v})\hat{\mathbf{u}} + \mathbf{v}$. Since $\hat{\mathbf{u}} \cdot \mathbf{v} = -\frac{1}{2}(\hat{\mathbf{u}}\mathbf{v} + \mathbf{v}\hat{\mathbf{u}})$, then $\hat{\mathbf{u}}\mathbf{v} = -2\hat{\mathbf{u}} \cdot \mathbf{v} - \mathbf{v}\hat{\mathbf{u}}$, and so:

$$\begin{aligned}
\hat{\mathbf{u}}\mathbf{v}\hat{\mathbf{u}} &= [-2\hat{\mathbf{u}} \cdot \mathbf{v} - \mathbf{v}\hat{\mathbf{u}}]\hat{\mathbf{u}} \\
&= -2(\hat{\mathbf{u}} \cdot \mathbf{v})\hat{\mathbf{u}} - \mathbf{v}\hat{\mathbf{u}}\hat{\mathbf{u}} \\
&= -2(\hat{\mathbf{u}} \cdot \mathbf{v})\hat{\mathbf{u}} - \mathbf{v}(\hat{\mathbf{u}} \times \hat{\mathbf{u}} - \hat{\mathbf{u}} \cdot \hat{\mathbf{u}}) \\
&= -2(\hat{\mathbf{u}} \cdot \mathbf{v})\hat{\mathbf{u}} - \mathbf{v}(-1) \\
&= -2(\hat{\mathbf{u}} \cdot \mathbf{v})\hat{\mathbf{u}} + \mathbf{v}
\end{aligned}$$

With these identities, we can expand and rewrite $p\mathbf{v}\bar{p}$ relatively easily. The labeled brackets refer to particular identities listed above.

$$
\begin{aligned}
p\mathbf{v}\bar{p} &= (\cos(\theta/2) + \sin(\theta/2)\mathbf{k})\,\mathbf{v}\,(\cos(\theta/2) - \sin(\theta/2)\mathbf{k}) \\
&= \cos^2(\theta/2)\mathbf{v} + \sin(\theta/2)\cos(\theta/2)\,(\mathbf{kv} - \mathbf{vk}) - \sin^2(\theta/2)\,\mathbf{kvk} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (d) \qquad\qquad (e) \\
&= \cos^2(\theta/2)\mathbf{v} + \sin(\theta/2)\cos(\theta/2)2(\mathbf{k}\times\mathbf{v}) - \sin^2(\theta/2)(-2(\mathbf{k}\cdot\mathbf{v})\mathbf{k} + \mathbf{v}) \\
&= (\cos^2(\theta/2) - \sin^2(\theta/2))\,\mathbf{v} + 2\sin(\theta/2)\cos(\theta/2)(\mathbf{k}\times\mathbf{v}) + 2\sin^2(\theta/2)(\mathbf{k}\cdot\mathbf{v})\mathbf{k} \\
&\qquad\quad (a) \qquad\qquad\quad (b) \qquad\qquad\quad (c) \\
&= (\cos\theta)\,\mathbf{v} + (\sin\theta)(\mathbf{k}\times\mathbf{v}) + (1 - \cos\theta)(\mathbf{k}\cdot\mathbf{v})\mathbf{k} \\
&= \mathbf{v}_{\text{Rot}}
\end{aligned}
$$

In other words, quaternion rotation is equivalent to Rodrigues' rotation. This may make you wonder why we should bother with quaternions at all. Although the two techniques are the same in the math sense, quaternions offer several practical advantages:

(1) **Numerical stability**. When the rotation angle is very small, numerical errors are more likely when using axis-angles. For instance, the direction of the rotation may suddenly change.

(2) **Simple composition**. The composition of multiple axis-angles requires them to be converted into matrices so they can be multiplied, which is both slower and more complex than quaternion composition.

(3) **Interpolation**. The interpolation of axis-angles between two orientations can generate incorrect results due to numerical issues, resulting in extraneous rotations or interpolations that don't follow the shortest path between the start and end points. Quaternions are very well suited to interpolation, as following sections will illustrate.

14.8 A Quaternion class in Python

Our Python quaternion class, `Quat.py`, utilizes `Vec.py` to represent a quaternion as a scalar plus a vector. Quat.py offers a range of basic arithmetic and other operations for quaternions, as well as an extensive range of interpolation methods that we'll describe in later sections. Quat's UML class diagram is given in Fig. 14.13.

Quat.py contains a simple test-rig which rotates an x-axis vector to point along the y-axis by employing a quaternion rotation of 90° around the z-axis. The code:

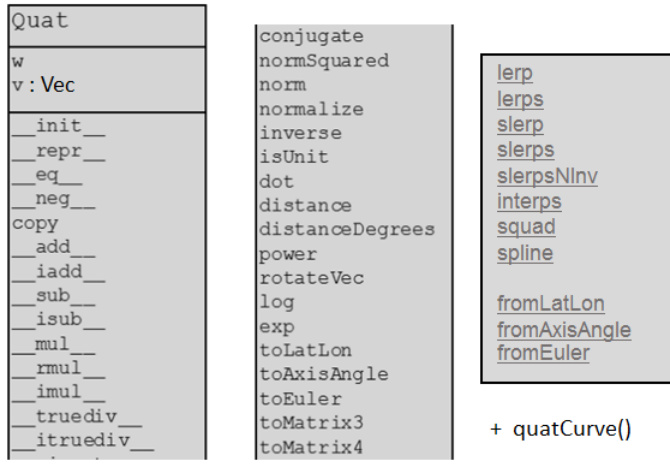


Figure 14.13. Quat Class Diagram

```
  axis = Vec(0, 0, 1)  # z-axis
  print("Create quat from Axis, Angle:\n", axis, ",", 90)
  p = Quat.fromAxisAngle(axis, math.pi/2)
  print("Result:", p)
  v1 = Vec(1, 0, 0) # x-axis
  v2 = p.rotateVec(v1)
  print(f"Quat rotates {v1} --> {v2}")
```

The output:

```
Create quat from Axis, Angle:
 Vec(0.000, 0.000, 1.000) , 90
Result: Quat(0.707, Vec(0.000, 0.000, 0.707))
Quat rotates Vec(1.000, 0.000, 0.000) --> Vec(0.000, 1.000, 0.000)
```

The p unit quaternion is created from the axis-angle by fromAxisAngle():

```
  @staticmethod
  def fromAxisAngle(axis, angleRad):
    half = angleRad/2
    return Quat(math.cos(half),
                axis.normalize()*math.sin(half))
```

This is a translation of:

$$p = \cos(\theta/2) + \sin(\theta/2)\,\mathbf{k}$$

Note that we ensure that the vector (called `axis` in the code) is normalized.

The rotation of a vector (`vec`) is achieved by passing it to p's rotateVec():

```
  def rotateVec(self, vec):
    if not isinstance(vec, Vec):
      raise TypeError("Input must be a Vec instance")
    if not self.isUnit(1e-6):
      print("Warning: Quaternion is not unit length.")

    v = Quat(0, vec)
    return (self * v * self.conjugate()).v
```

Aside from the error checking, the last line implements:

$$\mathbf{v}_{\text{Rot}} = p\mathbf{v}\bar{p}$$

The VecPlot class can be employed to show rotations like the one above. QuatPlot.py contains several examples separated into functions. quats1() replicates the example above:

```
def quats1(plotter):
  v1 = Vec(1, 0, 0)
  q = Quat.fromAxisAngle(z, math.pi/2)
  v2 = q.rotateVec(v1)
  print(f"Rotate {v1} around {q} to {v2}")

  plotter.addVector(v1, label='v1', color='blue')
  plotter.addVector(v2, label='v2', color='red')
  axis, angle =  Quat.toAxisAngle(q)
  ang = math.degrees(angle)
  plotter.addVector(axis, label='q('+str(ang) +')', color='green')
```

The result is shown in Fig. 14.14.

The only new element here is that toAxisAngle() is used to convert the quaternion back into an {axis, angle} pair so the axis can be drawn as an arrow and the angle written as part of its label.

A more complex example is implemented in quats2(). We want to rotate $\mathbf{v} = 2\mathbf{i} + 1\mathbf{j}$ by $\theta = \pi/3$ radians about $\mathbf{u} = \frac{1}{\sqrt{2}}\mathbf{j} + \frac{1}{\sqrt{2}}\mathbf{k}$. This is implemented by:

```
def quats2(plotter):
  v1 = Vec(2, 1, 0)
  u = Vec(0, 1/math.sqrt(2), 1/math.sqrt(2))
  # q = Quat(math.cos(math.pi/6),
             # u * math.sin(math.pi/6))
  q = Quat.fromAxisAngle(u, math.pi/3)
  v2 = q.rotateVec(v1)
  print(f"Rotate {v1} around {q} to {v2}")

  axis, angle = q.toAxisAngle()
  ang = round(math.degrees(angle), 1)
  plotter.addVector(v1, label='v1', color='blue')
```

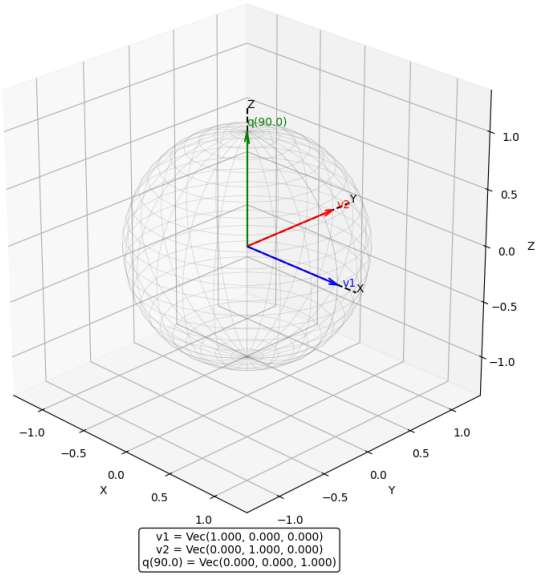


Figure 14.14. Plotting a 90° Rotation around a Quaternion

```
plotter.addVector(v2, label='v2', color='red')
plotter.addVector(axis, label='Ax:'+str(ang), color='green')
```

Note that we've commented out a slightly longer way to define the quaternion since using fromAxisAngle() is more intuitive. The plot is shown in Fig. 14.15.

We've rotated the plot in Fig. 14.15 to better highlight the 60° rotation. Also note that the vectors are all displayed in normalized form since the sphere has unit radius.

14.9 Spherical Linear Interpolation (Slerp)

Since quaternions can be represented by vectors, they're easy to animate through intermediate orientations via interpolation. The simplest approach is *linear interpolation* which utilizes two unit quaternions $\mathbf{q}_0$ and $\mathbf{q}_1$ in

$$\mathbf{q}(t) = (1-t)\mathbf{q}_0 + t\mathbf{q}_1.$$

$\mathbf{q}(t)$ changes smoothly along the line segment connecting $\mathbf{q}_0$ and $\mathbf{q}_1$ as t varies from 0 to 1 (see Fig. 14.16).

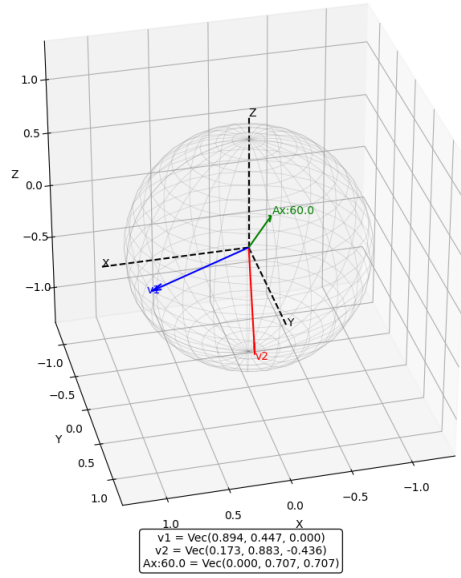


Figure 14.15. Plotting a 60° Rotation around a Quaternion

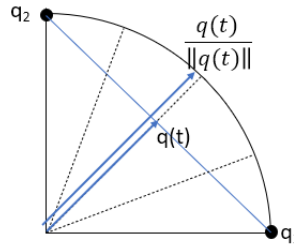


Figure 14.16. Linear Interpolation

Fig. 14.16 also illustrates that $\mathbf{q}(t)$ doesn't maintain a unit length as it moves, but this can be remedied by normalization:

$$\mathbf{q}(t) = \frac{(1-t)\mathbf{q}_0 + t\mathbf{q}_1}{\|(1-t)\mathbf{q}_0 + t\mathbf{q}_1\|}.$$

The implementation of Lerp in `Quat.py` follows this definition:

```
@staticmethod
def lerp(q0, q1, t):
  t = max(0, min(1, t))
  return ((q0 * (1 - t)) + (q1 * t)).normalize()
```

The dotted lines in Fig. 14.16 divide the straight-line route into four equal quarter lengths with a trip time for each equal to 1/4 of the total. However, after normalization the four quarter-arcs are no longer equal in length, being noticeably shorter at the two ends. Consequently, the object speeds up during the middle part of the interpolation.

What we really want is a function that interpolates between the quaternions, preserves unit length, and sweeps through the angle between them at a *constant* rate. We'll utilize Fig. 14.17 to define these requirements more mathematically.

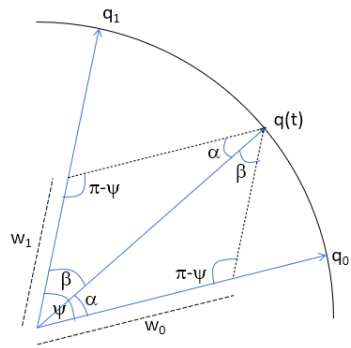


Figure 14.17. Spherical Linear Interpolation

Given the unit quaternions $\mathbf{q}_0$, $\mathbf{q}_1$, and parameter t:

$$\psi = \cos^{-1}(\mathbf{q}_0 \cdot \mathbf{q}_1)$$

This assumes that the quaternion dot product is defined in the same way as the dot product of 4D vectors:

$$\mathbf{q}_0 \cdot \mathbf{q}_1 \;=\; |\mathbf{q}_0||\mathbf{q}_1|\cos(\psi)$$

where ψ is the angle between them. However, since these are unit quaternions, the equation simplifies to:

$$\mathbf{q}_0 \cdot \mathbf{q}_1 = \cos(\psi)$$

Using ψ:

$$\alpha = t\psi; \quad \beta = (1-t)\psi$$

The function calculates a weighted sum:

$$\mathbf{q}(t) = w_0\mathbf{q}_0 + w_1\mathbf{q}_1$$

This can be rewritten by applying the Law of Sines to the triangles in Fig. 14.17:

$$\frac{\sin\alpha}{w_1} = \frac{\sin\beta}{w_0} = \frac{\sin(\pi-\psi)}{1} = \sin\psi$$

and so

$$w_0 = \sin\beta/\sin\psi \quad w_1 = \sin\alpha/\sin\psi$$

The interpolation function becomes:

$$\mathbf{q}(t) = \frac{\mathbf{q}_0\sin\beta + \mathbf{q}_1\sin\alpha}{\sin\psi}$$

If we replace α and β:

$$\mathbf{q}(t) = \frac{\mathbf{q}_0\sin((1-t)\psi) + \mathbf{q}_1\sin(t\psi)}{\sin\psi}$$

The Slerp implementation in :

```
@staticmethod
def slerp(q0, q1, t):
  t = max(0, min(1, t))
  q0 = q0.normalize()
  q1 = q1.normalize()

  # get cosine of angle between quats
  dotCos = q0.dot(q1)
  if dotCos < 0:
    # negate q1 to take shorter path
    q1 = -q1
    dotCos = -dotCos

  # Clamp dotCos to avoid numerical issues with acos
  dotCos = max(-1.0, min(1.0, dotCos))

  # If quaternions are very close, use linear interpolation
  if dotCos > 0.9995:
    return Quat.lerp(q0, q1, t)

  # Use more numerically stable formulation
  theta = math.acos(dotCos)
  sinTheta = math.sin(theta)
  t_theta = theta * t
  s0 = math.cos(t_theta) - (math.sin(t_theta) * dotCos)/sinTheta
  s1 = math.sin(t_theta) / sinTheta
  return (s0*q0 + s1*q1).normalize()
```

One complication is that as ψ approaches zero (called `theta` in the code), the estimation may lose accuracy, so the implementation switches to linear interpolation.

The interpolation at the end of the function uses a numerically more stable version of $\sin(\psi(1-t))/\sin\psi$:

$$\begin{aligned}\frac{\sin\psi(1-t)}{\sin\psi} &= \frac{\sin(\psi-\psi t)}{\sin\psi}\\ &= \frac{\sin\psi\cdot\cos\psi t - sin\psi t\cdot\cos\psi}{\sin\psi}\\ &= \cos\psi t - \frac{\sin\psi t\cdot\cos\psi}{\sin\psi}\end{aligned}$$

In summary:

$$\text{Slerp}(\mathbf{q}_0,\mathbf{q}_1,t) = (\cos\psi t - \frac{\sin\psi t\cdot\cos\psi}{\sin\psi})\mathbf{q}_0 + \frac{\sin(\psi t)}{\sin\psi}\mathbf{q}_1$$

Slerp() has the following properties:

(a) When $t=0$, $\text{Slerp}(\mathbf{q}_0,\mathbf{q}_1,0)=\mathbf{q}_0$.

(b) When $t=1$, $\text{Slerp}(\mathbf{q}_0,\mathbf{q}_1,1)=\mathbf{q}_1$.

(c) For all t, $|\text{Slerp}(\mathbf{q}_0,\mathbf{q}_1,t)|=1$

(d) Slerp()'s speed remains constant, which can be expressed as:

$$\left|\frac{d}{dt}\text{Slerp}(\mathbf{q}_0,\mathbf{q}_1,t)\right| = |\psi|$$

(a) and (b) can be shown by plugging $t=0$ and $t=1$ into the function. Proving (c) and (d) are a bit more involved, but here are two hints. The proof of (c) utilizes the fact that for unit vectors $\mathbf{v}$ and $\mathbf{w}$ we have:

$$\begin{aligned}|\alpha\mathbf{v}+\beta\mathbf{w}|^2 &= (\alpha\mathbf{v}+\beta\mathbf{w})\cdot(\alpha\mathbf{v}+\beta\mathbf{w}) = \alpha^2\mathbf{v}\cdot\mathbf{v} + 2\alpha\beta\mathbf{v}\cdot\mathbf{w} + \beta^2\mathbf{w}\cdot\mathbf{w}\\ &= \alpha^2 + 2\alpha\beta\cos\psi + \beta^2\end{aligned}$$

This expression comes in useful for rewriting the right-hand side of the Slerp() equation.

The proof of (d) starts with the observation:

$$\begin{aligned}\text{Slerp}(\mathbf{q}_0,\mathbf{q}_t,t) &= \frac{\sin(\psi(1-t))}{\sin\psi}\mathbf{q}_0 + \frac{\sin(\psi t)}{\sin\psi}\mathbf{q}_1\\ &= \frac{1}{\sin\psi}[\sin(\psi(1-t))\mathbf{q}_0 + \sin(\psi t)\mathbf{q}_1]\end{aligned}$$

The derivative is therefore:

$$\frac{d}{dt}\text{Slerp}(\mathbf{q}_0,\mathbf{q}_t,t) = \frac{1}{\sin\psi}[-\psi\cos(\psi(1-t))\mathbf{q}_0 + \psi\cos(\psi t)\mathbf{q}_1]$$

If you're still in the dark, this approach is based on material prepared by Justin Wyss-Gallifent for his course on 'Mathematics and Geometry for Computer Graphics' available at `https://www.math.umd.edu/~immortal/MATH431/`.

14.10 Visualizing Lerp and Slerp

As we've seen, normalized linear interpolation (Lerp) has the drawback that it doesn't trace out the arc between the starting and ending quaternions at a constant rate. This is remedied by switching to spherical linear interpolation (Slerp). One way to visualize this difference is to calculate a series of interpolated quaternions using both approaches, and plot their turning angles.

interps.py sets the starting and ending quaternions to be a y-axis vector with a initial rotation of 0° and a final rotation of 180°. Their Lerp and Slerp interpolations are divided into 100 steps, and the resulting angles relative to a reference direction are plotted in Fig. 14.18.

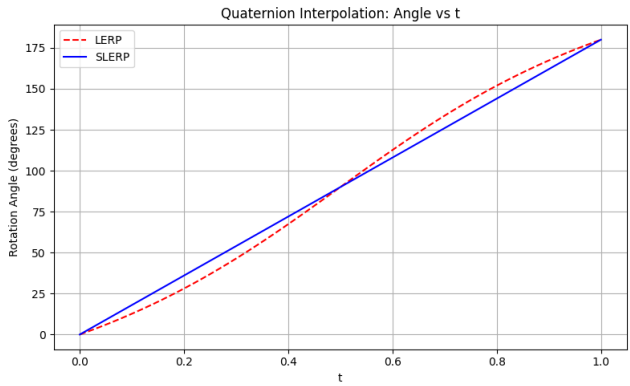


Figure 14.18. Angle Interpolations for Lerp and Slerp

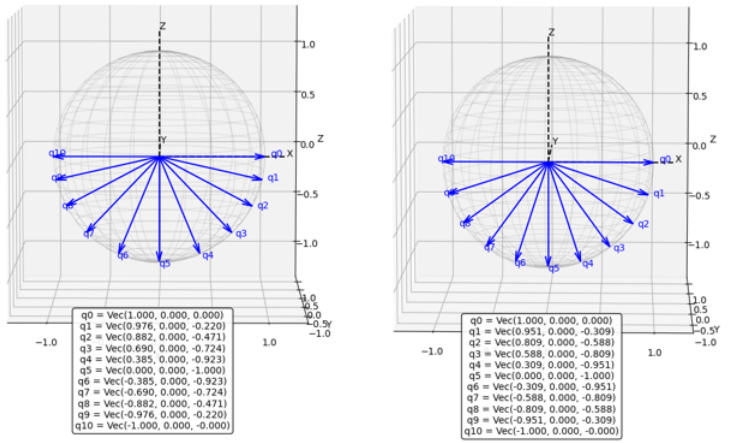
The solid blue Slerp line is linear, meaning that the rate of change of its rotation angle is constant, whereas the Lerp curve shows the angle changing relatively slow at the endpoints, and fastest in the middle (when $t = 0.5$).

The code that generates this data:

```
# two quaternions: 0° and 180° rotation about Y-axis
q1 = Quat.fromAxisAngle(Vec(0, 1, 0), 0)
q2 = Quat.fromAxisAngle(Vec(0, 1, 0), math.pi)
vRef = Vec(1, 0, 0)  # Reference dir for angle calc
tValues = []
angleLerp = [] ; angleSlerp = []
for i in range(101):
  t = i / 100
  qLerp = Quat.lerp(q1, q2, t)
  qSlerp = Quat.slerp(q1, q2, t)
  vL = qLerp.rotateVec(vRef)
  vS = qSlerp.rotateVec(vRef)
```

```
  tValues.append(t)
  angleLerp.append( vRef.angleToDeg(vL))
  angleSlerp.append( vRef.angleToDeg(vS))
```

Another visualization approach is to use the VecPlot class to display the Lerp and Slerp interpolations. This is done by the lerpShow() and slerpShow() examples in QuatPlot.py. The starting and ending quaternions are the same as before: 0° and 180° rotations about the y-axis, but only nine intermediate quaternions are generated. These are used to rotate a vector originally pointing along the x-axis, and the results are drawn in Fig. 14.19.



a) Using Lerp b) Using Slerp

Figure 14.19. Lerp vs. Slerp Interpolation

The most visible difference is around the middle of the interpolations.

lerpShow() and slerpShow() are very similar apart from their calls to Quat.lerp() and Quat.slerp(). slerpShow() is:

```
def slerpShow(plotter):
  q1 = Quat.fromAxisAngle(Vec(0, 1, 0), 0)
  q2 = Quat.fromAxisAngle(Vec(0, 1, 0), math.pi)
  n = 10
  steps = [t/n for t in range(n+1)]
  for i, t in enumerate(steps):
    qt = Quat.slerp(q1, q2, t)
    v = qt.rotateVec(Vec(1, 0, 0)) # x-axis vec
    plotter.addVector(v, label='q'+str(i), color='blue')
```

14.11 Navigating Again

Section 13 on 'Navigating the Earth' discusses spherical trigonometry using the problem of navigating over the Earth's surface (assuming that it's a feature-less sphere). The two main types of curve we encountered were Great Circles and Rhumb Lines, and these were illustrated in Python with the help of the Cartopy mapping module (`https://scitools.org.uk/cartopy/docs/latest/`).

When Slerp() interpolates between two quaternions it traces out the shortest path between them in a Great Circle arc. If these quaternions are converted into latitude and longitude coordinates, they can be used to plot a path between two locations on the Earth.

The conversion of a latitude and longitude into a quaternion assumes that the sphere (see Fig. 14.20) employs an (x, y, z) coordinate system centered around O, with polar coordinates (ρ, λ, ψ). ρ is the length of the point's vector (which we'll treat as a unit radius in the code), λ corresponds to its longitude, while ψ is its inclination, which is equivalent to $\pi/2 - \phi$ (ϕ is the point's latitude).

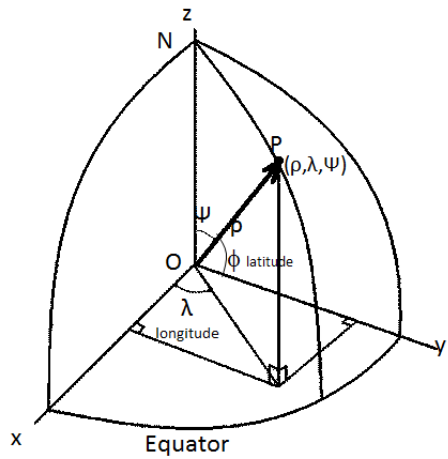


Figure 14.20. 3D Polar and Cartesian Coordinates

The Cartesian coordinates can be expressed as:

$$x = \rho \sin\psi \cos\lambda, \quad y = \rho \sin\psi \sin\lambda \quad z = \rho \cos\psi$$

ψ can be replaced by the latitude ϕ:

$$\begin{array}{rcccl} x & = & \rho \sin(\pi/2 - \phi) \cos\lambda & = & \rho \cos\phi \cos\lambda \\ y & = & \rho \sin(\pi/2 - \phi) \sin\lambda & = & \rho \cos\phi \sin\lambda \\ z & = & \rho \cos(\pi/2 - \phi) & = & \rho \sin\phi \end{array}$$

Since (x, y, z) is a unit vector, it becomes the pure quaternion $q = [0, (x, y, z)]$.

This is implemented by fromLatLon() in Quat.py:

```
@staticmethod
def fromLatLon(latDeg, lonDeg):
  latRad = math.radians(latDeg)
  lonRad = math.radians(lonDeg)
  x = math.cos(latRad) * math.cos(lonRad)
  y = math.cos(latRad) * math.sin(lonRad)
  z = math.sin(latRad)
  return Quat(0, Vec(x, y, z))
```

QuatOrtho.py draws a Great Circle arc between two cities on an Orthographic map, and adds Lerp and Slerp interpolated points along the arc. The curve of the arc is drawn in two ways – using Cartopy and with greatCircleCoords() from our nav.py. Fig. 14.21 shows the map generated for a trip between New York and Cape Town.

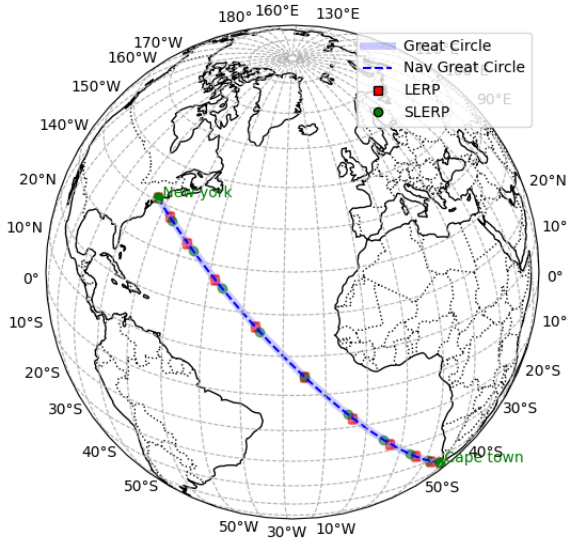


Figure 14.21. From New York to Cape Town

It's possible to call QuatOrtho.py with additional arguments to switch off the Lerp or Slerp points, which makes it easier to judge the arc distances.

Inside QuatOrtho.py the Lerp and Slerp points are generated in a similar way, so we'll focus on just the Slerp code. The interpolated quaternions are generated using:

```
quats = Quat.slerpsNInv(qStart, qEnd, markStep)
coords = [ q.toLatLon() for q in quats]
plotMarkers(ax, coords, "o", "green")
```

slerpsNInv() calls Quat.slerpNInv() multiple times with different t values to generate a list of interpolated quaternions. slerpNInv() is a simplified version of slerp() which doesn't perform quaternion inversion to switch from a long path to a short one, which isn't needed here.

toLatLon() converts a quaternion of the form $q = [0, (x, y, z)]$ back to a latitude and longitude pair. This is achieved by treating (x, y, z) as a Cartesian coordinate and converting it to polar values:

```
def toLatLon(self):
  axis = self.v
  r = axis.magnitude()
  if r == 0:
    raise ValueError("Quaternion has zero vector part")

  latDeg = math.degrees( math.asin(axis.z/r))
  lonDeg = math.degrees( math.atan2(axis.y, axis.x))
  return latDeg, lonDeg
```

14.12 Spherical Cubic Interpolation (Squad)

Slerp() produces smooth animations, but it always follows a Great Circle arc between its two quaternions. This means that a more complex path using a series of quaternions will usually be quite jagged, perhaps changing direction drastically at each quaternion. This is illustrated by TourSlerps.py which draws a series of Great Circles arcs between cities on an Orthographic map (see Fig. 14.22).

To smoothly interpolate through such a sequence, splines are needed, producing something similar to Fig. 14.23.

Consider four quaternions, **p**, **a**, **b**, and **q** as the vertices of a quadrilateral (see Fig. 14.24). **c** is interpolated along the 'edge' from **p** to **q** using Slerp(). At the same time, **d** is interpolated along the 'edge' from **a** to **b**. Finally, the edge between **c** and **d** is interpolated to get **e**, creating a *Squad* quaternion. This is defined like so:

$$\text{Squad}(t, \mathbf{p}, \mathbf{q}, \mathbf{a}, \mathbf{b}) = \text{Slerp}\big(2t(1-t),\ \text{Slerp}(t, \mathbf{p}, \mathbf{q}),\ \text{Slerp}(t, \mathbf{a}, \mathbf{b})\big)$$

The path from **p** to **q** is no longer the shortest (i.e. it's *not* a Great Curve arc), but curves toward the edge linking **a** and **b**.

Squad() is the quaternion equivalent of a cubic spline – it generates a smooth rotation interpolation through a sequence of quaternions by using tangents to

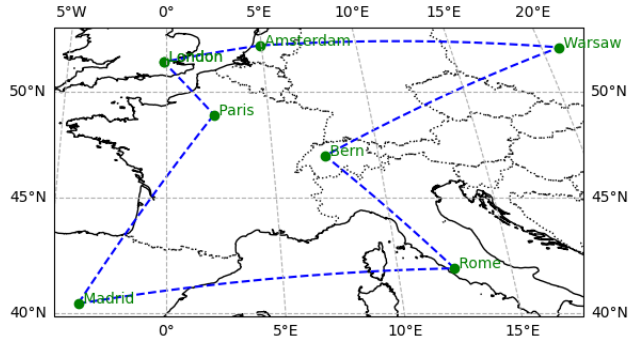


Figure 14.22. Slerps Cities Tour

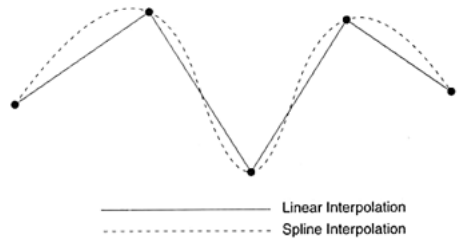


Figure 14.23. Linear vs. Spline Interpolation

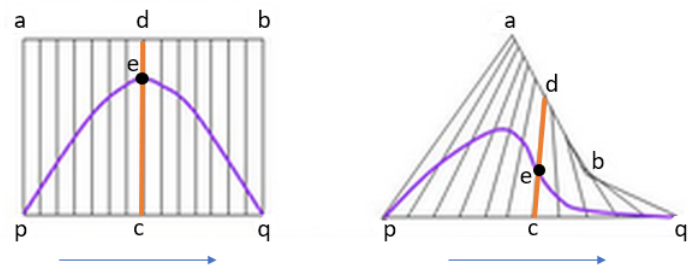


Figure 14.24. Two Examples of Spherical Cubic Interpolation

ensure that the change in direction and rotation between the quaternions is continuous.

Let $\{\mathbf{q}_n, \mathbf{a}_n, \mathbf{b}_n\}_{n=0}^{N-1}$ be a sequence of N quaternions, with

$$S_n(t) = \text{Squad}(t; \mathbf{q}_n, \mathbf{q}_{n+1}, \mathbf{a}_n, \mathbf{b}_n)$$

To ensure the smooth interpolations, $\mathbf{a}_n$ and $\mathbf{b}_n$ are calculated with:

$$\begin{aligned} \mathbf{a}_n &= \mathbf{q}_n \exp((-\log(\mathbf{q}_n^{-1}\mathbf{q}_{n-1}) + \log(\mathbf{q}_n^{-1}\mathbf{q}_{n+1}))/4) \\ \mathbf{b}_n &= \mathbf{a}_{n+1} \end{aligned}$$

$\mathbf{q}_n^{-1}\mathbf{q}_{n-1}$ is the rotation from $\mathbf{q}_n$ to $\mathbf{q}_{n-1}$, $\mathbf{q}_n^{-1}\mathbf{q}_{n+1}$ the rotation from $\mathbf{q}_n$ to $\mathbf{q}_{n+1}$, and log() maps them into vectors in the tangent space at $\mathbf{q}_n$.

A tangent space is a "flat" space that best approximates a curved surface at a given point, meaning that we can do vector math there without worrying about curvature. The unit quaternion $\mathbf{q}$ is a 3-sphere in 4D space, and Fig. 14.25 attempts to illustrate it in 3D.

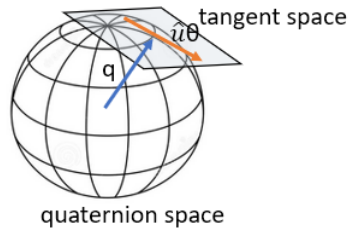


Figure 14.25. Quaternion and Tangent Spaces

After the forward and backward tangent vectors have been averaged and scaled, exp() maps the resulting tangent back to the quaternion sphere. The behavior of log() and exp() can be understood by writing the unit quaternion in Euler form:

$$\mathbf{q} = \cos\theta + \hat{\mathbf{u}}\sin\theta = \exp(\hat{\mathbf{u}}\theta)$$

where $\hat{\mathbf{u}}$ is the rotation axis and θ the rotation angle divided by 2. $\log(\mathbf{q})$ returns $\hat{\mathbf{u}}\theta$, an angular velocity vector that's a tangent to the quaternion.

A good mathematical description of both Slerp and Squad can be found in the online paper "Quaternion Algebra and Calculus" by David Eberly at `https://www.geometrictools.com/Documentation/Quaternions.pdf` [**Ebe10**]. An even more detailed study can be found in [**DKL98**].

The implementation of Squad in `Quat.py` is based around three functions: squad(), slerpNInv(), and spline(). squad() looks much like its math definition:

```
def squad(q0, q1, a, b, t):
  c = Quat.slerpNInv(q0, q1, t)
  d = Quat.slerpNInv(a, b, t)
  return Quat.slerpNInv(c, d, 2*t*(1 - t))
```

It's common for slerp() implementations to invert one of the input quaternions when the angle between the two exceeds 90°. Although it's true that $\mathbf{q}$ and $-\mathbf{q}$ represent the same rotation, Slerp$(t, \mathbf{p}, \mathbf{q})$ does not produce the same result as Slerp$(t, \mathbf{p}, -\mathbf{q})$. Since the control points $\mathbf{a}$ and $\mathbf{b}$ are chosen to work with positive values of $\mathbf{q}$, it's best not to invert the input quaternions in the version of slerp() used with squad().

spline() does the job of generating a control point $\mathbf{a}_n$ by using $\mathbf{q}_{n-1}$, $\mathbf{q}_n$, and $\mathbf{q}_{n+1}$:

```
  @staticmethod
  def spline(qPrev, q, qNext):
    qInv = q.inverse()
    delta = ((qInv * qPrev).log() +
             (qInv * qNext).log()) * -0.25
    return q * delta.exp()
```

Squad() works with a sequence of N quaternions, iterating through them using $\mathbf{q}_{n-1}$, $\mathbf{q}_n$, and $\mathbf{q}_{n+1}$ as n is incremented. This process is managed by quatCurve() located in Quat.py but not part of the Quat class. The code:

```
def quatCurve(quats, segSteps):
  if not quats:
      return []
  if len(quats) == 1:
      return [quats[0]]

  curve = []
  numSegs = len(quats) - 1
  for i in range(numSegs):
    q1 = quats[i]
    q2 = quats[i+1]

    # previous and next quaternions
    qPrev = quats[max(0, i-1)]  # 0 is min index
    qNext = quats[min(numSegs, i+2)] # numSegs is max index

    # control points 'a' and 'b'
    a = Quat.spline(qPrev, q1, q2)
    b = Quat.spline(q1, q2, qNext)

    # squad interpolation
    for step in range(segSteps):
      t = step / segSteps
      iquat = Quat.squad(q1, q2, a, b, t)
      curve.append(iquat)

  # Add last control point to ensure the curve ends
```

```
  # at the final quaternion.
  if numSegs > 0:
    curve.append(quats[-1])

  return curve
```

quatCurve() returns a list of interpolated quaternions using squad(). One tricky aspect is dealing with the beginning and end of the sequence of quaternions. In particular, how should the control points a_{-1} and b_N be determined? The simplest solution is to just use the nearest quaternions: q_0 and q_{N-1} respectively.

quatCurve() makes the implementation of the Squad-like version of the cities tour quite straight forward. TourSquad.py draws a series of curves (i.e. *not* Great Circles arcs) between the cities on an Orthographic map (see Fig. 14.26).

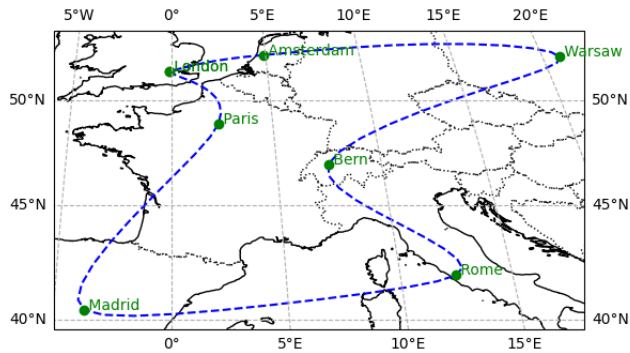


Figure 14.26. Squad Cities Tour

The relevant snippet of code from TourSquad.py is:

```
quats = [ Quat.fromLatLon(*coord) for coord in cityCoords]
iQuats = quatCurve(quats, 10)
iCoords = [ q.toLatLon() for q in iQuats]
```

`quats` holds the quaternions for the cities, which are used by quatCurve() to generate ten interpolated quaternions between each city, which are collected in the `iquats` list. This is converted back to a list of latitude and longitudes, and plotted on the map by Cartopy.

Exercises

(1) Given that $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1$, show that -1 commutes with $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$.

(2) Given that $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$, show that $\mathbf{jk} = \mathbf{i}$.

(3) Show that the real quaternions embed the complex numbers as Hamilton intended; that is

$$(a, b, 0, 0) + (c, d, 0, 0) = (a + c, b + d, 0, 0)$$

and

$$(a, b, 0, 0)(c, d, 0, 0) = (ac - bd, ad + bc, 0, 0).$$

(4) What quaternion represents a rotation through 60° about the x-axis?

(5) What rotation does the quaternion $\mathbf{q} = 0.707 + 0.189\mathbf{i} + 0.378\mathbf{j} + 0.567\mathbf{k}$ represent?

(6) Consider a rotation through 30° about $(0, 1, 0)$. Find the new position under this rotation of the point $\mathbf{v} = (3, 4, 5)$ using quaternions.

(7) Find a single rotation that is equivalent to the following sequence of rotations: a rotation through 50° about the axis $(1, 2, 4)$, followed by a rotation through 37° about the axis $(2, -1, 2)$.

(8) Prove that a quaternion $\mathbf{q}$ represents a rotation if and only if $\mathbf{q}$ is a unit quaternion.

(9) Complete the proofs of properties (c) and (d) for the Slerp() function in section 14.9 using the hints provided.

Answers

Code for questions 4-7 can be found in exs.py

(1) $\mathbf{i}^2 = -1 \Rightarrow \mathbf{i}^3 = (-1)\mathbf{i} \Rightarrow \mathbf{i}\mathbf{i}^2 = (-1)\mathbf{i} \Rightarrow \mathbf{i}(-1) = (-1)\mathbf{i}$. Similarly for $\mathbf{j}$ and $\mathbf{k}$.

(2) $\mathbf{ijk} = -1 \Rightarrow \mathbf{i}^2\mathbf{jk} = -\mathbf{i} \Rightarrow -1\mathbf{jk} = -\mathbf{i} \Rightarrow \mathbf{jk} = \mathbf{i}$.

(3) $(a + b\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}) + (c + d\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}) = (a + b) + (c + d)\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}$;
$(a + b\mathbf{i} + 0\mathbf{j} + 0\mathbf{k})(c + d\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}) = (ac - bd) + (ad + bc)\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}$.

(4) The quaternion that represents the rotation is

$$\mathbf{q} = \cos(60°/2) + \sin(60°/2)(\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}) = \frac{\sqrt{3}}{2} + \frac{1}{2}\mathbf{i}.$$

(5) The magnitude of the rotation is $2\cos^{-1} 0.707 = 2 \times 45° = 90°$. The axis of the rotation is $(0.189, 0.378, 0.567)$.

(6) Using quaternions, $\mathbf{q} = 0.966 + 0.259\mathbf{j}$, $\mathbf{q}^{-1} = 0.966 - 0.259\mathbf{j}$, and $\mathbf{v} = 3.0\mathbf{i} + 4.0\mathbf{j} + 5.0\mathbf{k}$, so $\mathbf{qvq}^{-1} = 5.098\mathbf{i} + 4.000\mathbf{j} + 2.830\mathbf{k}$.

(7) The quaternion $\mathbf{q}_1 = 0.906 + 0.092\mathbf{i} + 0.184\mathbf{j} + 0.369\mathbf{k}$ represents a rotation through 50° about $(1, 2, 4)$, while $\mathbf{q}_2 = 0.948 + 0.212\mathbf{i} - 0.106\mathbf{j} + 0.212\mathbf{k}$ represents a rotation through 37° about $(2, -1, 2)$. Thus, the quaternion

$$\mathbf{q}_3 = \mathbf{q}_2\mathbf{q}_1 = 0.781 + 0.201\mathbf{i} + 0.021\mathbf{j} + 0.590\mathbf{k}$$

represents the first rotation followed by the second. The magnitude of this rotation is $2\cos^{-1} 0.781 = 77.22°$, and its unit axis is

$$1/\sin(77.22°/2)(0.201, 0.021, 0.590) = (0.322, 0.033, 0.946).$$

(8) Let $\mathbf{v} = (x, y, z)$ be a unit vector and $\mathbf{q} = \sin(\theta/2) + \cos(\theta/2)\mathbf{v}$ be the quaternion that represents a rotation through θ about $\mathbf{v}$. Then $\mathbf{q} = \sin(\theta/2) + x\cos(\theta/2)\mathbf{i} + y\cos(\theta/2)\mathbf{j} + z\cos(\theta/2)\mathbf{k}$, so that

$$\begin{aligned} |\mathbf{q}| &= \sqrt{\sin^2(\theta/2) + x^2\cos^2(\theta/2) + y^2\cos^2(\theta/2) + z^2\cos^2(\theta/2)} \\ &= \sqrt{\sin^2(\theta/2) + \cos^2(\theta/2)(x^2 + y^2 + z^2)}. \end{aligned}$$

But $\mathbf{v} = (x, y, z)$ is a unit vector, so $x^2 + y^2 + z^2 = 1$, and

$$|\mathbf{q}| = \sqrt{\sin^2(\theta/2) + \cos^2(\theta/2)} = 1.$$

Thus, $\mathbf{q}$ is a unit quaternion.

Conversely, let $\mathbf{q} = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$ be a unit quaternion. Then $w^2 + x^2 + y^2 + z^2 = 1$, so that the absolute value of each component of $\mathbf{q} \leq 1$. In particular, $-1 \leq w \leq 1$; there exists an angle θ with $\sin(\theta/2) = w$. Then $x^2 + y^2 + z^2 = 1 - \sin^2(\theta/2) = \cos^2(\theta/2)$, and $\mathbf{q}$ represents the rotation through θ about the axis $\mathbf{v} = (x/\cos(\theta/2), y/\cos(\theta/2), z/\cos(\theta/2))$.

Note that $\mathbf{v}$ is a unit vector:

$$(x/\cos(\theta/2))^2 + (y/\cos(\theta/2))^2 + (z/\cos(\theta/2))^2 = \cos^2(\theta/2)/\cos^2(\theta/2) = 1.$$

15

Curve Fitting

Why Study Curve Fitting, Splines and Bézier curves?

- **They produce smooth, flexible curves from simple data**.
- **They're numerically stable and easy to compute**. Bézier curves use the de Casteljau algorithm, which is robust and works with linear interpolation. Splines provide local control and avoid oscillations.
- **They reduce complexity and improve performance**. A spline can approximate a complicated curve or dataset using just a few parameters, avoiding the high degree and instability of a single polynomial.

15.1 Introduction

The general problem discussed in this chapter is how to fit a curve to a given set of (2D) data points $(x_0, y_0), (x_1, y_1), \cdots (x_n, y_n)$.

If the number of points is small (say, 7 or less) then interpolation is a good way to determine the coefficients a_i of a polynomial that fits the data:

$$P(x) = a_0 + a_1 x + \cdots + a_n x^n$$

such that

$$P(x_i) = y_i, \quad i = 0, 1, \cdots, n$$

There are many methods that do this – Lagrange, Aitken-Neville, and Hermite – but we'll utilize *Newton's divided differences interpolation*, which is both simple mathematically and has a straightforward, fast implementation.

However, all of these approaches are prone to becoming numerically unstable when applied to more data, with the resulting polynomial oscillating wildly to accommodate every point. One solution is to switch to approximation – fit a lower-order polynomial to the data which doesn't pass through all of the supplied points but still offers a satisfactory close fit. We've already used the most common technique of this type, *least-squares*, back in Section 6.8, so we'll only briefly discuss it here.

Another choice is to switch to piece-wise polynomials of small degree, such as *cubic splines*, which fit multiple curve segments through point intervals while ensuring that the curves run together smoothly to give the appearance of a single equation.

A radically different approach is treat most of the data as *control points* which affect the curve's curvature. This lends itself to an interactive definition of the curve by moving the control points. We'll examine *Bézier curves* here, which offer a pleasing geometric and algebraic interpretation for how control points influence the curve.

Polynomial interpolation, least squares approximation, and cubic splines are usually discussed in textbooks on numerical analysis, such as Burden [**BFB16**] and Kiusalaas [**Kiu13**]. Bézier curves, and extensions such as B-splines and NURBS, are more likely to be described in analytic geometry textbooks, such as Marsh [**Mar05**] and Mortenson [**Mor99**].

15.2 Newton Interpolation

Newton's method manipulates a polynomial of the form

$$P_n(x) = a_0 + (x-x_0)a_1 + (x-x_0)(x-x_1)a_2 + \cdots + (x-x_0)(x-x_1)\cdots(x-x_{n-1})a_n$$

For example, four data points will be modeled by

$$\begin{aligned} P_3(x) &= a_0 + (x-x_0)a_1 + (x-x_0)(x-x_1)a_2 + (x-x_0)(x-x_1)(x-x_2)a_3 \\ &= a_0 + (x-x_0)\{a_1 + (x-x_1)[a_2 + (x-x_2)a_3]\} \end{aligned}$$

There's a natural recursive way to evaluate $P_n(x)$ for arbitrary n:

$$P_0(x) = a_n \quad P_k(x) = a_{n-k} + (x - x_{n-k})P_{k-1}(x), \quad k = 1, 2, ..., n$$

but, in practice, it's more efficient to implement the nested multiplications using a loop as in evalPoly() in newtonInterp.py:

```
def evalPoly(a, xData, x):
  n = len(xData) - 1
  poly = a[n]
  for k in range(1, n + 1):
    poly = a[n - k] + (x - xData[n - k]) * poly
  return poly
```

P_n's coefficients are determined by making the polynomial pass through every data point: $y_i = P_n(x_i),\ i = 0, 1, \ldots, n$. This yields:

$$\begin{aligned} y_0 &= a_0 \\ y_1 &= a_0 + (x_1 - x_0)a_1 \\ y_2 &= a_0 + (x_2 - x_0)a_1 + (x_2 - x_0)(x_2 - x_1)a_2 \\ &\vdots \\ y_n &= a_0 + (x_n - x_0)a_1 + \cdots + (x_n - x_0)(x_n - x_1)\cdots(x_n - x_{n-1})a_n \end{aligned}$$

The values for a_i are obtained by calculating increasing levels of *divided difference* equations (∇^i):

$$\begin{aligned} \nabla y_i &= \frac{y_i - y_0}{x_i - x_0}, \quad i = 1, 2, \ldots, n \\ \nabla^2 y_i &= \frac{\nabla y_i - \nabla y_1}{x_i - x_1}, \quad i = 2, 3, \ldots, n \\ \nabla^3 y_i &= \frac{\nabla^2 y_i - \nabla^2 y_2}{x_i - x_2}, \quad i = 3, 4, \ldots, n \\ &\vdots \\ \nabla^n y_n &= \frac{\nabla^{n-1} y_n - \nabla^{n-1} y_{n-1}}{x_n - x_{n-1}} \end{aligned}$$

After the level 1 equations (∇y_i) have been evaluated, their values are used to obtain the level 2 results ($\nabla^2 y_i$), and this process continues until a single level n equation is reached. This procedure can be represented by Table 15.1.

x_0	y_0				
x_1	y_1	∇y_1			
x_2	y_2	∇y_2	$\nabla^2 y_2$		
x_3	y_3	∇y_3	$\nabla^2 y_3$	$\nabla^3 y_3$	
x_4	y_4	∇y_4	$\nabla^2 y_4$	$\nabla^3 y_4$	$\nabla^4 y_4$

Table 15.1. Divided Difference Table for $n = 4$.

Once the table is completed, the a_i coefficients can be extracted from the main diagonal:

$$a_0 = y_0, \quad a_1 = \nabla y_1, \quad a_2 = \nabla^2 y_2, \ldots, \quad a_n = \nabla^n y_n$$

However, there's no need to maintain the entire table if each column is evaluated in a bottom-up order, since the new data can be stored by overwriting the old column, thereby allowing the use of a single list to hold all the values. This is implemented by coefs() in newtonInterp.py:

```
def coefs(xData, yData):
  m = len(xData)
  a = yData[:]  # make a copy
```

```
  for k in range(1, m):
    for j in range(m-1, k-1, -1):
      a[j] = (a[j] - a[j-1]) / (xData[j] - xData[j-k])
  return a
```

Initially, a contains the y-coordinates corresponding to the second column of Table 15.1. Each pass through the outer loop generates the entries in the next column, which are stored by overwriting a.

Fig. 15.1 shows newtonInterp.py's plot of the interpolated curve for the data:

```
xData = [0.15, 2.3, 3.15, 4.85, 6.25, 7.95]
yData = [4.79867, 4.49013, 4.2243, 3.47313, 2.66674, 1.51909]
```

against the actual curve $y = 4.8\cos(\pi x/20)$.

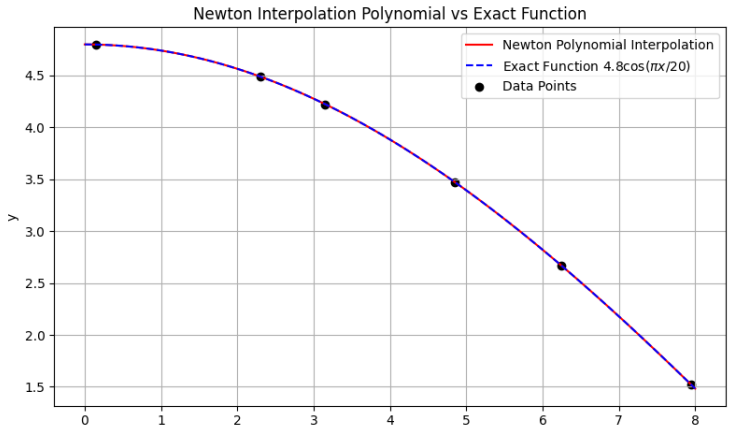


Figure 15.1. Newton Interpolation

The polynomial equation is:

```
P(x) = 4.79867 - 0.144(x-0.15) - 0.056(x-0.15)(x-2.30) +
          0.001(x-0.15)(x-2.30)(x-3.15) +
          0.000(x-0.15)(x-2.30)(x-3.15)(x-4.85) -
          0.000(x-0.15)(x-2.30)(x-3.15)(x-4.85)(x-6.25)
```

which illustrates a drawback of this approach – the equation isn't in a standard power form (e.g. $c_0 + c_1x + c_2x^2 + ...$). Also note that in this case a_4 and a_5 are 0 to 3 d.p., so the curve could be simplified.

15.2.1 The Runge Phenomenon. The Runge phenomenon arises when a curve is approximated by a high-order polynomial that passes through many points, especially points that are evenly spaced out. The polynomial will tend to increasingly oscillate as the number of coordinates grows, as illustrated in Fig.

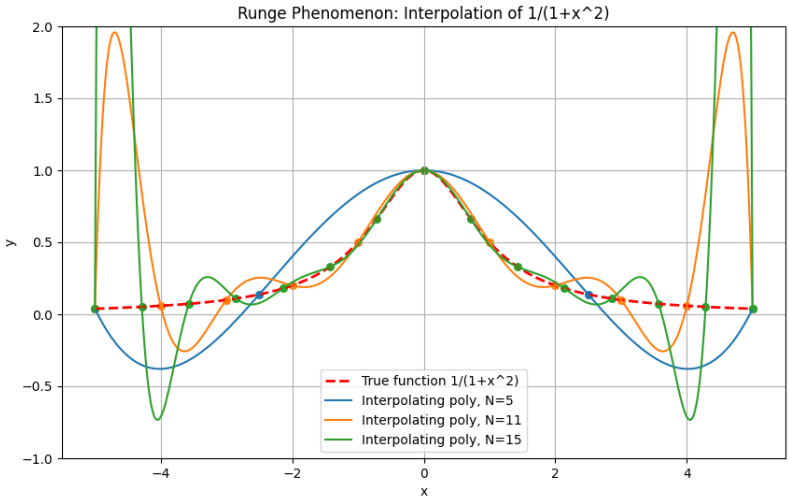


Figure 15.2. The Runge Phenomenon

15.2 which plots the equation $f(x) = 1/(1+x^2)$ against polynomial interpolations using 5, 11, and 15 equally spaced points.

The code that generates Fig. 15.2 is in `runge.py`, and uses the same Newton interpolation as `newtonInterp.py`.

One solution is to vary the distribution of the sampled coordinates so they're more densely concentrated at the edges of the interpolation interval. However, in the next two sections we'll employ least squares fitting and cubic splines instead. The former gives us the freedom to choose lower-degree polynomial approximations to the data, while the latter fits multiple small order polynomials over sub-intervals of the data.

15.3 Least Squares Curve Fitting

Least Squares curve fitting was the subject of section 6.8, so won't be discussed again in much depth. One implementation issue is that the code in that section applied Gaussian elimination to row lists (see `quadReg.py`) whereas we'll use the matrix module, `Mat.py`, from Appendix I here. For example, the coefficients for the quadratic polynomial $a_0 + a_1x + a_2x^2$) approximating n data points will be represented by $\mathbf{Aa} = \mathbf{b}$ where

$$\mathbf{A} = \begin{bmatrix} n & \sum x_i & \sum x_i^2 \\ \sum x_i & \sum x_i^2 & \sum x_i^3 \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 \end{bmatrix}$$

and

$$\mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \\ \sum x_i^2 y_i \end{bmatrix}$$

polyLS.py uses the supplied n data points to create matrices for **A** and **b**, and then multiplies the inversion of **A** to **b** to find **a** (which is called `X` in the code snippet below):

```
# Data
xs = [0.1, 0.4, 0.5, 0.7, 0.7, 0.9]
ys = [0.61, 0.92, 0.99, 1.52, 1.47, 2.03]
nPts = len(xs)
order = int(input(f"Order of polynomial (n < {nPts-1}): "))
m = order + 1

A, B = createMats(xs, ys, m)
# Solve system of linear equations A*X = B
try:
  AInv = A.inverse()
  X = AInv * B
  coefs = X.transpose().toList()[0]
except ValueError as e:
  print(f"Error: {e}")
  sys.exit(1)
xsFit, ysFit = fitCurve(xs, ys, coefs)
```

Note that the code asks the user to supply a polynomial order which determines the number of simultaneous equations to solve. The program plots the resulting polynomial against the original data and reports the goodness of fit.

Fig. 15.3 shows four separate runs of polyLS.py with polynomial orders from 1 (linear) to 4.

Naturally, the goodness of fit improves as the order increases, but this measure doesn't reflect our desire for curve simplicity. Probably, the best approximation is of order two.

15.4 Cubic Spline Interpolation

Spline interpolation divides the data range into intervals and constructs a different polynomial for each piece. The simplest type is piece-wise linear interpolation, which joins the data points with straight lines, with the obvious drawback of a lack of "smoothness" between the pieces. We really want the two curve segments that meet at a common point to have the same slope (i.e. the same tangent) and curvature (the same rate of change for their tangents). These smoothness requirements can be defined geometrically for a given function:

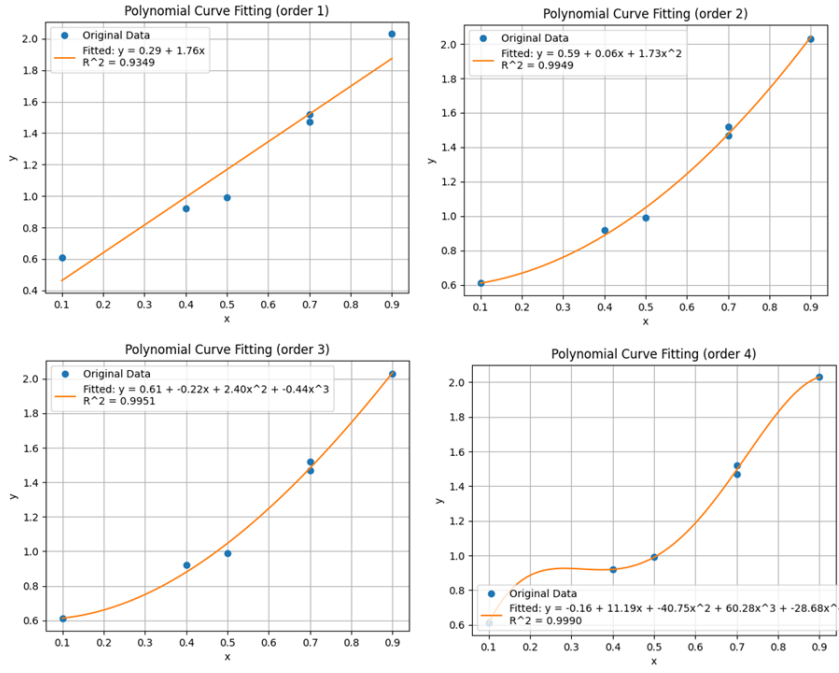


Figure 15.3. Polynomial Curve Fittings with Different Orders

- C^0: the function is continuous;
- C^1: the function is continuously differentiable (i.e. it's first derivative is continuous);
- C^2: the function has a continuous second derivative.

The simplest piece-wise approximation that satisfies these properties is the cubic spline, and Fig. 15.4 shows how these requirements are expressed. The spline S is made up of a series of cubic polynomials, denoted $S_i(x)$, on the intervals $[x_i, x_{i+1}]$ for $i = 0, 1, \ldots, n-1$.

We'll assume that the end points at x_0 and x_n have no curvature (i.e. $S''(x_0) = S''(x_n) = 0$. This creates a *natural* spline, so named because it mimics the natural shape a flexible strip assumes if forced to pass through specified points (as in Fig. 15.5)

The polynomial for each interval $[x_i, x_{i+1}]$ is written as:

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3. \tag{15.1}$$

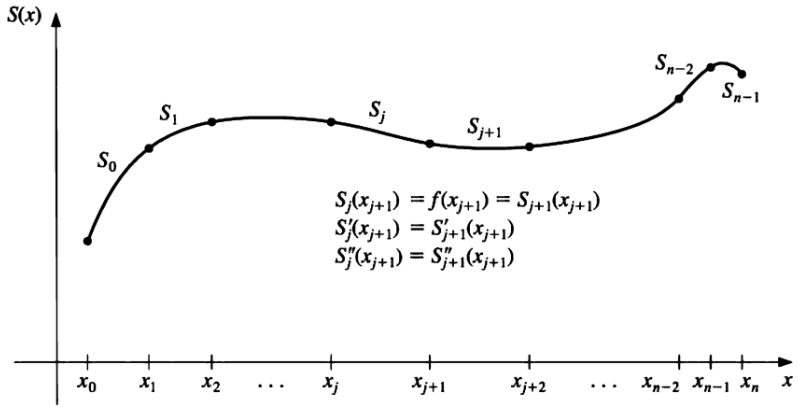


Figure 15.4. Smoothness Requirements for a Cubic Spline

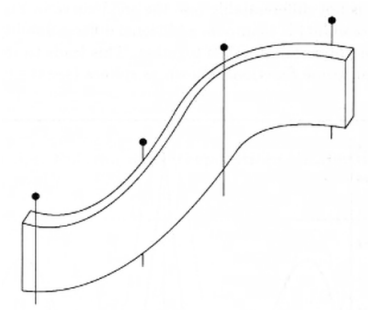


Figure 15.5. A Real-world Spline

Note that $S_i(x_i) = a_i = y_i$, so the unknown coefficients are b_i, c_i, and d_i. Our goal is to obtain expressions for b_i and d_i in terms of c_i, and to formulate c_i as simultaneous linear equations that can be solved.

We start by formalizing the smoothness requirements for each $S_i(x)$. Each interior point on the spline, x_i $(1 \leq i \leq n-1)$, exhibits:

- First derivative continuity (i.e. slope equality):

$$S_i'(x_{i+1}) = S_{i+1}'(x_{i+1}). \tag{15.2}$$

- Second derivative continuity (i.e. slope change (curvature) equality):

$$S_i''(x_{i+1}) = S_{i+1}''(x_{i+1}). \tag{15.3}$$

Also, the natural spline boundary conditions mean that:

$$S''(x_0) = 0, \quad S''(x_n) = 0. \tag{15.4}$$

The derivatives of $S_i()$ are:

$$S_i'(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2, \tag{15.5}$$
$$S_i''(x) = 2c_i + 6d_i(x - x_i). \tag{15.6}$$

Let $h_i = x_{i+1} - x_i$, and then at x_{i+1}:

$$S_i(x_{i+1}) = a_i + b_i h_i + c_i h_i^2 + d_i h_i^3, \tag{15.7}$$
$$S_i'(x_{i+1}) = b_i + 2c_i h_i + 3d_i h_i^2, \tag{15.8}$$
$$S_i''(x_{i+1}) = 2c_i + 6d_i h_i. \tag{15.9}$$

Apply Equation (15.5) at x_{i+1}:

$$S_{i+1}'(x_{i+1}) = b_{i+1}$$

Substitute this equation and Equation (15.2) into Equation (15.8):

$$b_{i+1} = b_i + 2c_i h_i + 3d_i h_i^2 \tag{15.10}$$

Apply Equation (15.6) at x_{i+1}:

$$S_{i+1}''(x_{i+1}) = 2c_{i+1}$$

Substitute this equation and Equation (15.3) into Equation (15.9):

$$2c_{i+1} = 2c_i + 6d_i h_i$$
$$c_{i+1} = c_i + 3d_i h_i$$

Rearranging:

$$d_i = (c_{i+1} - c_i)/3h_i \tag{15.11}$$

We'll use this equation later to obtain d_i values once we've calculated c_i's.

Substitute $S_i(x_{i+1}) = a_{i+1}$ into Equation (15.7):

$$a_{i+1} = a_i + b_i h_i + c_i h_i^2 + d_i h_i^3 \tag{15.12}$$

Substitute Equation (15.11) into Equation (15.12):

$$a_{i+1} = a_i + b_i h_i + \frac{h_i^2}{3}(2c_i + c_{i+1})$$

Rearrange for b_i:

$$b_i = \frac{1}{h_i}(a_{i+1} - a_i) - \frac{h_i}{3}(2c_i + c_{i+1}) \tag{15.13}$$

This equation will be used later to obtain values for b_i once we have the c_i's.

Substitute Equation (15.11) into Equation (15.10):

$$\begin{aligned} b_{i+1} &= b_i + 2c_i h_i + 3(\frac{c_{i+1} - c_i}{3h_i})h_i^2 \\ &= b_i + h_i(c_i + c_{i+1}) \end{aligned}$$

So

$$h_i(c_i + c_{i+1}) = b_{i+1} - b_i$$

Reducing the subscripts:

$$h_{i-1}(c_{i-1} + c_i) = b_i - b_{i-1}$$

Replace the right hand side of this equation by the right hand sides of Equation (15.13) for b_i and b_{i-1}:

$$h_{i-1}(c_{i-1}+c_i) = \frac{1}{h_i}(a_{i+1}-a_i) - \frac{1}{h_{i-1}}(a_i - a_{i-1}) - \frac{h_i}{3}(2c_i + 2c_{i+1}) - \frac{h_{i-1}}{3}(2c_{i-1} + c_i)$$

Simplify:

$$h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_i c_{i+1} = \frac{3}{h_i}(a_{i+1} - a_i) - \frac{3}{h_{i-1}}(a_i - a_{i-1}) \quad (15.14)$$

Note that this equation only applies in the range $i = 1, 2, \ldots n-1$ due to its use of c_{i-1} and c_{i+1}, so we must determine c_0 and c_n separately. This is possible by using the boundary conditions in Equation (15.4) and Equation (15.6):

$$\begin{aligned} S''(x_0) &= 0 \\ S_i''(x) &= 2c_i + 6d_i(x - x_i). \end{aligned}$$

So

$$S_0''(x_0) = 0 = 2c_0 + 6d_0(x_0 - x_0)$$

and so $c_0 = 0$. Similarly:

$$S''(x_n) = 0.$$

So

$$S_n''(x_n) = 0 = 2c_n + 6d_n(x_n - x_n)$$

and so $c_n = 0$.

$c_0 = 0$, $c_n = 0$ and Equation (15.14) specify a linear system described by $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a $(n+1) \times (n+1)$ matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & \\ h_0 & 2(h_0 + h_1) & h_1 & & 0 & \\ 0 & h_1 & 2(h_1 + h_2) & h_2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & h_{n-2} & 2(h_{n-2} + h_{n-1}) & h_{n-1} \\ 0 & \cdots & 0 & 0 & 1 & \end{bmatrix},$$

and $\mathbf{x}$ and $\mathbf{b}$ are the vectors:

$$\mathbf{x} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} 0 \\ \frac{3}{h_1}(a_2 - a_1) - \frac{3}{h_0}(a_1 - a_0) \\ \vdots \\ \frac{3}{h_{n-1}}(a_n - a_{n-1}) - \frac{3}{h_{n-2}}(a_{n-1} - a_{n-2}) \\ 0 \end{bmatrix}.$$

The matrix $\mathbf{A}$ is diagonally dominant; that is, in each row, the magnitude of the diagonal entry exceeds the sum of the magnitudes of all the other entries in the row. Such a *tridiagonal* matrix can be solved using the Thomas algorithm (`https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm`), which produces a result in $O(n)$ time rather than the $O(n^3)$ required by Gaussian elimination. However, we'll stick with the Gaussian so we can use our `Mat.py` class (see Appendix I), and also because the number of data points (n) is small. An optimization we will take is to note that there's no need to encode all of the matrices since we already have values for c_0 and c_n (both equal to 0). Its only necessary to consider the range $i = 1, 2, \ldots, n-1$, when building $\mathbf{A}$, $\mathbf{x}$, and $\mathbf{b}$.

cubicSplineMat.py fits a spline to the data points represented by the lists:

```
xs = [0, 1, 2, 3, 4, 5]
ys = [0, 0.5, 2, 1.5, 0.5, 0]
```

It produces the graph shown in Fig. 15.6.

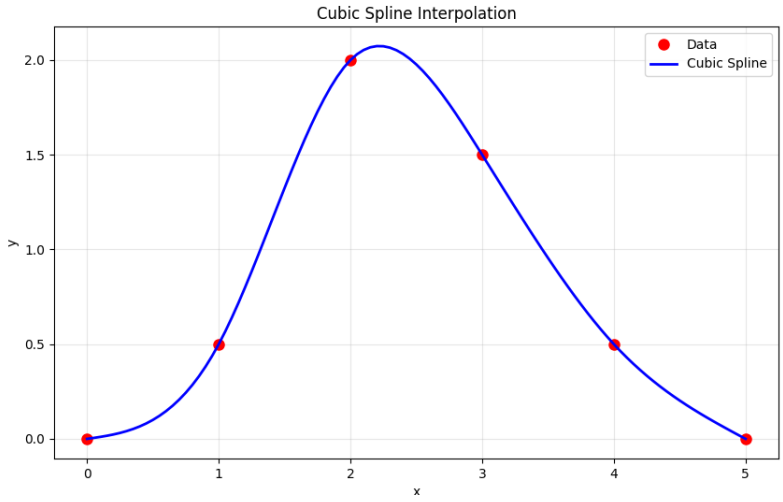


Figure 15.6. A Cubic Spline and Data

The coefficients (stored in the lists `ais`, `bs`, `cs`, and `ds`) are calculated by cubicSpline():

```
def cubicSpline(xs, ys):
  n = len(xs)
  if n < 3:
    raise ValueError("Need at least 3 points for cubic spline")

  ais = ys.copy()
  bs = [0]*n
  cs = [0]*n
  ds = [0]*n

  # Compute spacing
  hs = [xs[i+1]-xs[i] for i in range(n-1)]

  # Create matrices and solve the system
  A, b = buildMats(ais, hs, n)
  try:
    cSoln = A.inverse() * b
    cs[0] = 0
    for i in range(cSoln.nRows):
      cs[i+1] = cSoln[i][0]
    cs[n-1] = 0
  except ValueError as e:
    raise ValueError(f"Failed to solve spline system: {e}")

  # Compute other coefs using the cs values
  for j in range(n-1):
    bs[j] = (ais[j+1] - ais[j]) / hs[j] - \
            hs[j] * (cs[j+1] + 2 * cs[j]) / 3
    ds[j] = (cs[j+1] - cs[j]) / (3 * hs[j])

  return ais, bs, cs, ds
```

`ais` and `hs` are passed to `buildMat()` to build the reduced size **A** and **b** matrices. **A** is inverted using the `Mat.inverse()` implementation of Gaussian elimination, and the `cs` coefficients are read from `A.inverse() * b`. Note that these values are only for the range $i = 1, 2, \ldots, n-1$. The `bs` and `ds` values are obtained via implementations of Equation (15.13) and (15.11).

`buildMats()` creates a tridiagonal matrix for **A** and the **b** right-hand side, but restricted to the range $i = 1, 2, \ldots, n-1$. The function also prints the matrices as a useful debugging check. For the supplied input, the output is:

```
A =                                                    b=
 4.00    1.00    0.00    0.00                           3.00
 1.00    4.00    1.00    0.00                          -6.00
```

```
 0.00     1.00     4.00     1.00                    -1.50
 0.00     0.00     1.00     4.00                     1.50
```

Note that since six data points were supplied, then **A** is 4x4 and **b** is a 4-element vector.

The program finishes by printing out the coefficients used in $S_0, S_1, \ldots S_4$:

```
Interval [0, 1]: a=0.0000, b=0.1005, c=0.0000, d=0.3995
Interval [1, 2]: a=0.5000, b=1.2990, c=1.1986, d=-0.9976
Interval [2, 3]: a=2.0000, b=0.7033, c=-1.7943, d=0.5909
Interval [3, 4]: a=1.5000, b=-1.1124, c=-0.0215, d=0.1340
Interval [4, 5]: a=0.5000, b=-0.7536, c=0.3804, d=-0.1268
```

This makes it quite clear that the spline is a piece-wise curve made up of five cubic polynomials.

15.5 The Bézier Curve

Cubic splines and Bézier curves are both used for modeling smooth curves, but while a spline passes through all the data points, a Bézier curve treats all but the first and last coordinates as *control* points which influence the curve's curvature. The result is a single polynomial rather than the piece-wise polynomials that make up a spline. A cubic Bézier (degree 3) requires two control points, a quadratic Bézier (degree 2) requires just one, but a curve may utilize any number of control points if necessary.

A Bézier curve can be contructed geometrically with the de Casteljau algorithm, or algebraically as a combination of Bernstein polynomials. We'll consider each in turn.

15.5.1 A Geometric Construction. A second-degree Bézier curve can be built with the de Casteljau's algorithm in the manner illustrated by Fig. 15.7.

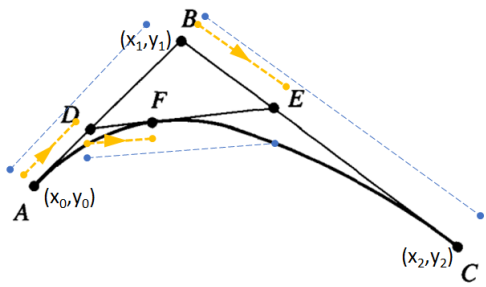


Figure 15.7. Geometric construction of a second-degree Bézier curve

The points A, B, C are selected so that the line AB is tangent to the curve at A, and BC is tangent to C. For some ratio u, where $0 \leq u \leq 1$, the points D and E are constructed so that

$$\frac{AD}{AB} = \frac{BE}{BC} = u$$

Crucially, F also utilizes the same ratio on DE so that $DF/DE = u$.

The curve is defined by how F moves from D to E while D and E are themselves moving (D from A to B, and E from B to C). This effect is best seen by running `animInterp.py` which animates the F point.

We can express this construction in terms of vectors with the following substitutions: Let the vector $\mathbf{a}$ represent point A, use $\mathbf{b}$ for point B, and $\mathbf{c}$ for C. The vectors $\mathbf{d}$ and $\mathbf{e}$ for D and E are:

$$\mathbf{d} = \mathbf{a} + u(\mathbf{b} - \mathbf{a})$$
$$\mathbf{e} = \mathbf{d} + u(\mathbf{c} - \mathbf{b})$$

which let's us write $\mathbf{f}$ for F as:

$$\mathbf{f} = \mathbf{d} + u(\mathbf{e} - \mathbf{d})$$

The equations for $\mathbf{d}$ and $\mathbf{e}$ can be substituted into this equation to give:

$$\mathbf{f} = (1 - u^2)\mathbf{a} + 2u(1 - u)\mathbf{b} + u^2\mathbf{c}$$

We generalize this equation for any point on the curve by parameterizing the point in terms of u:

$$\mathbf{p}(u) = (1 - u^2)\mathbf{a} + 2u(1 - u)\mathbf{b} + u^2\mathbf{c}$$

This second-degree equation utilizes $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ as control points, and the ratio u as the parametric variable. Similar constructions are employed for Bézier curves of any degree.

Fig. 15.8 shows the construction of a cubic Bézier curve, which requires four control points A, B, C, and D. The curve begins at A as a tangent to line AB, and ends at D, as a tangent to CD.

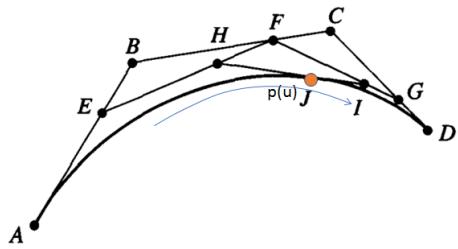


Figure 15.8. Geometric construction of a Cubic Bézier curve

We construct E, F, and G so that

$$\frac{AE}{AB} = \frac{BF}{BC} = \frac{CG}{CD} = u$$

On EF and FG, H and I are located at

$$\frac{EH}{EF} = \frac{FI}{FG} = u$$

Finally, on HI, J is positioned so that

$$\frac{HJ}{HI} = u$$

No further subdivisions are possible, which means that J moves along the curve.

If A, B, C, and D are represented by the vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$, then the intermediate points E, F, G, H, and I can be expressed as:

$$\mathbf{e} = \mathbf{a} + u(\mathbf{b} - \mathbf{a}) \quad \mathbf{f} = \mathbf{b} + u(\mathbf{c} - \mathbf{b}) \quad \mathbf{g} = \mathbf{d} + u(\mathbf{d} - \mathbf{c})$$
$$\mathbf{h} = \mathbf{e} + u(\mathbf{f} - \mathbf{e}) \quad \mathbf{i} = \mathbf{f} + u(\mathbf{g} - \mathbf{f})$$

J is then:

$$\mathbf{j} = \mathbf{h} + u(\mathbf{i} - \mathbf{h})$$

This expression can be expanded:

$$\begin{aligned}
\mathbf{p}(u) &= \mathbf{h} + u(\mathbf{i} - \mathbf{h}) \\
&= \mathbf{e} + u(\mathbf{f} - \mathbf{e}) + u(\mathbf{f} + u(\mathbf{g} - \mathbf{f}) - (\mathbf{e} + u(\mathbf{f} - \mathbf{e}))) \\
&= \mathbf{a} + u(\mathbf{b} - \mathbf{a}) + u(\mathbf{b} + u(\mathbf{c} - \mathbf{b}) - (\mathbf{a} + u(\mathbf{b} - \mathbf{a}))) + \\
&\quad u(\mathbf{b} + u(\mathbf{c} - \mathbf{b}) + u(\mathbf{d} + u(\mathbf{d} - \mathbf{c}) - (\mathbf{b} + u(\mathbf{c} - \mathbf{b})))) - \\
&\quad (\mathbf{a} + u(\mathbf{b} - \mathbf{a}) + u(\mathbf{b} + u(\mathbf{c} - \mathbf{b}) - (\mathbf{a} + u(\mathbf{b} - \mathbf{a}))))
\end{aligned}$$

This (thankfully) simplifies to:

$$\mathbf{p}(u) = (1-u)^3\mathbf{a} + 3u(1-u)^2\mathbf{b} + 3u^2(1-u)\mathbf{c} + u^3\mathbf{d},$$

a cubic curve with four control points.

15.5.2 Implementing De Casteljau's Algorithm.

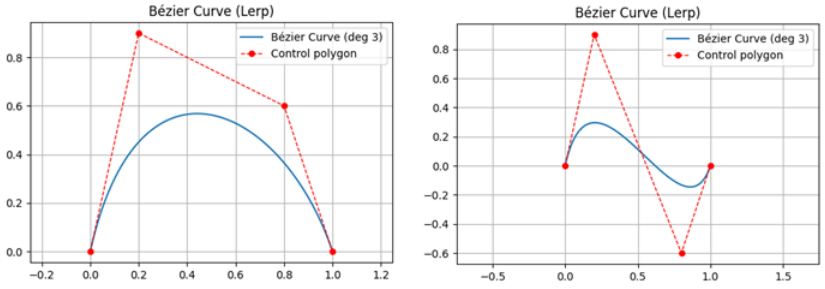
bezLerp.py plots a Bézier curve by passing a list of control points to `bezierCurveDC()` which implements de Casteljau's algorithm:

```
ctrlPts = [(0,0),(0.2,0.9),(0.8,0.6),(1,0)]
curveX, curveY = bezUtils.bezierCurveDC(ctrlPts, 200)
```

The graph is shown in Fig. 15.9(a). If the third control point is changed to `(0.8,-0.6)` then the plot on the right is produced.

`bezierCurveDC()` in bezUtils.py:

```
def bezierCurveDC(ctrlPts, nSamps):
  xs = []   # points making up the curve
  ys = []
  if nSamps < 2:  # no. of samples
```



(a) Third Control Point: (0.8, 0.6) (b) Third Control Point: (0.8, -0.6)

Figure 15.9. Using De Casteljau to Draw Bézier Curves

```
    nSamps = 2
  for s in range(nSamps+1):
    u = s / nSamps
    qx, qy = bezLerp(ctrlPts, u)
    xs.append(qx)  # one point for each sample
    ys.append(qy)
  return xs, ys
```

The real work is done by `bezLerp()` which returns a curve coordinate for a given u parameter.

```
def bezLerp(ctrlPts, u):
  pts = list(ctrlPts)  # make a working copy
  n = len(pts)
  for r in range(1, n):   # r = interpolation depth
    for i in range(n - r):
      x = lerp(pts[i][0], pts[i+1][0], u)
      y = lerp(pts[i][1], pts[i+1][1], u)
      pts[i] = (x, y)
  return pts[0]

def lerp(a, b, u):  # Linear interpolation
  return a + (b-a)*u
```

To understand `bezLerp()` it helps to manually execute it on a small example, such as the curve in Fig. 15.7. In that case, three control points (A, B, C) are passed to the function along with u.

The `r` loop will execute twice: once with $r = 1$ and then with $r = 2$. For $r = 1$, the i loop will range over 0 and 1 which will interpolate the line AB and then BC, which 'moves' D and E (see Fig. 15.7). Note that these points are written over the existing data in `pts` before the next iteration of r begins.

When $r = 2$, the inner loop will only consider 0 which means that it interpolates the line DE (since D and E are in `pts`). This is equivalent to calculating the position of point F (see Fig. 15.7), the current coordinate of the curve. F is stored in `pts`, and returned when `bezLerp()` finishes.

15.5.3 An Algebraic Definition. Any point $\mathbf{p}(u)$ on a Bézier curve can be written as:

$$\mathbf{p}(u) = \mathbf{p_0} f_0(u) + \mathbf{p_1} f_1(u) + ... + \mathbf{p_n} f_n(u)$$

where $0 \leq u \leq 1$, the vectors $\mathbf{p_i}$ are the control points, and the $f_i(u)$ functions must produce a curve that has certain characteristics:

(1) The curve must start at the first control point, $\mathbf{p_0}$, and end at the last, $\mathbf{p_n}$.

(2) The curve must start as a tangent to the line $\mathbf{p_1} - \mathbf{p_0}$ at $\mathbf{p_0}$ and end as a tangent to $\mathbf{p_n} - \mathbf{p_{n-1}}$ at $\mathbf{p_n}$.

(3) The functions $f_i(u)$ must be symmetric with respect to u and $(1 - u)$. This lets us reverse the sequence of control points without changing the shape of the curve.

A family of functions called *Bernstein* polynomials satisfies these requirements, and $\mathbf{p}(u)$ can be rewritten using them:

$$\mathbf{p}(u) = \sum_{i=0}^{n} \mathbf{p_i} B_{i,n}(u) \tag{15.15}$$

where

$$B_{i,n}(u) = \binom{n}{i} u^i (1-u)^{n-i} \tag{15.16}$$

and

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

A second-degree Bézier curve (where $n = 2$ and there are three control points) is obtained from Equation (15.15) as:

$$\mathbf{p}(u) = \mathbf{p_0} B_{0,2}(u) + \mathbf{p_1} B_{1,2}(u) + \mathbf{p_2} B_{2,2}(u)$$

From Equation (15.16) we find

$$B_{0,2}(u) = (1-u)^2$$
$$B_{1,2}(u) = 2u(1-u)$$
$$B_{2,2}(u) = u^2$$

Substituting them into $\mathbf{p}(u)$:

$$\mathbf{p}(u) = (1-u^2)\mathbf{p_0} + 2u(1-u)\mathbf{p_1} + u^2\mathbf{p_2}$$

This is the same as the expression we obtained from the geometric construction, albeit with **a**, **b**, and **c** replaced by the corresponding $\mathbf{p_i}$'s.

Now, let's expand Equation (15.15) for a cubic curve, where $n = 3$:

$$\mathbf{p}(u) = \mathbf{p_0}B_{0,3}(u) + \mathbf{p_1}B_{1,3}(u) + \mathbf{p_2}B_{2,3}(u) + \mathbf{p_3}B_{3,3}(u)$$

From Equation (15.16) we find

$$\begin{aligned} B_{0,3}(u) &= (1-u)^3 \\ B_{1,3}(u) &= 3u(1-u)^2 \\ B_{2,3}(u) &= 3u^2(1-u) \\ B_{3,3}(u) &= u^3 \end{aligned}$$

Substituting these into $\mathbf{p}(u)$:

$$\mathbf{p}(u) = (1-u)^3\mathbf{p_0} + 3u(1-u)^2\mathbf{p_1} + 3u^2(1-u)\mathbf{p_2} + u^3\mathbf{p_3}$$

bernsteins.py draws the Bernstein curves for a given Bézier curve degree order. For instance, Fig. 15.10 shows the four curves $(B_{0,3}(), B_{1,3}(), B_{2,3}(), B_{3,3}())$.

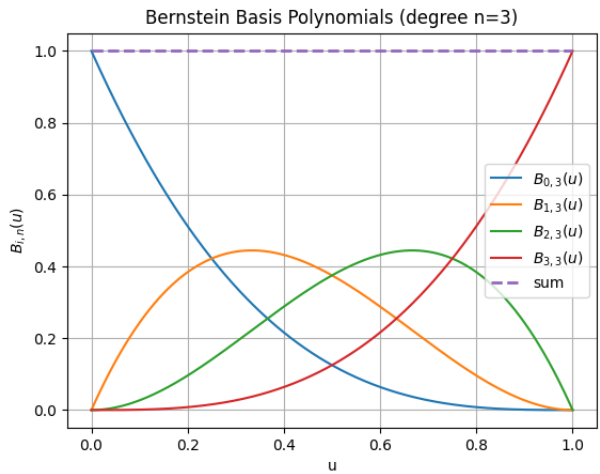


Figure 15.10. Bernstein Polynomials for a Cubic Bézier

The contribution of the first control point, $\mathbf{p_0}$, is propagated throughout the curve by $B_{0,3}()$ and is most influential at $u = 0$. Note that the other control points don't contribute to $\mathbf{p}(u)$ at $u = 0$ at all because $B_{1,3}(0) = B_{2,3}(0) = B_{3,3}(0) = 0$.

Control point $\mathbf{p_1}$ is most influential at $u = 1/3$, and $\mathbf{p_2}$ at $u = 2/3$. At $u = 1$, only $\mathbf{p_3}$ affects $\mathbf{p}(u)$. Also, notice the symmetry of $B_{0,1}(u)$ and $B_{3,3}(u)$, as well as that of $B_{1,3}(u)$ and $B_{2,3}(u)$.

Another useful way to get a feeling for how control points influence a curve is by moving the points interactively in dragPath.py. Three screenshots of the program running are shown in Fig. 15.11.

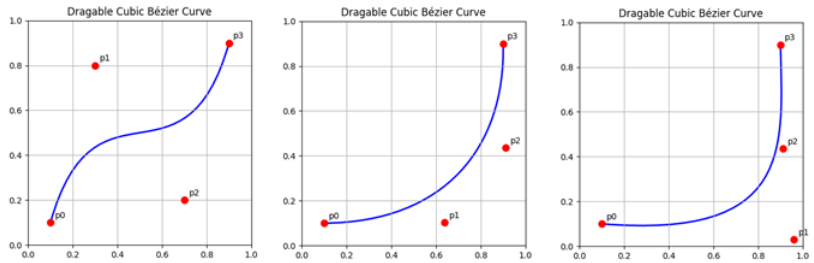


Figure 15.11. Moving Control Point P1

The images show the effect on a cubic curve of moving control point $P1$ from the top-left to the bottom-right of the plot. Indeed, any of the four points can be moved.

15.5.4 Implementing Bézier Using Bernstein Functions.

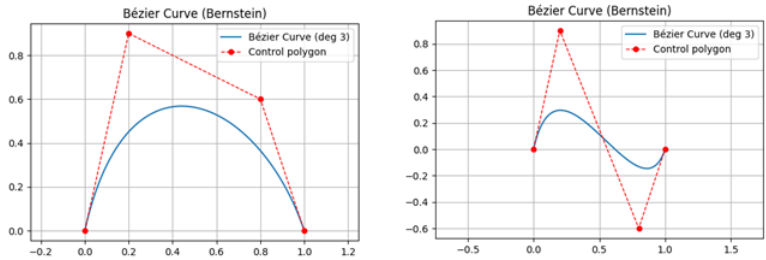
bezBern.py plot a Bézier curve by calling `bezierCurve()` with a list of control points. The relevant code snippet in `bezBern.py` is:

```
ctrlPts = [(0,0),(0.2,0.9),(0.8,0.6),(1,0)]
curveX, curveY = bezUtils.bezierCurve(ctrlPts, 200)
```

The graph is shown in Fig. 15.12(a). If the third point is changed to `(0.8,-0.6)` then the plot on the right (b) is produced. As we'd expect, Fig. 15.12 is identical to Fig. 15.9.

Much of `bezierCurve()` in bezUtils.py is similar to `bezierCurveDC()`:

```
def bezierCurve(ctrlPts, nSamps=100):
  n = len(ctrlPts) - 1
  xs = []
  ys = []
  if nSamps < 2:
    nSamps = 2
  for step in range(nSamps + 1):
    u = step / nSamps
    qx, qy = 0, 0
    for i in range(n+1):
      b = bernstein(n, i, u)
      qx += b * ctrlPts[i][0]  # x
      qy += b * ctrlPts[i][1]  # y
```



(a) Third Control Point: (0.8, 0.6) (b) Third Control Point: (0.8, -0.6)

Figure 15.12. Using Bernstein Functions to Draw Bézier Curves

```
    xs.append(qx)
    ys.append(qy)
  return xs, ys

def bernstein(n, i, u):
  # Bernstein polynomial B_{i,n}(u)
  return math.comb(n, i) * (u**i) * ((1 - u)**(n - i))
```

The important differences lie inside the i loop, which calls `bernstein(n, i, u)` to obtain a particular $B_{i,n}(u)$ value. `bernstein()` is a direct translation of Equation (15.16).

15.5.5 Bézier 3D curves. Although we've been concentrating on 2D curves, it's not much work to extend the ideas to 3D. `bezier3D()` in bezUtils.py is virtually identical to `bezierCurve()` given above, but accepts control points of the form (x, y, z). The most important change this causes is the need to use `bernstein()`'s result *three* times rather than twice inside the i loop:

```
    # inside bezier3D()
    for i in range(n+1):
      b = bernstein(n, i, u)
      qx += b * ctrlPts[i][0]   # x
      qy += b * ctrlPts[i][1]   # y
      qz += b * ctrlPts[i][2]   # z
```

bez3D.py illustrates how to use `bezier3D()`:

```
ctrlPts = [ [0,0,0], [1,5,2],
            [5,1,8], [6,6,0] ]
curveX, curveY, curveZ = bezUtils.bezier3D(ctrlPts)
```

It generates the plot shown in Fig. 15.13.

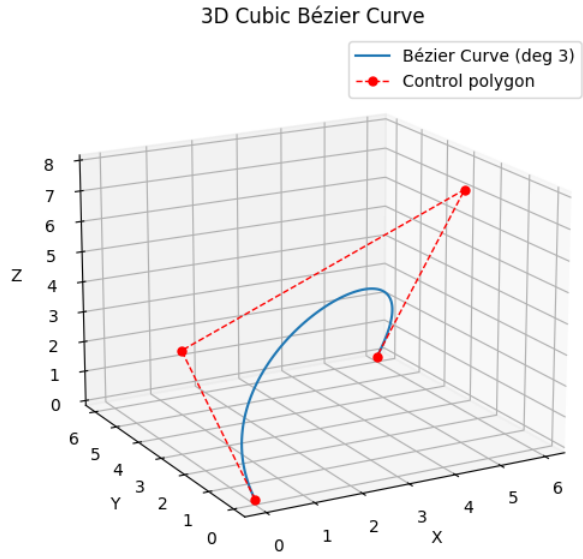


Figure 15.13. Using Bernstein Functions to Draw a 3D Bézier Curve

15.5.6 Bézier curves in Matplotlib. The Matplotlib `Path` class supports a set of `moveto`, `lineto`, and `curveto` commands for drawing line segments and curves. Most relevant for this section are the commands `CURVE3` and `CURVE4` for drawing a quadratic Bézier curve using three control points and a cubic curve with four points (`https://matplotlib.org/stable/users/explain/artists/paths.html`). Sadly, it's not currently possibly to add move control points to the curve, although it is possible to stitch together multiple quadratic and/or cubic curves using other `Path` commands.

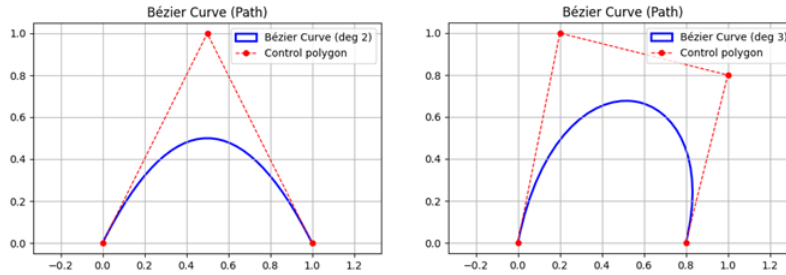
bezPath.py builds a Matplotlib patch using `Path` by calling `bezPath()`:

```
# control points (choose one of these)
# ctrlPts = [(0,0),(0.5,1),(1,0)]
ctrlPts = [(0,0),(0.2,1),(1,0.8),(0.8,0)]

bezierPath = bezUtils.bezPath(ctrlPts)
patch = PathPatch(bezierPath,
          facecolor="none", edgecolor="blue", lw=2,
          label=f"Bézier Curve (deg {len(ctrlPts)-1})")
```

Fig. 15.14 shows the two possible plots depending on which version of `ctrlPts` is uncommented.

`bezPath()` in bezUtils.py decides which `Path` commands to string together depending on the number of control points:



(a) Three Control Points (CURVE3) (b) Four Control Points (CURVE4)

Figure 15.14. Bézier Curves Using Matplotlib Paths

```
def bezPath(ctrlPts):
  n = len(ctrlPts)
  codes = []
  if n == 3:   # CURVE3 is for a quadratic B\'{e}zier
    codes = [Path.MOVETO, Path.CURVE3, Path.CURVE3]
  elif n == 4: # CURVE4 is for a cubic B\'{e}zier
    codes = [Path.MOVETO, Path.CURVE4,
             Path.CURVE4, Path.CURVE4]
  else:
    raise ValueError("Path lengths must == 3 or 4")
  return Path(ctrlPts, codes)
```

15.5.7 Rational Bézier Curves. We represent a rational Bézier curve in vector form as

$$\mathbf{p}(u) = \frac{\sum h_i \mathbf{p_i} B_{i,n}(u)}{\sum h_i B_{i,n}(u)}$$

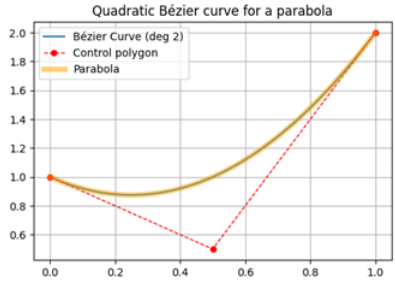
where the $\mathbf{p_i}$s are the control points and the h_is are weights. Note that this equation reduces to the previous definition for Bézier curves when all $h_i = 1$ (we'll call these *integral* Bézier curves from here on). Weights allow a much wider range of curves to be modeled, with one important class being conics; in particular, the circle, ellipse, and hyperbola. The parabola doesn't need the use of weights.

The two programs, parabola.py and circle.py, use integral (i.e. non-weighted) Bézier curves to mimic a parabola and a circle. The results are shown in Fig. 15.15.

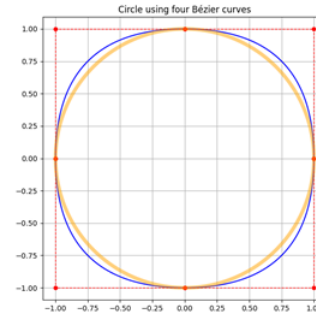
In parabola.py, the parabolic function is defined as:

```
a, b, c = 2, -1, 1

def parabola(x):
  return a*x**2 + b*x + c
```



(a) parabola.py



(b) circle.py

Figure 15.15. Integral Bézier Curves for a Parabola and a Circle

The a, b, and c coefficients are also used to specify the curve's control points:

```
ctrlPts = [(0, c), (0.5, c/2), (1, a+b+c)]
```

The plot in Fig. 15.15(a) strongly suggests that the integral Bézier curve is an exact fit for the parabola, and it's not difficult to prove that the curve is a perfect match. A quadratic Bézier curve is:

$$\mathbf{p}(u) = (1-u)^2\mathbf{p_0} + 2(1-u)u\mathbf{p_1} + u^2\mathbf{p_2}$$

Consider a simple parabola, such as $y = x^2$, and three points on it:

$$\mathbf{p_0} = (0,0) \quad \mathbf{p_1} = (0.5, 0.25) \quad \mathbf{p_2} = (1,1)$$

Plug those into the x-axis and y-axis components of $\mathbf{p}(u)$:

$$\begin{aligned} x(u) &= (1-u)^2 \cdot 0 + 2(1-u)u \cdot 0.5 + u^2 \cdot 1 = u \\ y(u) &= (1-u)^2 \cdot 0 + 2(1-u)u \cdot 0.25 + u^2 \cdot 1 = u^2. \end{aligned}$$

This shows that the polynomial Bézier is equivalent to the parabolic equation.

Things are not so rosy for the curves mimicking a circle. Fig. 15.15(b) employs four curves to model the quadrants, and it's clear that the fit is rather poor, especially around the middle of each quadrant. The curves are implemented using four control point lists in `circle.py`:

```
cpArcs = [ # control points for each quarter arc
  [(1,0), (1,1), (0,1)],
  [(0,1), (-1,1), (-1,0)],
  [(-1,0), (-1,-1), (0,-1)],
  [(0,-1), (1,-1), (1,0)],
]
```

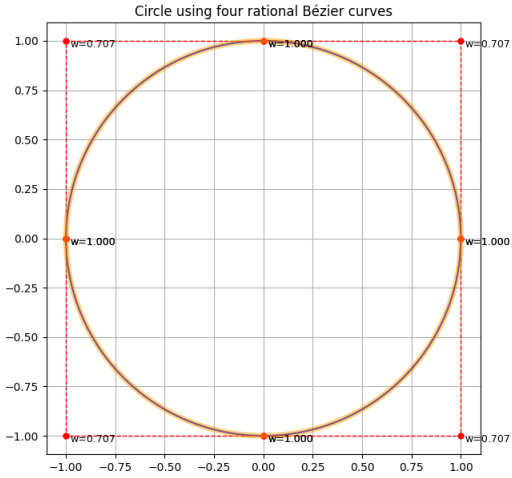


Figure 15.16. Rational Bézier Curve for a Circle

ratBez.py contains several rational Bézier curve examples, with the third one being the rational curve for a circle, shown in Fig. 15.16.

The relevant changes to the data from `circle.py` are:

```
weights = [1, math.sqrt(2)/2, 1]
cpArcs = [  # control points for each quarter arc
  [(1,0), (1,1), (0,1)],
  [(0,1), (-1,1), (-1,0)],
  [(-1,0), (-1,-1), (0,-1)],
  [(0,-1), (1,-1), (1,0)],
]

% some lines not shown...
curveX, curveY = bezUtils.ratBezier(ctrlPts, weights, nSamps)
```

The control point lists have *not* changed. What's new is a list of weights that reduces the emphasis of the middle control point of each list.

`ratBezier()` in bezUtils.py is very similar to `bezierCurve()` for integral curves.

```
def ratBezier(ctrlPts, weights, nSamps=100):
  if len(ctrlPts) != len(weights):
    raise ValueError("length error")
  n = len(ctrlPts) - 1
  xs = []; ys = []
  if nSamps < 2:
    nSamps = 2
```

```
for s in range(nSamps+1):
  u = s / nSamps
  denom = 0
  qx, qy = 0, 0
  for i in range(n+1):
    b = bernstein(n, i, u) * weights[i]
    denom += b
    qx += b * ctrlPts[i][0]
    qy += b * ctrlPts[i][1]
  xt = qx/denom
  yt = qy/denom
  xs.append(xt)
  ys.append(yt)
return xs, ys
```

The main alteration is inside the i loop where the weights are multiplied to each Bernstein function and the denominator is summed for use after the loop finishes.

A quadratic rational Bézier curve has the form

$$\mathbf{p}(u) = \frac{w_0\mathbf{p}_0 B_{0,2}(u) + w_1\mathbf{p}_1 B_{1,2}(u) + w_2\mathbf{p}_2 B_{2,2}(u)}{w_0 B_{0,2}(u) + w_1 B_{1,2}(u) + w_2 B_{2,2}(u)}.$$

where w_i are the weight associated with each control point.

Fig. 15.16 is a strong indicator that a circle can be exactly modeled by four rational Bézier curves, and it isn't too difficult to prove it. Consider the upper right hand quadrant where

$$\mathbf{p_0} = (1,0), \quad \mathbf{p_1} = (1,1), \quad \mathbf{p_2} = (0,1), \qquad w_0 = w_2 = 1,\ w_1 = \frac{\sqrt{2}}{2}.$$

The quadratic equation becomes

$$\mathbf{p}(u) = \frac{(1-u)^2 w_0\mathbf{p_0} + 2(1-u)uw_1\mathbf{p_1} + u^2 w_2\mathbf{p_2}}{(1-u)^2 w_0 + 2(1-u)uw_1 + u^2 w_2}.$$

Let's write $a = 1-u$ and $b = u$ for convenience (so $a+b=1$), and compute the numerator components and the denominator for a given point (x,y) on the curve. The numerator x-component:

$$N_x = a^2 \cdot 1 + 2ab \cdot w_1 \cdot 1 + b^2 \cdot 0 = a^2 + 2w_1\, ab.$$

The numerator y-component:

$$N_y = a^2 \cdot 0 + 2ab \cdot w_1 \cdot 1 + b^2 \cdot 1 = 2w_1\, ab + b^2.$$

The denominator:

$$D = a^2 + 2w_1\, ab + b^2.$$

So

$$x = \frac{N_x}{D} = \frac{a^2 + 2w_1ab}{D}, \qquad y = \frac{N_y}{D} = \frac{2w_1ab + b^2}{D}.$$

We want to show that $x^2 + y^2 = 1$, so compute the numerator of our $x^2 + y^2$:

$$N_x^2 + N_y^2 = (a^2 + 2w_1ab)^2 + (2w_1ab + b^2)^2.$$

If we let $A = a^2$, $B = b^2$, and $C = 2w_1ab$, then after several steps, we obtain

$$N_x^2 + N_y^2 - D^2 = (4w_1^2 - 2)AB.$$

Let's choose a value for w_1 so that $4w_1^2 - 2 = 0$, i.e.

$$w_1^2 = \frac{1}{2} \quad \Longrightarrow \quad w_1 = \frac{1}{\sqrt{2}} = \frac{\sqrt{2}}{2},$$

then $N_x^2 + N_y^2 - D^2 = 0$, and so $N_x^2 + N_y^2 = D^2$. Therefore $x^2 + y^2 = \frac{N_x^2 + N_y^2}{D^2} = 1$.

That proves that the rational quadratic Bézier with $\mathbf{p_0} = (1, 0)$, $\mathbf{p_1} = (1, 1)$, $\mathbf{p_2} = (0, 1)$ and weight $w_1 = \frac{\sqrt{2}}{2}$ lies exactly on the quarter arc from $(1, 0)$ to $(0, 1)$.

To model a particular conic, the weights of the curve must be as shown in Table 15.2.

ellipse	$w_1^2 - w_0w_2 < 0$
parabola	$w_1^2 - w_0w_2 = 0$
hyperbola	$w_1^2 - w_0w_2 > 0$

Table 15.2. Conic weights in a quadratic rational Bézier curve

The ability to represent a parabola using an integral Bézier curve is reflected by the fact that when $w_1^2 - w_0w_2 = 0$ then the weights can be eliminated from the rational equation.

In the circle example, we used $w_0 = w_2 = 1$ and $w_1 = \frac{\sqrt{2}}{2}$, so

$$w_1^2 - w_0w_2 = \frac{1}{2} - 1 = -\frac{1}{2} < 0,$$

so the curve is an ellipse (in fact, a circle).

15.5.8 Beyond Bézier Curves. Most textbooks that describe Bézier curves also contain chapters on two common extensions: B-splines (Basis splines) and NURBS (Non-Uniform Rational B-Splines).

B-splines add the concept of *knots* to smoothly attach multiple curves together. The utilize the same continuity ideas as in cubic splines – the use of tangent and curvature equality between the end point of one curve and the start point of the next.

NURBS add weights to B-splines in much the same way that we added them to Bézier curves in the last section.

Exercises

(1) For the given functions $f(x)$, let $x_0 = 0$, $x_1 = 0.6$, and $x_2 = 0.9$. Construct polynomials of degree one and two to approximate $f(0.45)$, and find the absolute errors.

(a) $f(x) = \cos x$
(b) $f(x) = \sqrt{1+x}$
(c) $f(x) = \ln(x+1)$
(d) $f(x) = \tan x$

(2) A census of the population of the United States is taken every 10 years. The following table lists the population, in thousands of people, from 1960 up to 2020.

Year	1960	1970	1980	1990	2000	2010	2020
Population (in thousands)	179,323	203,302	226,542	249,633	281,422	308,746	331,449

(a) Approximate the population in the years 1950, 1975, 2025, and 2030.
(b) The population in 1950 was approximately 150,697,360, and in 2025 the population was estimated to be 344,234,000. How accurate do you think your 1975 and 2030 figures are?

(3) Construct the natural cubic splines for the following data.

(a)

x	$f(x)$
−0.5	−0.0247500
−0.25	0.3349375
0	1.1010000

(b)

x	$f(x)$
0.1	−0.62049958
0.2	−0.28398668
0.3	0.00660095
0.4	0.24842440

(4) The data in the previous exercise was generated using the following functions. Calculate the errors due to the use of cubic splines.

(a) $f(x) = x^3 + 4.001x^2 + 4.002x + 1.101$; approximate $f(-1/3)$.
(b) $f(x) = x\cos x - 2x^2 + 3x - 1$; approximate $f(0.25)$.

(5) A natural cubic spline S on $[0, 2]$ is defined by

$$S(x) = \begin{cases} S_0(x) = 1 + 2x - x^3, & \text{if } 0 \le x < 1, \\ S_1(x) = 2 + b(x-1) + c(x-1)^2 + d(x-1)^3, & \text{if } 1 \le x \le 2. \end{cases}$$

Find b, c, and d.

(6) Construct enough points on the Bézier curve, whose control points are $p_0 = (4, 2)$, $p_1 = (8, 8)$, and $p_2 = (16, 4)$, to draw an accurate sketch.
(a) What degree is the curve?
(b) What are the coordinates at $u = 0.5$?

(7) A cubic Bézier curve has control points $(1, 0)$, $(3, 3)$, $(5, 5)$, and $(7, 2)$. Find the coordinate at $u = 0.25$ by applying the de Casteljau algorithm. Draw a sketch illustrating the curve and points.

(8) One property of Bernstein functions is that $\sum_i B_{i,n}(u) = 1$. (This property is displayed as a dashed line in Fig. 15.10.) Show that this property is true for $n = 2$.

(9) At what values of u do $B_{i,5}(u)$ reach a maximum?

(10) Prove that when the control points are colinear, that the resulting Bézier curve is a straight line segment.

(11) Fill in the gaps in the proof that the quadratic rational Bézier curve for a quadrant of a circle is a perfect match for that quadrant (see Section 15.5.7).

(12) Show that the equation for a quadratic rational Bézier curve for a parabola can be rewritten so that its weights w_i disappear, leaving an integral equation. *Hint*: set $a = \sqrt{w_0}$ and $b = \sqrt{w_2}$.

Answers

(1) (a) P1(x) $= -0.148878x + 1$; P2(x) $= -0.452592x^2 - 0.0131009x + 1$;
P1(0.45) $= 0.933005$; || f(0.45) - P1(0.45)|| $= 0.032558$;
P2(0.45) $= 0.902455$; || f(0.45) - P2(0.45) || $= 0.002008$
(b) P1(x) $= 0.4672511x + 1$; P2(x) $= -0.0780026x^2 + 0.4906522x + 1$;
P1(0.45) $= 1.210263$; || f(0.45) - P1(0.45)|| $= 0.006104$;
P2(0.45) $= 1.204998$; || f(0.45) - P2(045) || $= 0.000839$

(c) P1(x) = 0.874548x; P2(x) = $-0.268961x^2 + 0.955236x$;
P1(0.45) = 0.393546; || f(0.45) - P1(0.45)|| = 0.0212983;
P2(0.45) = 0.375392; || f(0.45) - P2(0.45)|| = 0.003828

(d) P1(x) = 1.031121x; P2(x) = $0.615092x^2 + 0.846593x$;
P1(0.45) = 0.464004; || f(0.45) - P1(0.45)|| = 0.019051;
P2(0.45) = 0.505523; || f(0.45) - P2(0.45)|| = 0.022468

(3) (a)

i	a_i	b_i	c_i	d_i
0	−0.02475000	1.03237500	0.00000000	6.50200000
1	0.33493750	2.25150000	4.87650000	−6.50200000

(b)

i	a_i	b_i	c_i	d_i
0	−0.62049958	3.45508693	0.00000000	−8.9957933
1	−0.28398668	3.18521313	−2.69873800	−0.94630333
2	0.00660095	2.61707643	−2.98262900	9.9420966

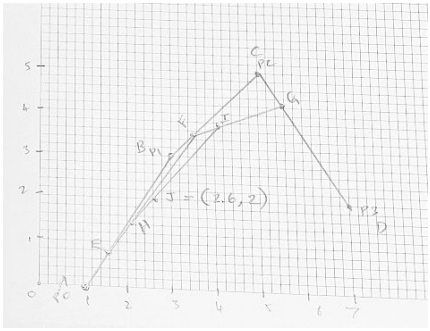
(4) x; Approx to $f(x)$; Actual $f(x)$; Error

(a) -1/3; 0.1774144; 0.17451852; 2.8959×10^{-3}

(b) 0.25; -0.1315912; -0.13277189; 1.1807×10^{-3}

(5) $b = -1, c = -3, d = 1$

(7)



$E = (1.5, 0.75), F = (3.5, 3.5), G = (5.5, 4.3),$
$H = (2, 1.4), I = (4.1, 3.7),$
$J = (2.6, 2).$

(10) When the control points are colinear the convex hull is a line segment. The Bézier curve is contained inside that line segment, so must itself be a line segment.

(12) The rational quadratic Bézier with control points P_0, P_1, P_2 and weights w_0, w_1, w_2 is

$$P(u) = \frac{w_0(1-u)^2 P_0 + 2w_1(1-u)uP_1 + w_2u^2P_2}{w_0(1-u)^2 + 2w_1(1-u)u + w_2u^2}, \qquad u \in [0,1].$$

Assuming the parabola condition $w_1^2 = w_0w_2$, then set $a = \sqrt{w_0}$ and $b = \sqrt{w_2}$. Then $w_1 = ab$ and the denominator factors as a perfect square:

$$D(u) = w_0(1-u)^2 + 2w_1(1-u)u + w_2u^2 = \big(a(1-u) + bu\big)^2.$$

The numerator becomes

$$\begin{aligned} N(u) &= a^2(1-u)^2P_0 + 2ab(1-u)tP_1 + b^2u^2P_2 \\ &= \big(a(1-u) + bu\big)^2\Big[(1-t)^2P_0 + 2ut(1-t)P_1 + t^2P_2\Big], \end{aligned}$$

if we define

$$t = \frac{bu}{a(1-u) + bu}.$$

(Indeed $1 - t = \dfrac{a(1-u)}{a(1-u) + bu}$, so the usual Bernstein quadratic appears.) Therefore

$$P(u) = \frac{N(u)}{D(u)} = (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2,$$

i.e. the rational curve equals an ordinary (polynomial) quadratic Bézier evaluated at the parameter t. The mapping $u \mapsto t$ is a smooth reparametrization of $[0,1]$ (it sends $u = 0 \mapsto t = 0, u = 1 \mapsto t = 1$) making it the same quadratic Bézier curve with control points P_0, P_1, P_2.

16

Conic Sections

Why Study Conic Sections?

- **They are key geometrical entities.** Conics are the curves obtained by slicing through a double cone, but also appear as the loci derived from simple distance properties. These definitions make them central in analytic geometry.

- **Their reflective and focusing properties.** Parabolic reflectors focus incoming rays to a single point. Ellipses reflect between two foci. These properties are essential in telescope mirrors, satellite dishes, headlights, and microphones.

- **As simple models for real-world phenomena.** Projectile motion (with no air resistance) is parabolic. Certain wavefronts and stress curves have conic shapes.

- **To describe motion in classical mechanics.** Planetary motion, satellite trajectories, and comet paths all follow conic geometries.

16.1 Introduction

One of the fascinating aspects of conic sections – ellipses (including circles), parabolas, and hyperbolas – is their definability in so many different ways. There's

the original geometric interpretation as curves formed when a plane cuts a double cone at various angles, a locus-distance construction where each conic section is a set of points satisfying a slightly different distance property, the focus-directrix (or eccentricity) approach where the points are obtained using the ratio of the distances to a focus and directrix, as quadratic equations of the form $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$, versions in terms of trigonometric and hyperbolic parameters, and definitions using polar coordinates.

In this chapter, we'll also look at how Dandelin spheres (a 19th century geometric construction) can be used to prove the equivalence of the cone-based and focus-directrix approaches, how reflections are handled by ellipses and parabolas, and consider the beauty of confocal curves.

Conics are almost certainly covered by every textbook with a title containing a phrase similar to "analytic geometry". An excellent example is *Calculus and Analytical Geometry* by Thomas et. al [**TFW96**], in particular the 9th edition which includes an entire chapter on conic sections. We also recommend Yates' *A Handbook on Curves and their Properties* [**Yat47**] and the discussion of conic section reflection in chapter 5 of Higgins' *Mathematics for the Imagination* [**Hig02**].

16.2 A Geometric Definition

The original definition of conic sections by Menaechmus (c.375-325 BC) is based on planes intersecting a cone at varying angles, although Apollonius (c.200 BC) is credited with introducing the double cone construction. An ellipse cuts obliquely across one of the cone's nappes, a parabola is parallel to the cone's edge, and a hyperbola slices both nappes (see Fig. 16.1).

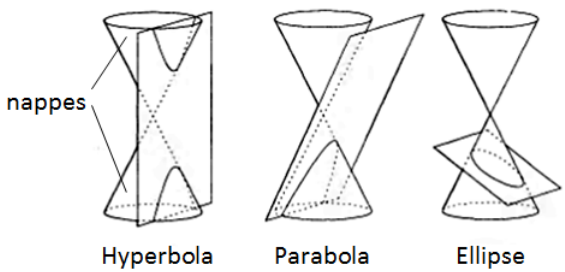


Figure 16.1. The Hyperbola, Parabola, and Ellipse

The circle is generally considered a variant of the ellipse, and there are three degenerate cases, shown in Fig. 16.2.

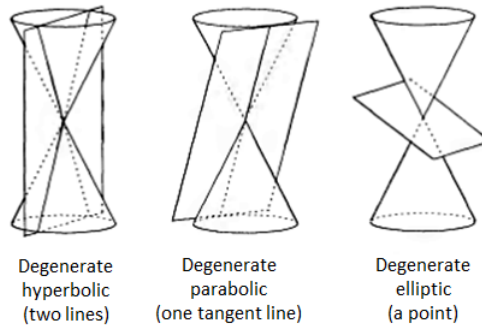


Figure 16.2. Degenerate Hyperbola, Parabola, and Ellipse

16.2.1 Classic Geometric Problems. Menaechmus probably first conceived of conic sections as a step to solving the three famous geometric problems of antiquity – trisecting an angle, doubling a cube, and squaring a circle (Fig. 16.3).

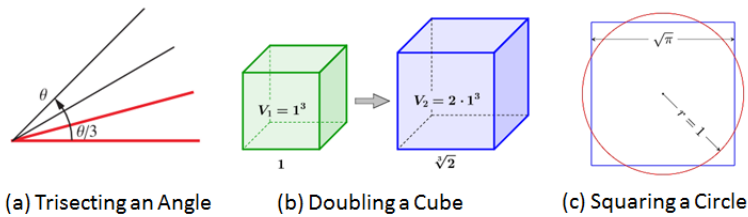


Figure 16.3. Three Classic Geometric Problems

For example, cube doubling requires the construction of an edge of a second cube whose volume is double that of the first. This can be addressed using conic sections in a variety of ways, as suggested by Fig. 16.4. Using a parabola and a hyperbola, as in Fig. 16.4(a), was Menaechmus's solution.

Famous Problems of Geometry and How to Solve Them by Benjamin Bold is an excellent brief introduction to these problems through the use of exercises [**Bol12**], while *A History of Mathematical Impossibility* by Jesper Lützen gives a historical presentation [**Lüt22**].

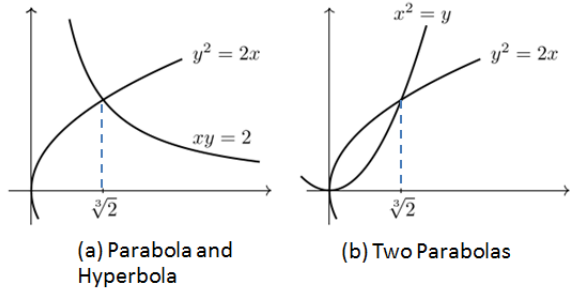


Figure 16.4. Two Cube Doublings Using Conics Sections

16.3 A Locus-Distance Definition

The locus-distance definition of conic sections is usually grouped with the focus-directrix approach described in the next section, but is a little simpler (not requiring directrixes for ellipses and hyperbolas), and suggests ways that these curves can be drawn. The equations are summarized in Fig. 16.5.

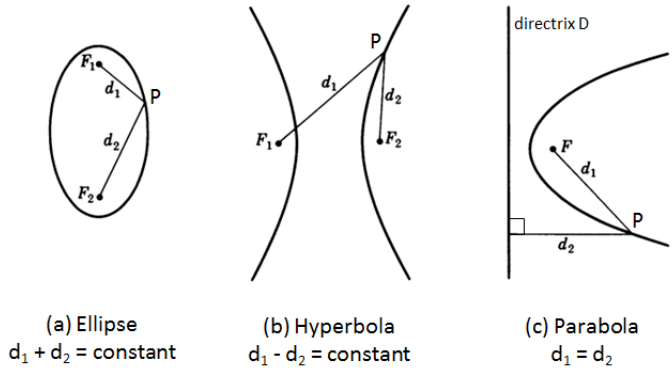


Figure 16.5. Locus/Distances for the Ellipse, Hyperbola, and Parabola

16.3.1 Parabolas. A parabola consists of all the points equidistant from a fixed point (the focus F) and a fixed line (the directrix D). In Fig. 16.5(c), the parabola is formed from the points P where $d_1 = d_2$.

More algebraically, suppose that the focus is the point $F(0, p)$ on the positive y-axis and that the directrix is the line $y = -p$ (Fig. 16.6).

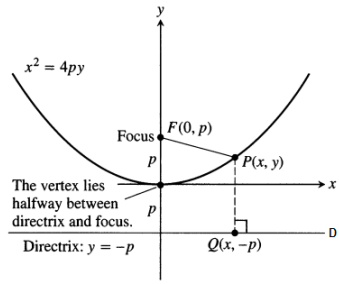


Figure 16.6. Locus/Distances for the Parabola

$P(x, y)$ lies on the parabola if and only if $PF = PQ$, which permits the distance formula to be written as

$$\begin{aligned} PF &= \sqrt{(x-0)^2 + (y-p)^2} = \sqrt{x^2 + (y-p)^2} \\ PQ &= \sqrt{(x-x)^2 + (y-(-p))^2} = \sqrt{(y+p)^2} \end{aligned}$$

We equate these expressions, square, and simplify to get

$$y = \frac{x^2}{4p} \quad \text{or} \quad x^2 = 4py$$

We can obtain similar equations for parabolas opening downwards, to the right, and to the left.

16.3.2 Ellipses. An ellipse is the set of points whose distances from two fixed points (its foci, F_1 and F_2) have a constant sum. In Fig. 16.5(a), the ellipse is formed from the points P where $d_1 + d_2 =$ a constant. Incidentally, the figure suggests a simple way to draw an ellipse – by tying a string to F_1 and F_2, and moving a pencil held tight against the string around the foci. We'll return to the issue of drawing conic sections using the locus-distance relations at the end of this section.

Suppose that the the foci are $F_1(-c, 0)$ and $F_2(c, 0)$ (Fig. 16.7), and $PF_1 + PF_2 = 2a$, then the coordinates of P on the ellipse satisfy the equation

$$\sqrt{(x+c)^2 + y^2} + \sqrt{(x-c)^2 + y^2} = 2a$$

Move the second square root to the right-hand side, square, isolate the remaining root, and square again, to obtain:

$$\frac{x^2}{a^2} + \frac{y^2}{a^2 - c^2} = 1.$$

Since $PF_1 + PF_2$ is greater than the length F_1F_2 (due to the triangle inequality), then $2a$ is greater than $2c$. Accordingly, $a > c$ and $a^2 - c^2$ in the above equation is positive.

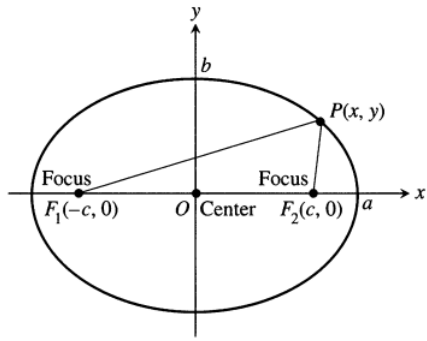


Figure 16.7. Locus/Distances for the Ellipse (algebraically)

Often we define $b = \sqrt{a^2 - c^2}$, where $(0, \pm b)$ are the points where the ellipse cross the y-axis; the equation becomes

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1.$$

The ellipse's *major axis* is the line segment joining the points $(\pm a, 0)$. The *minor axis* is the line segment joining the points $(0, \pm b)$. a is called the semi-major axis, b the semi-minor axis, and c the center-to-focus distance.

16.3.3 Hyperbola. A hyperbola (see Fig. 16.5(b)) is the set of points whose distances from two focus points have a constant *difference*. If the foci are $F_1(-c, 0)$ and $F_2(c, 0)$ (Fig. 16.8) and the constant difference is $2a$, then a point $P(x, y)$ lies on the hyperbola if and only if

$$\sqrt{(x + c)^2 + y^2} - \sqrt{(x - c)^2 + y^2} = \pm 2a.$$

Move the second square root to the right-hand side, square, isolate the remaining root, and square again, to produce:

$$\frac{x^2}{a^2} + \frac{y^2}{a^2 - c^2} = 1.$$

This looks uncannily like the equation for an ellipse, but $a^2 - c^2$ is negative because $2a$, being the difference of two sides of $\triangle PF_1F_2$, is less than $2c$, the third side.

If we let b be the positive square root of $c^2 - a^2$, then $a^2 - c^2 = -b^2$, and the equation becomes

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1.$$

Note that the hyperbola crosses the x-axis at the points $(\pm a, 0)$ (labeled as A_1 and A_2), and the hyperbola's constant difference $(d_1 - d_2)$ is $2a$. In other words,

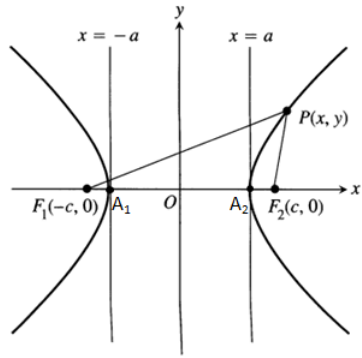


Figure 16.8. Locus/Distances for the Hyperbola (algebraically)

$2a$ is also the shortest perpendicular distance between the two branches of the hyperbola. Also, the distances F_1A_1 and F_2A_2 have the same $|c - a|$ value.

The hyperbola has two asymptotes, the lines

$$y = \pm\frac{b}{a}x.$$

One practical use for these lines is to help us graph the curve, as suggested by Fig. 16.9.

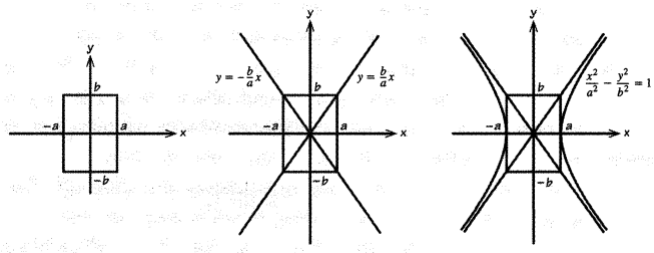


Figure 16.9. Sketching $\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1$ using Asymptotes

A quick way to find the asymptotes is to replace the 1 in the hyperbolic equation by 0 and solve for y:

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1 \;\Rightarrow\; \frac{x^2}{a^2} - \frac{y^2}{b^2} = 0 \;\Rightarrow\; y = \pm\frac{b}{a}x$$

|ɨ{z|}
0 for 1

16.3.4 Drawing Conics Sections. The locus-distance definitions for conic sections lend themselves to drawings using string, pins, and other simple tools. Fig. 16.10 illustrates the techniques.

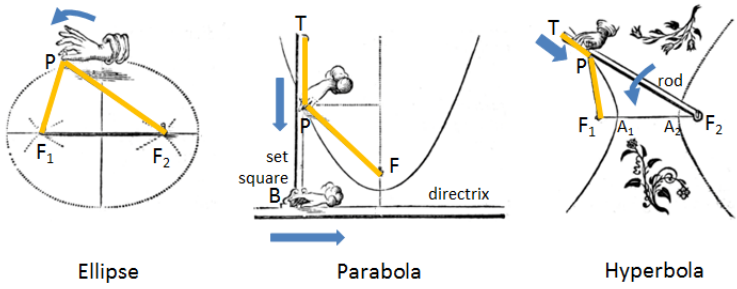


Figure 16.10. Locus/Distances Approaches for Drawing Conic Sections

The embellishments in Fig. 16.10 are due to our use of images from Frans van Schooten's *De Organica Conicarum Sectionum in Plano Descriptione Tractatus* textbook (1646). An animated version of these mechanisms can be found in "MathLapse: Constructions by pin-and-string: conics" at `https://www.youtube.com/watch?v=mldZ_7QwLvs`.

The drawing technique for the ellipse is fairly obvious, although one subtlety is that the string must be kept taut. There's a similar, but more complex requirement for the parabola: the string must be held tight against the line TP perpendicular to the directrix, and PF must also stay taut. The simplest way to see that this device actually works is to assume that the string is the same length as the set squares' height, which means that PB will always equal PF.

The sketching of a hyperbola uses a string whose length L_s equals that of the rod L_r minus the distance A_1A_2 (the shortest distance between the arms of the hyperbola:

$$L_s = L_r - A_1A_2 \quad \text{or} \quad L_r - L_s = A_1A_2$$

Recall that A_1A_2 is also the difference between the two foci distances: $d_2 - d_1 = A_1A_2 = 2a$.

Attach the rod to one of the foci (e.g. F_2 in Fig. 16.10(c)) so it can turn freely around the focus. Fasten the string to the other end of the rod (at T) and to the other focus (F_1). The user turns the rod around F_2 keeping the string tight by holding the pencil against the bar. We see that:

$$TP + PF_1 = L_s \quad \text{and} \quad TP + PF_2 = L_r$$

and so:

$$PF_2 - PF_1 = L_r - L_s = A_1A_2$$

As the rod turns, PF_1 and PF_2 change by the same amount, which means that $PF_2 - PF_1$ remains constant, or equivalently that $d_2 - d_1$ stays constant.

Although we've used van Schooten's drawings from 1620, it's worth mentioning the first printed popular textbook on geometric drawings was Albrecht Dürer's "Course in the Art of Measurement with Compass and Ruler" (*Underweysung der Messung mit dem Zirckel und Richtscheyt*), published in 1525. Aside from locus-direction drawings of conic sections, he also discussed point-by-point constructions and mechanical devices such as the Trammel of Archimedes. In other parts of the book, Dürer wrote about curves such as helices, conchoids, and epicycloids, how to draw polygons, and introduced the theory of perspective. A very nice overview of Dürer's contributions can be found in chapter 2 of Daniel Pedoe's *Geometry and the Liberal Arts* (later editions replace the word 'Liberal' with 'Visual') [**Ped83**].

16.3.5 Locus-Direction in Code. We've written three programs that reproduce the effect of drawing a parabola, ellipse, and hyperbola using the locus-direction approach: paraString.py (Fig. 16.11), ellipseString.py, and hyperbolaString.py (Fig. 16.12).

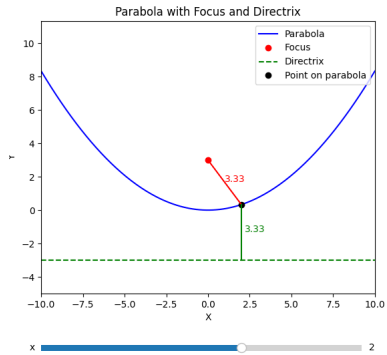
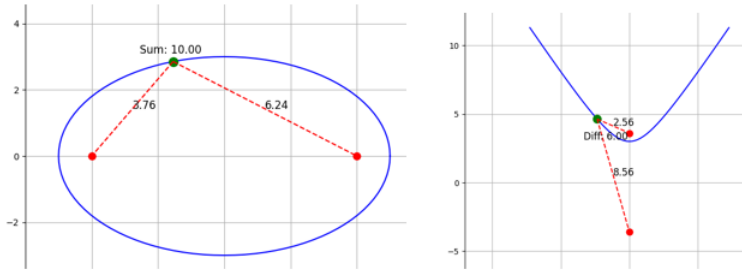


Figure 16.11. Drawing Locus/Distances for a Parabola

paraString.py lets the user slide a point along a parabola, and the distances between that point and the parabola's focus and directrix are reported. Relevant code snippets:

```
def parabola(x, d):
  return (x * x) / (4 * d)

d = 3.0  # directrix is y = -d
focus = (0, d)
# many lines not shown
```



(a) ellipseString.py (b) hyperbolaString.py

Figure 16.12. Drawing Locus-Distances for an Ellipse and a Hyperbola

```
    :
# draw curve, focus and directrix
ax.plot(xs, ys, color='blue', label='Parabola')
ax.plot([focus[0]], [focus[1]], 'ro', label='Focus')
ax.axhline(-d, color='green', ls='--', label='Directrix')
```

The slider returns an x-coordinate (`px`) which is used to calculate a parabola position, and its distances from the focus and directrix:

```
py = parabola(px, d)
distFocus = math.hypot(px - focus[0], py - focus[1])
distDirectrix = abs(py - (-d))
```

Reassuringly, the two distances, (`distFocus` and `distDirectrix`, remain equal throughout.

ellipseString.py and hyperbolaString.py use similar animation code to move a point repeatedly along an ellipse and (the upper half of) a hyperbola.

Relevant code fragments for the ellipse:

```
def ellipse(theta):
  x = semiMajor * math.cos(theta)
  y = semiMinor * math.sin(theta)
  return (x,y)

# ellipse parameters
semiMajor = 5  # axes lengths (a and b)
semiMinor = 3
c = math.sqrt(semiMajor**2 - semiMinor**2)
foci1 = (-c, 0); foci2 = (c, 0)
```

`ellipse()` is implemented using a parametric equation which we'll explain in section 16.10.3.

Matplotlib calls `animate()` to draw each frame of the animation. It converts the frame number into an angle which is passed to `ellipse()` to get the current

position of the moving point. The distances of that point from the two foci are reported (`len1` and `len2`), along with the sum (`sumLen`):

```
# animation for frame i
theta = 2*math.pi * i/100
xDot, yDot = ellipse(theta)
# many lines not shown
    :
len1 = math.sqrt((xDot - foci1[0])**2 + (yDot - foci1[1])**2)
len2 = math.sqrt((xDot - foci2[0])**2 + (yDot - foci2[1])**2)
sumLen = len1 + len2
```

Pleasingly, `sumLen` always retains the same value, 10, which is 2 × `semiMajor` ($2a$).

In hyperbolaString.py, the important parts of the code are:

```
def hyperbola(theta):
  x = semiMinor * math.sinh(theta)
  y = semiMajor * math.cosh(theta)
  return (x,y)

# vertical hyperbola parameters
semiMajor = 3  # axes lengths (a and b)
semiMinor = 2
c = math.sqrt(semiMajor**2 + semiMinor**2)
foci1 = (0, -c); foci2 = (0, c)
```

`hyperbola()` is implemented using a parametric equation that we'll explain in section 16.10.4.

Matplotlib calls `animate()` to draw each frame. The frame number is converted into an angle, and passed to `hyperbola()` to get the point's current position. The point's distances from the two foci are reported, (`len1` and `len2`), along with their absolute difference (`diffLen`):

```
# animation for frame i
theta = (i - 20)/10
xDot, yDot = hyperbola(theta)
# many lines not shown
    :
len1 = math.sqrt((xDot - foci1[0])**2 + (yDot - foci1[1])**2)
len2 = math.sqrt((xDot - foci2[0])**2 + (yDot - foci2[1])**2)
diffLen = abs(len1 - len2)
```

`diffLen` always reports the same value, 6, which is 2 × `semiMajor` ($2a$).

16.4 Classifying Conic Sections by Eccentricity

We have Pappus (c.320 AD) to thank for the focus–directrix definition of conics, which appeared in his multi-volume *Collection*, specifically Book VII, which describes the works of Euclid, Apollonius, Aristaeus, Eratosthenes, and others.

The *Collection* as a whole is a compendium of topics that were part of the classical mathematics curriculum, including geometry, astronomy, and mechanics. It had a significant impact on 17th century mathematics after being translated into Latin in 1588. For instance, it inspired Descartes' work on analytic geometry, specifically the quadratic equation treatment of conic sections that we'll meet later.

The focus–directrix definition states that $\frac{PF}{PD} = e$, where P is any point on the conic section, F a focus, D a directrix, and e (the eccentricity) is constant. The type of a conic section can be determined solely by considering the value of e:

- a parabola if $e = 1$,
- an ellipse if $e < 1$, and
- a hyperbola if $e > 1$.

Note that the parabola's definition is essentially unchanged from its locus-distance expression, but by utilizing eccentricity all of the conic sections can be formulated in a uniform way.

For the ellipse and hyperbola:

$$\texttt{Eccentricity} = \frac{\texttt{distance between foci}}{\texttt{distance between vertices}}$$

A more modern expression is

$$e = \frac{c}{a}.$$

This is equivalent to the more wordy definition since the distance between the foci is $2c$ and the distance between the vertices is $2a$ (see Fig. 16.13and Fig. 16.15).

Another version is:

$$PF_1 = e \cdot PD_1, \quad PF_2 = e \cdot PD_2$$

where F_1 and F_2 are the foci and D_1 and D_2 are the directrixes.

16.4.1 Ellipse. There are several ways to visualize e for an ellipse, as shown in Fig. 16.13.

The equivalence of

$$PF_1 = e \cdot PD_1, \quad PF_2 = e \cdot PD_2$$

and $e = \frac{c}{a}$ is easiest to see by considering when P is at a vertex, between a focus and its directrix (see Fig. 16.14).

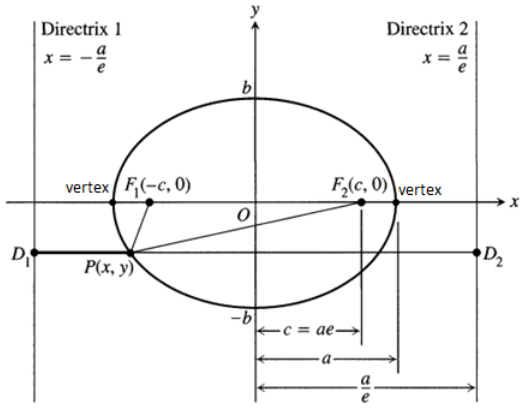


Figure 16.13. An Ellipse's Eccentricity

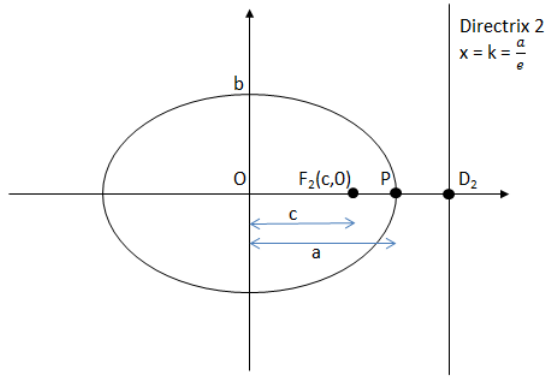


Figure 16.14. An Ellipse with P at a Vertex

In that case

$$\begin{aligned} e &= \frac{PF_2}{PD_2} = \frac{a-c}{(a/e)-a} \\ e((a/e)-a) &= a-c \\ a-ae &= a-c \\ ae &= c \\ e &= \frac{c}{a} \end{aligned}$$

If we fix a and vary c over the interval $0 \leq c \leq a$, then the ellipse will be circular when $c = 0$ (which means that $a = b$) and flattens as c increases. If $c = a$, the ellipse degenerates into a line segment.

16.4.2 Hyperbola. The eccentricity of the hyperbola $(x^2/a^2) - (y^2/b^2) = 1$ is:

$$e = \frac{c}{a} = \frac{\sqrt{a^2 + b^2}}{a}$$

As with the ellipse, there are several ways to visualize e, as shown in Fig. 16.15.

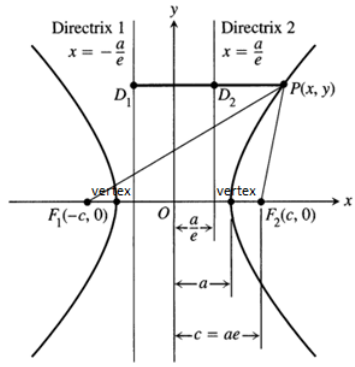


Figure 16.15. A Hyperbola's Eccentricity

16.5 Polar Coordinates

Every Cartesian point P can be assigned a polar coordinate (r, θ), where r is the distance from the origin O to P, and θ is the angle measured counterclockwise from the initial direction (typically the positive x-axis) to OP. This angle isn't unique, so a Cartesian point can have infinitely many polar coordinates. For instance, a point 2 units from the origin at $\theta = \pi/6$ has the polar coordinate $P(2, \pi/6)$, and also $P(2, -11\pi/6)$, $P(2, 13\pi/6)$, etc, in multiples of $\pm 2\pi$.

As a further source of ambiguity, r may sometimes be negative. For example, $P(2, 7\pi/6)$ can be reached by turning $7\pi/6$ radians counterclockwise and going forward 2 units, but it can also be located by turning only $\pi/6$ radians counterclockwise and going backward 2 units, resulting in $P(-2, \pi/6)$.

16.6 Ellipses, Parabolas, and Hyperbolas Unified

To formulate polar equations for the ellipse, parabola, and hyperbola, we place one focus at the origin and its directrix to the right of the origin at $x = k$ (Fig. 16.16).

$PF = r$ and $PD = k - FB = k - r\cos\theta$, which means that the conic section's focus-directrix equation $PF = e \cdot PD$ becomes

$$r = e(k - r\cos\theta),$$

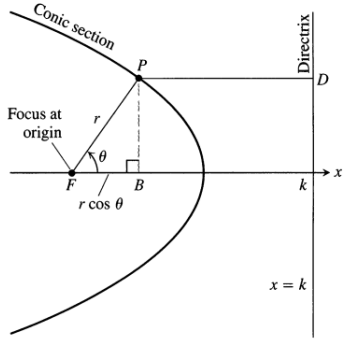


Figure 16.16. A Polar Equation for a Conic Section

which can be rearranged as:

$$r = \frac{ke}{1 + e\cos\theta}.$$

This represents an ellipse if $0 < e < 1$, a parabola if $e = 1$, and a hyperbola if $e > 1$. In other words, polar coordinates and eccentricities let us define all of the conic sections with a single equation.

You may see variations of this equation depending on the location of the directrix. For instance, if the directrix is $x = -k$ to the left of the origin, we get

$$r = \frac{ke}{1 - e\cos\theta}.$$

If the directrix is relative to the y-axis (as $y = k$ or $y = -k$ in Fig. 16.17), then the equation will use sines instead of cosines.

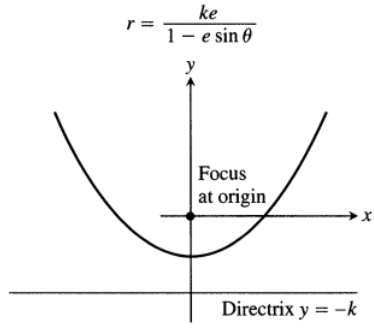


Figure 16.17. A Polar Sine Equation for a Conic Section

One variation is to replace k by the semi-major axis a. This is easy but we must remember that the polar equations so far have assumed the focus is at the origin, which is not the case in Fig. 16.13 and Fig. 16.15. In Fig. 16.13,

$$k = \frac{a}{e} - ea.$$

which can be rearranged as $ke = a(1 - e^2)$. Then the polar equation becomes:

$$r = \frac{a(1 - e^2)}{1 + e\cos\theta}$$

Notice that when $\theta = 0$, this equation becomes $r = a$, which represents a circle.

Yet another alternative is to utilize the conic section's *latus rectum*. This is the line segment that passes through a focus, perpendicular to the major axis, with its endpoints on the curve (Fig. 16.18). The *semi*-latus rectum, l, is half of the latus rectum, representing the distance from the focus to one of the line's endpoints.

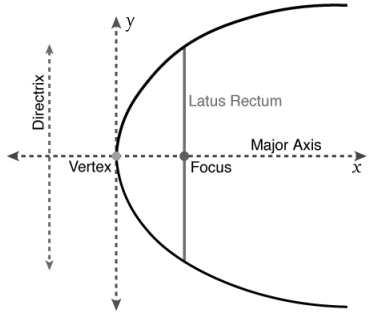


Figure 16.18. The Latus Rectum of a Conic Section

For the parabola $y^2 = 4ax$, $l = 2a$, while for the ellipse and hyperbola, $l = b^2/a$. However, since we're dealing with eccentricity, e, and a $x = \pm k$ directrix, it's more useful to express it as $l = ke$. This gives us the polar equation:

$$r = \frac{l}{1 + e\cos\theta}.$$

16.7 Drawing Conic Sections using their Eccentricity

We've implemented two programs, conicsEccen.py and conicsPolar.py, that draw conic sections with polar coordinates.

`conicsEccen.py` employs $r = l/(1 + e\cos(\theta))$ but keeps l constant ($l = 1$), and fixes one focus at the origin. However, e is adjustable via a slider, which

permits the curve to change between a circle, ellipse, parabola, and hyperbola. As it transforms, the directrix associated with the fixed focus moves, and a second focus and directrix will appear and disappear.

The second focus and directrix move to the left as the eccentricity grows from 0 towards 1, eventually reaching $-\infty$ as the ellipse becomes a parabola (see Fig. 16.19). As the eccentricity increases above 1, the eccentricity returns from $+\infty$ as a hyperbola. The focus and directrix reappear on the right of the window as the right hand branch of the hyperbola moves towards its left branch.

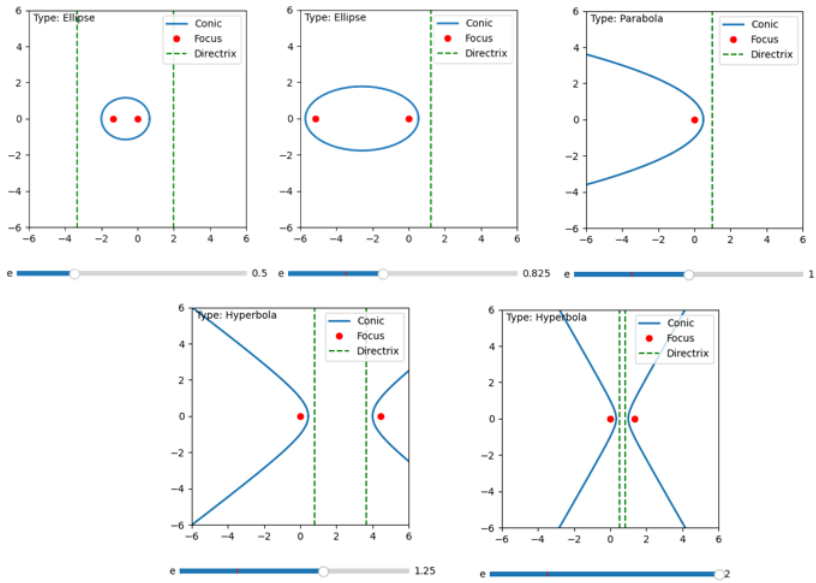


Figure 16.19. `conicsEccen.py`: increasing e

By referring to Fig. 16.13 and Fig. 16.15, it's clear that the second focus is at the same distance from the center at the fixed focus at the origin, and so is positioned at $(\pm 2c, 0)$ depending on whether the curve is an ellipse or hyperbola. The distance between the fixed focus and its directrix is k, a distance also used by the second focus and directrix. So the second directrix will be at $x = \pm 2c - k$. Since $l = ke$, this can be rewritten as $x = \pm 2c - l/e$, which is preferable since l is constant.

`conicType()` in conicsEccen.py converts e into a string:

```
def conicType(e):
  if abs(e) < 1e-6:  # v.close to 0
    return "Circle"
```

```
  elif e < 1:
    return "Ellipse"
  elif abs(e - 1) < 1e-6:  # v.close to 1
    return "Parabola"
  else:
    return "Hyperbola"
```

This string is used in `update()` which redraws the plot whenever the slider is moved. The following code fragment shows how the second focus (`focus2Dot`) and directrix (`drx2`) are calculated:

```
a = l/abs(1-e*e)
c = a*e
if eType == "Hyperbola":
  focus2Dot.set_data([2*c], [0])
  drxX2 = 2*c-(l/e) if e != 0 else 1e6
else: # ellipse
  focus2Dot.set_data([-2*c], [0])
  drxX2 = -2*c-(l/e) if e != 0 else 1e6

focus2Dot.set_visible(True)
drx2.set_data([drxX2, drxX2], [-10, 10])
drx2.set_visible(True)
```

`set_visible()` controls the visibility of the second focus and directrix, which aren't needed when a circle or parabola are being drawn.

conicsPolar.py draws a predetermined ellipse, parabola, or hyperbola with the polar equation $r = l/(1 + e\cos(\theta))$, as shown in Fig. 16.20. Only one branch of the hyperbola is drawn, and only one focus is displayed for the ellipse and hyperbola.

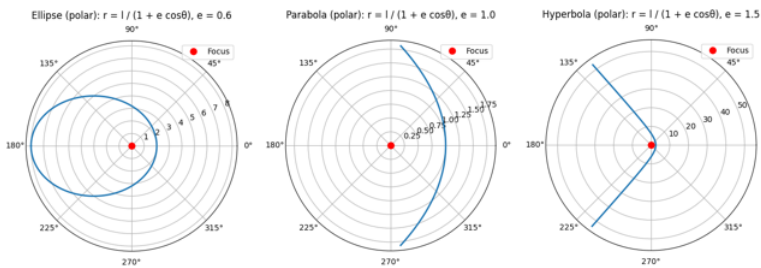


Figure 16.20. `conicsPolar.py`: ellipse, parabola, or hyperbola

The main purpose of this program is to show off matplotlib's polar plotting capabilities since `conicsEccen.py` converts polar coordinates to Cartesian values before they are graphed.

User input of 'e', 'p', or 'h' leads to a three-way branch that draws one of the graphs. The hyperbola code is:

```
elif choice == 'h':
  e = 1.5
  a = 5
  thetaLimit = math.acos(-1/e) - 0.1
  thetaRange = (-thetaLimit, thetaLimit)
  thetas, rs = conicPts(e, a, thetaRange=thetaRange)
  label = "Hyperbola"
```

Note that e and a are fixed before the curve is plotted, while `conicsEccen.py` fixes e and l.

`conicPts()` returns a list of θ's and r's for the polar coordinates of one branch of the hyperbola by using $r = l/(1 + e\cos\theta)$. l is calculated using $l = a(1 - e^2)$, and must deal with $\pm$ values of l depending on the conic type, set a sensible range for the generated angles, and avoid division by zero:

```
def conicPts(e, a, step=0.01, thetaRange=None):
  if e < 1:     # Ellipse
    l = a * (1 - e**2)
  elif abs(e - 1) < 1e-6:  # parabola; v.close to 1
    l = a
  else:         # Hyperbola
    l = a * (e**2 - 1)

  if thetaRange is None:
    thetaStart = -math.pi + 0.0001
    thetaEnd = math.pi - 0.0001
  else:
    thetaStart, thetaEnd = thetaRange

  thetas = []
  rs = []
  theta = thetaStart
  while theta <= thetaEnd:
    denom = 1 + e * math.cos(theta)
    if abs(denom) > 1e-6:  # not close to 0
      r = l / denom
      thetas.append(theta)
      rs.append(r)
    theta += step
  return thetas, rs
```

16.8 Dandelin Spheres

What's the connection between the focus-directrix definition of a conic section and its geometric definition in terms of a sliced cone? The first proof linking the

two was devised by the Belgian mathematician G.P. Dandelin in 1822. It requires a lemma from solid geometry, Oglivy's Wastebasket Theorem: "Two planes parallel to the base of a right circular cone cut off equal segments on all the rulings" [**Ogi68**]. The rulings can be thought of as the slants of the wastebasket in Fig. 16.21.

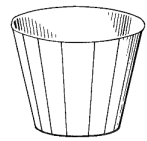


Figure 16.21. The Wastebasket Theorem Illustrated

16.8.1 The Ellipse. The Dandelin proof for an ellipse employs Fig. 16.22.

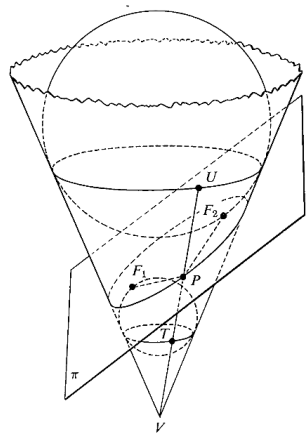


Figure 16.22. Dandelin Spheres and the Ellipse

Let π be a plane that cuts the cone to form an oval. A sphere will fit tightly inside the cone, tangent to it and also to π, touching it at F_1. A second sphere is tangent to π from above, touching it at F_2. F_1 and F_2 will turn out to be the foci of the ellipse.

Choose any point P on the oval. Draw PF_1 and PF_2, which lie in π and are therefore tangent to the lower and upper spheres respectively. Also draw the line $VTPU$ on the surface of the cone, making it tangent to both spheres. Now because tangents to a sphere from an external point P are equal, we have

$$PF_1 = PT \quad \text{and} \quad PF_2 = PU$$

Adding,

$$PF_1 + PF_2 = PT + PU = TU.$$

Now suppose that P is moved to a new position on the oval. TU will take a new position on another line, but by the Wastebasket Theorem, its length will not change. This means that for all positions of P,

$$PF_1 + PF_2 = \texttt{a constant},$$

the definition of an ellipse with foci at F_1 and F_2.

16.8.2 The Ellipse's Eccentricity. Fig. 16.23 shows another ellipse with a focus F and one of its Dandelin spheres.

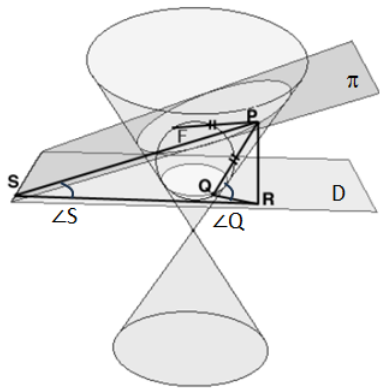


Figure 16.23. Conis Section Angles and the Ellipse

Given P, let Q be the corresponding point on the Dandelin circle; that is, let Q be the point where the segment joining P to the cone apex meets the Dandelin plane D. Let R be the foot of the perpendicular dropped from P to the Dandelin plane, and let S be the foot of the perpendicular dropped from P to the directrix along the plane π.

Since segments PF and PQ are both tangents to the Dandelin sphere, we have $PF = PQ$. But this time we manipulate the focus-directrix ratio for P like so:

$$\frac{PF}{PS} = \frac{PQ}{PS} = \frac{PR/\sin\angle Q}{PS} = \frac{PR}{PS}\frac{1}{\sin\angle Q} = \frac{\sin\angle S}{\sin\angle Q}$$

$\angle S$ is constant as P moves about the curve because it's the angle between the cutting plane π and the Dandelin plane D. But $\angle Q$ is also constant since it's the angle that the surface of the cone makes with D. Therefore, the focus-directrix ratio is also a constant, which again proves that the curve is an ellipse.

We want to go a step further and extract an expression for the eccentricity e in terms of these angles. For now, we'll assume that the conic section is an ellipse (which makes $\angle S$ smaller than $\angle Q$).

It helps to examine the figure side-on (see Fig. 16.24), reducing all of the elements to their intersections with the plane through the cone's axis, perpendicular to the directrix.

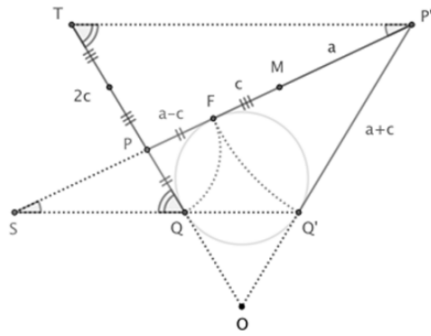


Figure 16.24. Conic Section Angles and the Ellipse from the Side

P is the point on the conic closest to the directrix (which has been projected onto the point S), and P' the farthest point. (Q and Q' are the corresponding points on the Dandelin plane, which has been projected into a line.) Then PP' is the major axis of the ellipse. The ellipse's focus, F, corresponds to the point where the incircle of $\triangle OPP'$ meets PP'; the ellipse's center corresponds to M, the midpoint of PP'.

Now that we know where everything is, a few more applications of the equal-tangent-segment property is all we need. With $a = MP'$ and $c = MF$, we have

$$\frac{\sin\angle S}{\sin\angle Q} = \frac{PT}{PP'} = \frac{QT - QP}{2a} = \frac{(a+c)-(a-c)}{2a} = \frac{2c}{2a} = \frac{c}{a} = e$$

Thus, we've linked a geometric property of the conic section (the ratio of two angles) to the eccentricity of the curve.

16.8.3 The Parabola. For the parabola, the Dandelin Sphere diagram (Fig. 16.25) only employs a single sphere and focus.

Let π be the cutting plane, this time parallel to the rulings on the cone, and let α be the plane determined by the circle of tangency of the sphere. Choose any point P on the (alleged) parabola, and draw PD perpendicular to CD, the line of intersection of α and π. We need another plane, not shown, through P and parallel to α, which cuts the cone in the indicated circle. Let RS be part of

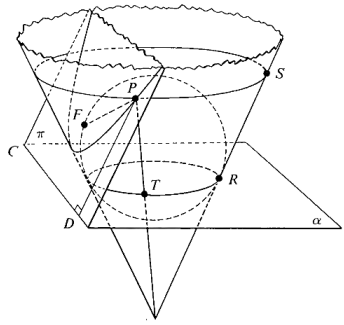


Figure 16.25. Dandelin Spheres and the Parabola

the ruling parallel to π. Then $PD = RS$, because parallel planes cut off equal segments from parallel lines. But $RS = PT$ by the Wastebasket Theorem, and $PT = PF$ because they are tangents to the sphere from P. So we have $PF = PD$, the definition of a parabola; and in addition, the directrix CD is the intersection of α and π.

16.8.4 The Hyperbola. This time we need two spheres, with the second located in the other nappe of the double cone. Since the intersecting plane is inclined near to the axis of the double cone, it meets both branches (see Fig. 16.26).

Consider the two inscribed spheres that touch both the cone and the intersecting plane. The spheres occupy different branches of the cone but lie on the same side of the plane, and so we have

$$\begin{aligned} BF_1 &= BP_1, \quad BF_2 = BP_2, \\ BF_2 - BF_1 &= BP_2 - BP_1 = P_1P_2 = \texttt{a constant}. \end{aligned}$$

16.9 Quadratic Equations

The graph of the quadratic equation

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0,$$

where A, B, and C are not all zero, is nearly always a conic section. In general, A, B, and C define the conic's shape, type, and orientation. More specifically, A controls its curvature in the x-direction, C controls the curvature in the y-direction, and B specifies the curve's rotation. If $B = 0$, then the conic section's axes are aligned with the x and y axes, but if $B \neq 0$, then it'll be rotated by an angle θ such that $\tan 2\theta = \frac{B}{A-C}$.

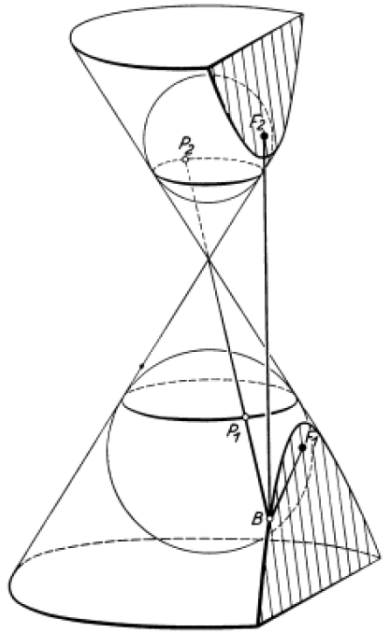


Figure 16.26. Dandelin Spheres and the Hyperbola

The shape/type is determined by the discriminant $\Delta = B^2 - 4AC$:

$$\begin{array}{lll} \Delta < 0 & \Rightarrow & \texttt{ellipse (or circle if A=C)} \\ \Delta = 0 & \Rightarrow & \texttt{parabola} \\ \Delta > 0 & \Rightarrow & \texttt{hyperbola} \end{array}$$

The D coefficient in the quadratic equation shifts the conic section horizontally, while E moves it vertically. This means that its possible to determine the conic's center (for a ellipse or hyperbola) by solving:

$$\frac{\partial}{\partial x}(Ax^2 + Bxy + Cy^2 + Dx + Ey + F) = 0$$
$$\frac{\partial}{\partial y}(Ax^2 + Bxy + Cy^2 + Dx + Ey + F) = 0$$

which reduces to two cases:

$$2Ax + By + D = 0, \quad Bx + 2Cy + E = 0$$

F controls the conic section's scaling. For an ellipse, a larger $|F|$ reduces its size, while it changes how far the branches of a hyperbola are from the origin.

Any conic section can be converted to an axes-aligned form by rotating it to eliminate Bxy, and translating it to eliminate the Dx and Ey terms. The result is a quadratic involving only $A'x^2$, $B'y^2$, and a constant.

16.9.1 Linking the Discriminant and Eccentricity. The discriminant and eccentricity for different conic sections are summarized in Table 16.1.

Conic Section	Discriminant Δ	Eccentricity e
Ellipse (includes circle)	$B^2 - 4AC < 0$	$0 < e < 1$
Parabola	$B^2 - 4AC = 0$	$e = 1$
Hyperbola	$B^2 - 4AC > 0$	$e > 1$

Table 16.1. Conic Discriminant and Eccentricity

$B^2 - 4AC$ and e can be related through the rotated and shifted forms of the quadratic equation.

After an ellipse has been rotated to remove the xy-term (making $B = 0$):

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

with $a > b$, and its eccentricity is:

$$e = \frac{c}{a} = \sqrt{\frac{c^2}{a^2}} = \sqrt{1 - \frac{b^2}{a^2}} < 1.$$

In the quadratic, A and $C > 0 \Rightarrow B^2 - 4AC < 0$.

For a left-facing, axis-aligned parabola $y^2 = 4ax$, $e = 1$, and the quadratic has $B = 0$ and one of A or $C = 0 \Rightarrow B^2 - 4AC = 0$.

The unrotated hyperbola (i.e. $B = 0$):

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1$$

has a similar eccentricity as the ellipse:

$$e = \sqrt{1 + \frac{b^2}{a^2}} > 1.$$

In the quadratic, A and C have opposite signs so $\Rightarrow B^2 - 4AC > 0$.

16.9.2 Rotating a Conic Section. When we rotate a graph's axes by an angle θ, the new coordinates (x', y') can be defined in terms of (x, y):

$$\begin{cases} x = x' \cos\theta - y' \sin\theta \\ y = x' \sin\theta + y' \cos\theta \end{cases}$$

We want to find coefficients A', B', C' so the rotated conic has the same geometric shape as before, but expressed using (x', y'):

$$A'x'^2 + B'x'y' + C'y'^2 + D'x' + E'y' + F' = 0$$

Let's substitute the x and y equations into $Ax^2 + Bxy + Cy^2$:

$$A(x'\cos\theta - y'\sin\theta)^2 + B(x'\cos\theta - y'\sin\theta)(x'\sin\theta + y'\cos\theta) + C(x'\sin\theta + y'\cos\theta)^2$$

Expand all the terms, and collect the powers of x'^2, $x'y'$, and y'^2. After (some considerable) simplification, the new coefficients are:

$$A' = A\cos^2\theta + B\cos\theta\sin\theta + C\sin^2\theta,$$
$$B' = 2(C - A)\sin\theta\cos\theta + B(\cos^2\theta - \sin^2\theta),$$
$$C' = A\sin^2\theta - B\sin\theta\cos\theta + C\cos^2\theta.$$

If the original conic section didn't have a xy term (i.e. $B = 0$), then these simplify to:

$$A' = A\cos^2\theta + C\sin^2\theta,$$
$$B' = 2(A - C)\sin\theta\cos\theta,$$
$$C' = A\sin^2\theta + C\cos^2\theta.$$

Consider an ellipse:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \quad \Rightarrow \quad A = \frac{1}{a^2},\ C = \frac{1}{b^2},\ B = 0$$

After this ellipse is rotated by an angle θ, we get:

$$\begin{aligned} A' &= (\cos(\theta)^2/a^2) + (\sin(\theta)^2/b^2) \\ B' &= 2 * \sin(\theta) * \cos(\theta) * (1/a^2 - 1/b^2) \\ C' &= (\sin(\theta)^2/a^2) + (\cos(\theta)^2/b^2) \end{aligned}$$

16.9.3 Drawing with the Quadratic Equation. conicsQuad.py gives the user a choice of plotting one of six quadratic equations representing ellipses, parabolas, and hyperbolas:

```
1. Ellipse: x^2/a^2 + y^2/b^2 = 1
2. Rotated Ellipse (30 deg)
3. Parabola: y^2 = 4ax
4. Rotated Parabola (45 deg)
5. Hyperbola: x^2/a^2 - y^2/b^2 = 1
6. Rotated Hyperbola (30 deg)
```

Fig. 16.27 shows what's plotted for each choice.

The graphs use matplotlib's contour plotting, the standard approach when an implicit equation of the form $F(x_1, \ldots, x_n) = 0$ needs to be drawn.

The code evaluates $Ax^2 + Bxy + Cy^2 + Dx + Ey + F$ at 800×800 = 640,000 (x, y) points, creating a grid of results. The contour plot is created by:

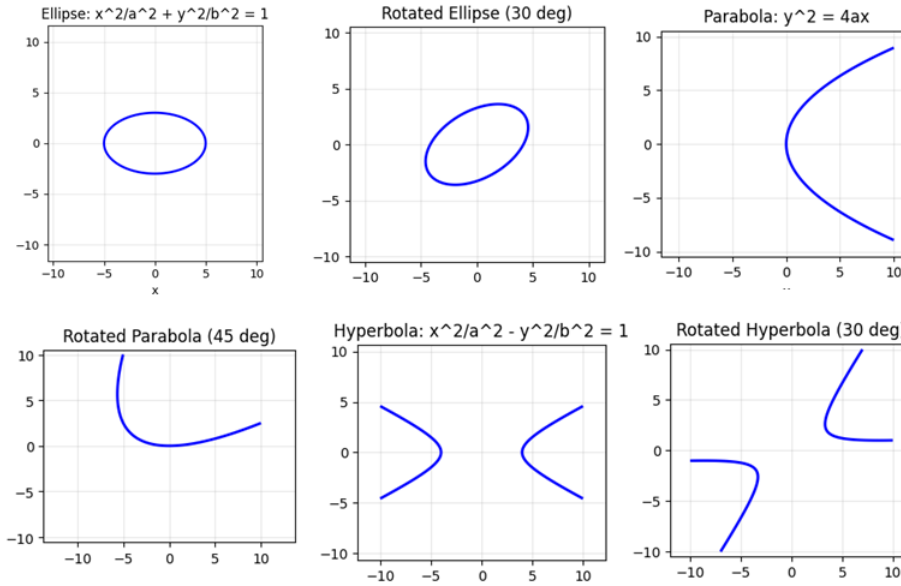


Figure 16.27. Possible `conicsQuad.py` Plots

```
contours = plt.contour(xs, ys, zs, levels=[0],
                                   colors='blue', linewidths=2)
```

`xs` and `ys` are lists of the x- and y- coordinates, and `zs` is the grid of results. The `levels` argument specifies that only the (x, y) coordinates with the result $z = 0$ will be plotted. Also, the contour algorithm doesn't just connect the coordinates where the equation equals zero, but also employs linear interpolation between adjacent (x, y) points to approximate where the zero-crossing occurs. For example, if one grid point has a z value 0.5 and its neighbor is -0.3, then the algorithm calculates that the zero must occur approximately 62.5% of the way between them $(0.5/(0.5+0.3))$. Interpolation means that the resulting contour can pass through positions that weren't supplied as (x, y) values, and the result is a smooth continuous curve

Each of the six choices assumes values for a and b, and uses them to calculate A, B, C, D, E, and F for the quadratic. For example, choice 1 (the axis-aligned ellipse) uses:

```
a, b = 5, 3
A = 1 / a**2
B = 0
C = 1 / b**2
```

```
  D = E = 0
  F = -1
```

These are passed to `plotConic()` which generates (x, y) points between -10 and 10 on both axes (coded as two lists of x- and y- values). These are passed to `conicEquation()` to generate z values:

```
def conicEquation(x, y, A, B, C, D, E, F):
  return A*x**2 + B*x*y + C*y**2 + D*x + E*y + F

# : code not shown
zs = []
for y in ys:
  row = []
  for x in xs:
    row.append( conicEquation(x, y, A, B, C, D, E, F))
  zs.append(row)
```

For a rotated conic, a hard-wired `angle` is used along with a and b to determine the quadratic's coefficients. For example, to rotate an ellipse by 30°:

```
  a, b = 5, 3
  angle = math.radians(30)
  A = (math.cos(angle)**2 / a**2) + (math.sin(angle)**2 / b**2)
  B = 2 * math.sin(angle) * math.cos(angle) * (1 / a**2 - 1 / b**2)
  C = (math.sin(angle)**2 / a**2) + (math.cos(angle)**2 / b**2)
  D = E = 0
  F = -1
```

Note that the math used here corresponds to what was worked out in the previous section.

16.10 Parameterized Conic Sections

Single-valued functions, such as $y = x^2$ are easily plotted by plugging a series of x values into the equation, and collecting the output. Unfortunately, multi-valued functions, such as those for the ellipse and the hyperbola, can be more problematic. One approach is to express the x- and y- coordinates as parameterized functions of time t, $x = f(t)$ and $y = g(t)$. Of course, this is particularly useful when studying motion (such as elliptical orbits) because a point's position is determined by time. However, t doesn't need to be time-related, it can just as readily denote an angle or a distance traveled along a curve.

A parameterized equation requires a parameter interval, such as $a \leq t \leq b$, to set the range of generated coordinates – starting at $(f(a), g(a))$, to $(f(b), g(b))$.

16.10.1 The circle $x^2 + y^2 = 1$. The equations and parameter interval

$$x = \cos t, \quad y = \sin t, \quad 0 \leq t \leq 2\pi,$$

describe the position $P(x, y)$ of a point that moves counterclockwise around the circle $x^2 + y^2 = 1$ as t increases (Fig. 16.28).

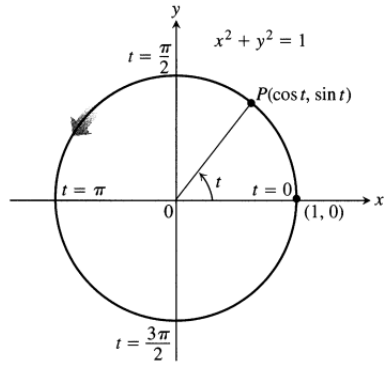


Figure 16.28. A Parameterized Circle

We know the point lies on this circle for every value of t because

$$x^2 + y^2 = \cos^2 t + \sin^2 t = 1.$$

The point starts at $(1, 0)$, moves up and to the left as t approaches $\pi/2$, and continues around the circle to stop back at $(1, 0)$ when $t = 2\pi$.

16.10.2 A Parabola. The points $P(x, y)$ forming a parabola are given by the equations and parameter interval

$$x = t, \quad y = t^2, \quad -\infty < t < \infty.$$

If we eliminate t between the equations, then $y = (t)^2 = x^2$.

As t increases from $-\infty$ to ∞, the point moves down the left arm of the curve, passes through the origin, and up the right-hand side (Fig. 16.29).

16.10.3 The ellipse, $x^2/a^2 + y^2/b^2 = 1$. The points forming an ellipse are given by

$$x = a\cos t, \quad y = b\sin t, \quad 0 \le t \le 2\pi.$$

The Cartesian equation is obtained by eliminating t from:

$$\cos t = \frac{x}{a}, \quad \sin t = \frac{y}{b}$$

with the help of the identity $cos^2 t + sin^2 t = 1$:

$$\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 = 1, \quad \text{or} \quad \frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

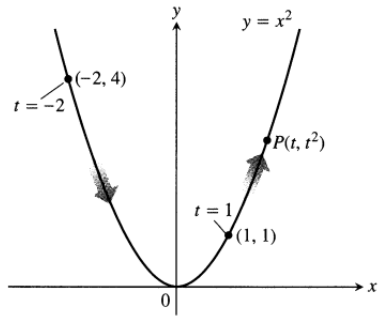


Figure 16.29. A Parameterized Parabola

When $t = 0$, the coordinates are

$$x = a\cos(0) = a, \quad y = b\sin(0) = 0,$$

so the point starts at $(a, 0)$. As t increases, the point moves counterclockwise. It traverses the ellipse once, returning to its starting position $(a, 0)$ at time $t = 2\pi$ (Fig. 16.30).

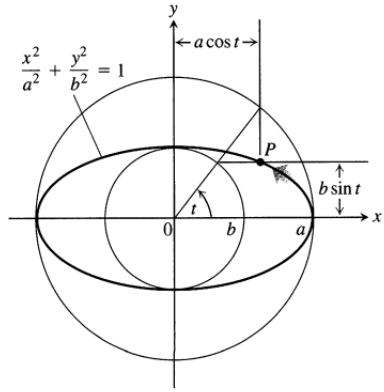


Figure 16.30. A Parameterized Ellipse

16.10.4 The Right-hand Branch of the Hyperbola, $x^2/a^2 - y^2/b^2 = 1$.

$P(x, y)$ at time t is given by

$$x = a\sec t, \quad y = b\tan t, \quad -\frac{\pi}{2} < t < \frac{\pi}{2}$$

The Cartesian equation is obtained by eliminating t, with the help of the identity $\sec^2 t - \tan^2 t = 1$, to yield $x^2/a^2 - y^2/b^2 = 1$.

As t ranges from $-\pi/2$ and $\pi/2$, $x = a \sec t$ remains positive and $y = b \tan t$ runs between $-\infty$ and ∞, so P traverses the hyperbola's right-hand branch. It comes in along the branch's lower half as $t \to 0$, reaches $(a, 0)$ at $t = 0$, and moves into the first quadrant as t increases towards $\pi/2$ (Fig. 16.31).

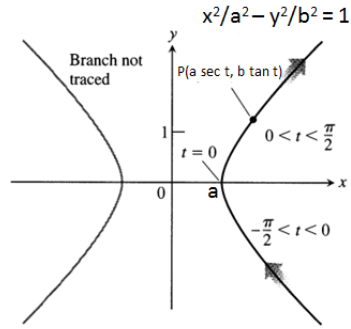


Figure 16.31. Half of a Parameterized Hyperbola

$\pi/2 < t < 3\pi/2$ represents the portion of the hyperbola in the second and third quadrants, but note that $t = \pi/2$ and $t = -\pi/2$ have to be dealt with carefully since the points are at $\pm\infty$.

There are alternative ways to parameterize a hyperbola. For example, the right-hand branch can be expressed as

$$x = a\sqrt{t^2 + 1}, \quad y = bt.$$

Or we can use hyperbolic trigonometric functions:

$$x = \pm a \cosh t, \quad y = b \sinh t.$$

We met this version back in Section 16.3.5, except that because the hyperbola was rotated so its arms turned up and down rather than to the left and right, the equations were $x = a \sinh \theta$ and $y = b \cosh \theta$.

16.11 Drawing Parameterized Equations

We've a general-purpose plotting program for parameterized equations in paramPlot.py and paramUtils.py.

`paramPlot.py` reads parameters supplied in a text file. For example:

```
python paramPlot.pt parabola.txt
```

parabola.txt contains:

```
title = parabola
x = t
y = a*t**2
tMin = -5
tMax = 5
step = 0.1
```

The parameterized equation are the two lines $x = ...$ and $y = ...$, and the interval for t is set with `tMin`, `tMax`, and `step`, while `title` is used to label the plot. These values can be omitted, and `paramPlot.py` will switch to defaults.

The plot (see Fig. 16.32) includes two sliders for `a` and `b` variables which may appear in the parameterized equations. We only use `a` in the parabolic equation given above.

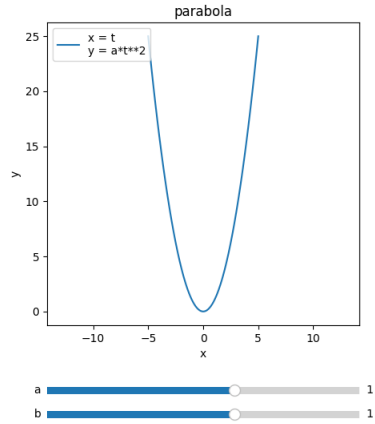


Figure 16.32. Plot of a Parameterized Parabola

The parameter files for the ellipse and hyperbola are ellipse.txt and hyperbola.txt. Their key lines are:

```
# ellipse.txt
a = 3
x = a * math.cos(t)
y = b * math.sin(t)
```

and

```
# hyperbola.txt
a = 3
x = a * math.cosh(t)
y = b * math.sinh(t)
```

No b is specified, which causes the plotter to use $b = 1$. The graphs are shown in Fig. 16.33.

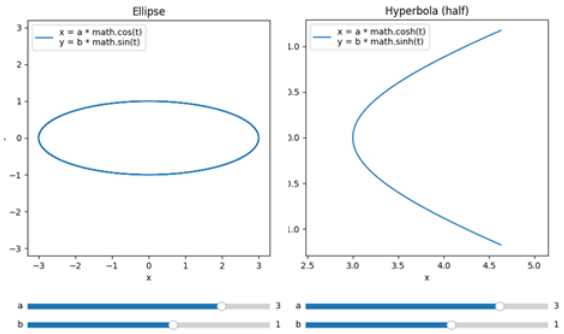


Figure 16.33. Plots of a Parameterized Ellipse and Hyperbola

16.12 Reflections in Conic Sections

The reflection problem was first considered by Heron (c.150 BC to 250 AD) in his *Catoptrica* text concerned with mirrors and the behavior of light. What position for a point P on a line L minimizes the sum of the distances $F_1P + PF_2$ between two fixed points F_1 and F_2 on the same side of the line (see Fig. 16.34)?

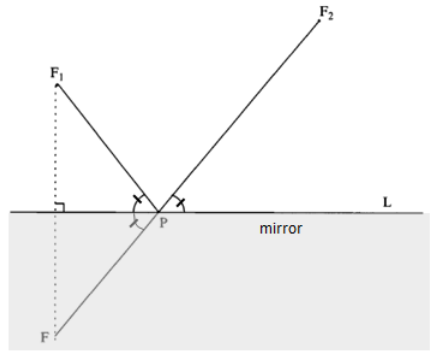


Figure 16.34. The Heron Reflection Problem

Reflect F_1P in L to create FP of the same length. The length of F_1PF_2 equals FPF_2 and so we can choose P to minimize the former by minimizing the latter.

We select P so F, P, and F_2 lie in a line, which means that P is located where FF_2 meets L.

The angles that F_1P and F_2P make with L are equal, which Heron used to infer the Law of Reflection: the angle of an incident ray equals the angle of reflection since light is naturally efficient and takes the shortest path possible via a reflecting line. Therefore, a ray travels from one point to another in the least possible time, now known as Fermat's Principle of Least Time.

16.12.1 Parabolic Reflection. We'll explain how reflection applies to a parabola by referring to Fig. 16.35. In the figure, $PF = PD$ by definition. Draw DF, bisect it at M, and connect PM. Then, $\triangle FMP$ and $\triangle DMP$ have three equal sides, and so are congruent, making $\angle 1 = \angle 2$. But $\angle 2 = \angle 3$ (they're vertical angles), so $\angle 1 = \angle 3$.

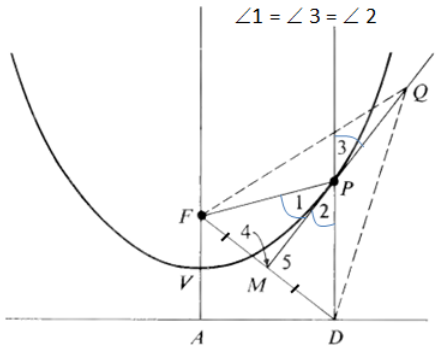


Figure 16.35. The Parabolic Reflection Problem

It remains to be shown that PM, which so far is just a line connecting two points, is actually a tangent to the parabola at P. Let's assume that it isn't, and so intersects the curve for a second time at some point Q. $\triangle FQM$ is congruent to $\triangle DQM$, because they share QM, $FM = MD$, and $\angle 4 = \angle 5$. These are corresponding parts of *thecongruent* $\triangle FMP$ and $\triangle DMP$, and so $FQ = QD$, making QD the perpendicular distance to AD by applying the definition of a parabola. In other words, Q and P must coincide, making PM a tangent at P.

This means that light entering a parabolic mirror parallel to its axis of symmetry (in Fig. 16.35 this is a ray travelling straight down to hit P) will reflect into its focus, F.

Reflecting telescopes use parabolic mirrors because they gather a great deal of light into one spot (the focus) making objects visible that otherwise would be too dim to see. Parabolic mirrors are also used in solar collectors, long distance

microphones, and receiving antennas. Going in 'reverse', light radiating out from the focus reflects to become rays parallel to the axis of symmetry, making parabolic reflectors ideal for headlights, spotlights, and directional transmitting antennas. A common feature of most science museums are two large concave parabolic dishes, serving as acoustic mirrors, facing each other in a room designated as a whispering gallery (see Fig. 16.36).

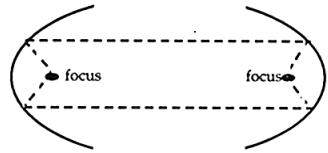


Figure 16.36. Parabolic Mirrors in a Whispering Gallery

16.12.2 Elliptic Reflection. The reflection property for an ellipse is similar to the parabolic case, but causes light reflected from one focus (F_1) to travel to the other focus (F_2) (see Fig. 16.37).

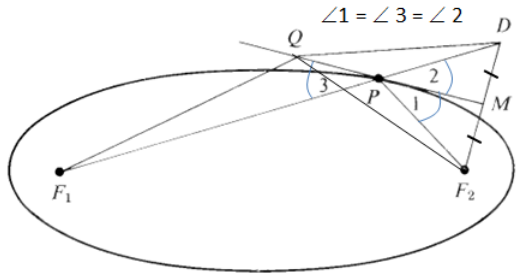


Figure 16.37. The Elliptical Reflection Problem

Focus-to-focus mirroring can be proved by extending F_1P in the figure so $PD = PF_2$. Draw DF_2, bisect it at M, and connect PM. The result, as in the parabola's case, is that $\angle 1 = \angle 3$.

To show that PM is a tangent we again assume a second point Q where PM cuts the curve (thereby implying that it isn't a tangent). $QD = QF_2$ as before, but if Q is on the curve then

$$F_1Q + QD = F_1Q + QF_2 = F_1P + PF_2 = F_1P + PD.$$

The first and last members of this equality state

$$F_1Q + QD = F_1D$$

which is impossible unless Q coincides with P.

16.12.3 Hyperbolic Reflection. The reflection property for the hyperbola is closely related to elliptic reflection, but the focus-to-focus link is less obvious. In Fig. 16.38, a beam emitted from the focal point F_2 is reflected to the right in such a way that $\angle 1 = \angle 3 = \angle 2$. If the reflected ray is extended backwards it forms a line that passes through the other focal point F_1.

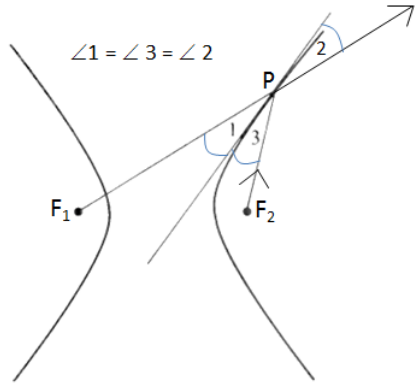


Figure 16.38. The Hyperbolic Reflection Problem

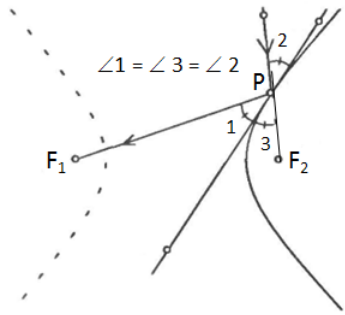


Figure 16.39. Another Hyperbolic Reflector

Fig. 16.39 shows a variant where light is projected onto the right-hand mirror from the left. The reflected ray is directed towards F_1, but if the original beam is extended, it'll pass through F_2. The physical effect is different, but it relies on the

same underlying equality as Fig. 16.38, that $\angle 1 = \angle 3 = \angle 2$. The proof is very similar to that for the elliptic reflector, so will be skipped.

Some telescopes use a hyperbolic mirror in addition to a main parabolic reflector to redirect the light from the main focus to a more convenient point (see Fig. 16.40). The parabola and hyperbola share the same focus, F_2. Light hitting the parabolic mirror reflects toward F_2, bounces off the hyperbolic mirror, and travels to its other focus at F_1. It's much easier to place a telescope eyepiece at this point rather than at F_2.

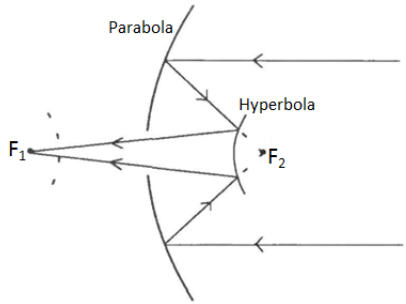


Figure 16.40. Compound Parabolic-Hyperbolic Reflector

16.12.4 Drawing Conic Reflections. paraRays.py and ellipseRays.py implement reflection by treating light rays as vectors, using our Vec.py class (see section 14.4 on quaternions).

Fig. 16.41 gives an overview of the problem: a vector **v** is reflected off the plane p around the normal vector **n**. The goal is to define the reflection **w** in terms of **v**.

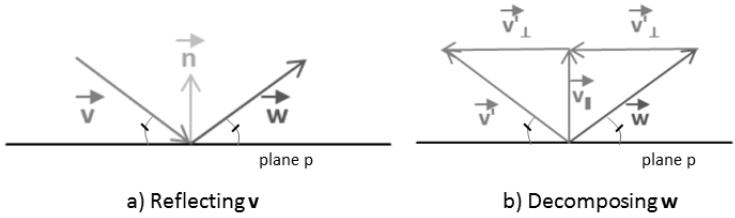


Figure 16.41. Reflection as Vectors

We start by splitting **v** into its parallel component $\mathbf{v}_{\parallel}$ and perpendicular component $\mathbf{v}_{\perp}$. Fig. 16.41(b) shows the reflection again, but with **v** negated as $\mathbf{v}'$. If

we follow the path from the start of $\mathbf{w}$ along $\mathbf{v}'$ and take two steps along $\mathbf{v}'_\perp$, we reach the head of $\mathbf{w}$. In other words:

$$\mathbf{w} = \mathbf{v}' - 2(\mathbf{v}'_\perp).$$

As $\mathbf{v}' = -\mathbf{v}$, we can rewrite $\mathbf{w}$ in terms of $\mathbf{v}$ after some work involving the dot product (see section 10.3):

$$\begin{aligned}
\mathbf{w} &= -\mathbf{v} - 2(-\mathbf{v}_\perp) \\
&= -\mathbf{v} - 2(-\mathbf{v} - (\mathbf{v}\cdot\mathbf{n})\,\mathbf{n}) \\
&= -\mathbf{v} + 2(\mathbf{v} + (-\mathbf{v}\cdot\mathbf{n})\,\mathbf{n}) \\
&= -\mathbf{v} + 2\mathbf{v} + 2(-\mathbf{v}\cdot\mathbf{n})\,\mathbf{n} \\
&= \mathbf{v} + 2(-\mathbf{v}\cdot\mathbf{n})\,\mathbf{n} \\
&= \mathbf{v} - 2(\mathbf{v}\cdot\mathbf{n})\,\mathbf{n}
\end{aligned}$$

Both `paraRays.py` and `ellipseRays.py` use vector code very like this. For example:

```
w = v - (2 * v.dot(n) * n)
```

However, $\mathbf{n}$, the normal vector, is more complex than the one in Fig. 16.41 (where $\mathbf{n} = \mathbf{v}_\parallel$) since the reflecting surface is curved.

16.12.5 The Parabolic Reflector. `paraRays.py` generates several random vectors heading downwards to hit an upward facing parabola. The reflections are calculated, and all of them pass through the parabola's focus (see Fig. 16.42).

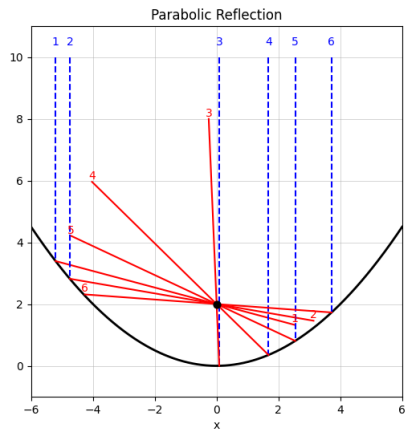


Figure 16.42. Parabolic Reflection

The following code excerpt shows how a single downward facing vector reflects off the parabola.

```
def parabolaY(x, f):
  return (x * x) / (4.0 * f)

def reflectVector(v, n):
  return v - n * (2 * v.dot(n))


downVec = Vec(0, -1, 0)  # downward vector

$ x is a random x-coordinate
yHit = parabolaY(x, f)
pHit = Vec(x, yHit, 0)  # contact with parabola

# convert parabola slope at x into a normal
m = x / (2.0 * f)
normal = Vec(-m, 1, 0).normalize()

w = reflectVector(downVec, normal)
```

The code hardwires the parabola as $y = x^2/(4f)$, which means that its slope at any x-coordinate is $m = x/(2f)$. The slope of its normal will be $-1/m$ which is converted into a vector by using $-m$ and 1 as the normal's (x, y) coordinates.

16.12.6 The Elliptic Reflector. ellipseRays.py generates several vectors leaving the left focus of an ellipse. The reflected vectors are calculated, and the plot shows that they all pass through the ellipse's other focus (see Fig. 16.43).

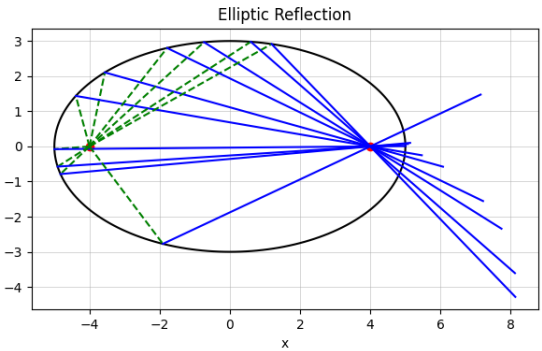


Figure 16.43. Elliptic Reflection

One tricky aspect of this program is obtaining the coordinates where the rays emitted from the left-hand focus intersect the ellipse. The second problem is calculating the normal at an intersection so the reflection can be generated.

16.12.7 Intersecting a Ray with the Ellipse. A ray is parameterized as:

$$x = x_0 + t \cdot d_x, \quad y = y_0 + t \cdot d_y$$

where (x_0, y_0) is the origin of the ray, (d_x, d_y) is its direction, and $t \geq 0$. These equations are substituted into the ellipse equation:

$$(x_0 + t \cdot d_x)^2/a^2 + (y_0 + t \cdot d_y)^2/b^2 = 1$$

This is expanded and rearranged into a quadratic of the form:

$$At^2 + Bt + C = 0$$

where:

$$\begin{aligned} A &= (d_x^2/a^2) + (d_y^2/b^2) \\ B &= 2(x_0 \cdot d_x/a^2 + y_0 \cdot d_y/b^2) \\ C &= (x_0^2/a^2) + (y_0^2/b^2) - 1 \end{aligned}$$

We solve this quadratic and use its positive root to calculate the intersection. Here's the code:

```
def intersectPt(origin, dir):
  x0, y0 = origin.x, origin.y
  dx, dy = dir.x, dir.y

  A = (dx*dx)/(a*a) + (dy*dy)/(b*b)
  B = 2*x0*dx/(a*a) + 2*y0*dy/(b*b)
  C = (x0*x0)/(a*a) + (y0*y0)/(b*b) - 1

  # get positive intersection distance
  t = (-B + math.sqrt(B*B - 4*A*C)) / (2*A)
  return origin + dir*t
```

16.12.8 Calculating a Normal to the Ellipse. For an ellipse centered at the origin:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1,$$

differentiate both sides with respect to x:

$$\frac{2x}{a^2} + \frac{2y}{b^2}\frac{dy}{dx} = 0.$$

Simplify:

$$\frac{dy}{dx} = -\frac{b^2 x}{a^2 y}.$$

This gives the slope of the tangent at any point (x, y). The normal is perpendicular to the tangent, so its slope is the negative reciprocal:

$$m = \frac{a^2 y}{b^2 x}.$$

A vector pointing in the direction of the normal is proportional to

$$\mathbf{n} = (1, m) = (1, \frac{a^2 y}{b^2 x}).$$

Since any scalar multiple of a point leaves the direction unchanged, we can write

$$\mathbf{n} = (\frac{x}{a^2}, \frac{y}{b^2}).$$

This is used to calculate the reflection $\mathbf{r}$ of the ray $\mathbf{d}$:

$$\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$$

Here's the function:

```
def reflectRay(hitPt, d):
  nx = hitPt.x /(a*a)
  ny = hitPt.y /(b*b)
  n = Vec(nx, ny, 0).normalize()
  return d - n * (2 * d.dot(n))
```

16.13 Confocal Conic Sections

Two conic sections are confocal if they share the same focus. Fig. 16.44 shows three confocals 'families' involving parabolas, ellipses, and hyperbolas.

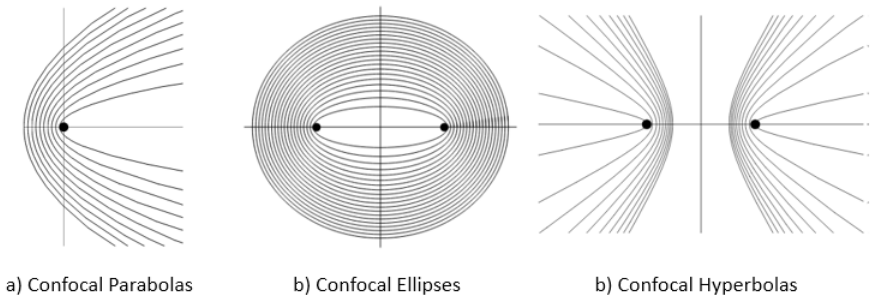


Figure 16.44. Confocal Families

A visually pleasing example is when confocal hyperbolas and ellipses are brought together, forming an orthogonal net of curves (see Fig. 16.45)

It's quite straightforward to prove that these curves always intersect at 90° if we utilize the reflection properties for the ellipse and hyperbola from the last section, combined as in Fig. 16.46 [**Ogi90**].

Referencing the angles marked on Fig. 16.46, we know

$$\angle 1 = \angle 3 \quad \texttt{(ellipse reflection)}$$
$$\angle 2 = \angle 4 \quad \texttt{(hyperbola reflection)}$$

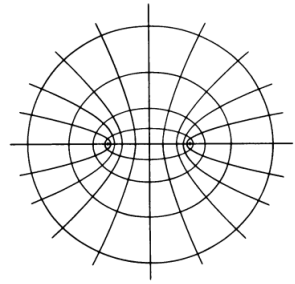


Figure 16.45. Confocal Ellipses and Hyperbolas

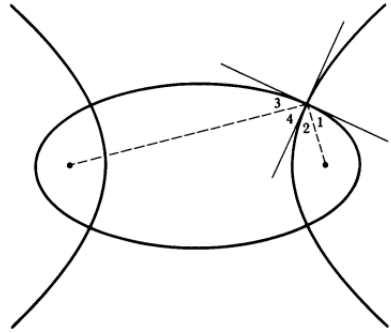


Figure 16.46. Confocal Ellipses and Hyperbolas are Orthogonal

Adding,

$$\angle 1 + \angle 2 = \angle 3 + \angle 4$$

Since we also know that $\angle 1 + \angle 2 + \angle 3 + \angle 4$ is a straight line (the ellipse's tangent), we can conclude that

$$\angle 1 + \angle 2 = \angle 3 + \angle 4 = 90°$$

Another visually interesting example are confocal parabolas opening in opposite directions, as in Fig. 16.47.

It is similarly easy to prove that these curves always intersect at 90° by utilizing the parabola's reflection properties as in Fig. 16.48.

Referencing the angles marked on the figure, the reflection property, applied to parabola α, means that:

$$\angle 1 = \angle 2$$

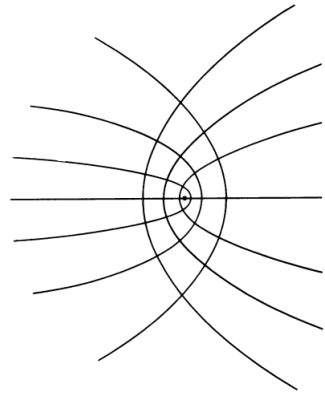


Figure 16.47. Confocal Opposite-facing Parabolas

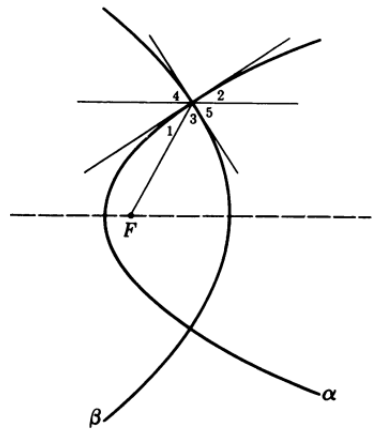


Figure 16.48. Confocal Opposite-facing Parabolas are Orthogonal

and with β as the reflector,

$$\angle 3 = \angle 4 = \angle 5.$$

Adding equals to equals,

$$\angle 1 + \angle 3 = \angle 2 + \angle 5$$

Once again the total of these four angles is a straight line, so each sum is a right angle.

16.13.1 Drawing Confocal Conics. confocalEl.py draws a single ellipse and multiple hyperbolas all with the same foci $(\pm c, 0)$. It's 'obvious' by looking at Fig. 16.49 that all of the hyperbolas intersect with the ellipse at 90°.

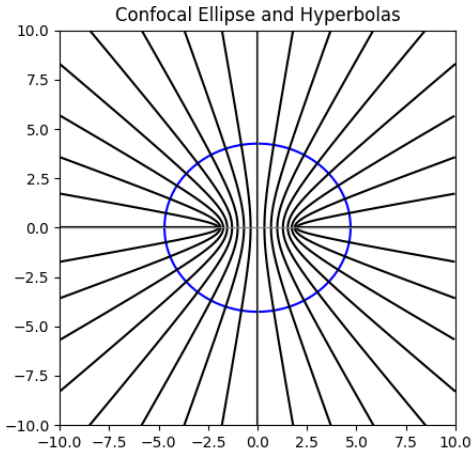


Figure 16.49. Confocal Ellipse and Hyperbolas

`confocalEl.py` utilizes elliptical coordinates ((μ, ν) and (`mu`, `nu`) in the code), which lend themselves to solving problems involving elliptical symmetry (the sums of distances from two foci). Aside from the obvious cases of ellipses and hyperbola, many real-world problems have this property, such as:

- Orbital mechanics;
- Acoustics (ellipse-shaped whispering galleries);
- Antenna radiation with two foci;
- Potential from two charges (dipoles).

(μ, ν) are utilized in the parameterization:

$$x = c \cosh \mu \cos \nu \qquad y = c \sinh \mu \sin \nu$$

where $c > 0$ is the semi-focal distance, $0 \leq \mu < \infty$, and $0 \leq \nu < 2\pi$. Note that μ and ν have different ranges. μ is used in cosh() and sinh(), hyperbolic functions that grow, while ν is employed in cos() and sin() which are periodic.

Curves of constant μ form ellipses with foci at $(\pm c, 0)$:

$$\frac{x^2}{(c \cosh \mu)^2} + \frac{y^2}{(c \sinh \mu)^2} = 1$$

By assuming a fixed $\mu = \mu_0$ in $x = c\cosh\mu\cos\nu$ and $y = c\sinh\mu\sin\nu$, then

$$\frac{x}{c\cosh\mu_0} = \cos\nu, \qquad \frac{y}{c\sinh\mu_0} = \sin\nu.$$

Square and add:

$$\left(\frac{x}{c\cosh\mu_0}\right)^2 + \left(\frac{y}{c\sinh\mu_0}\right)^2 = \cos^2\nu + \sin^2\nu = 1.$$

This is the equation for an ellipse with a semi-major axis $a = c\cosh\mu_0$, and semi-minor axis $b = c\sinh\mu_0$. The focal distance is

$$f = \sqrt{a^2 - b^2} = \sqrt{c^2\cosh^2\mu_0 - c^2\sinh^2\mu_0} = \sqrt{c^2} = c$$

so the foci are at $(\pm c, 0)$.

Curves of constant ν form hyperbolas with the same foci:

$$\frac{x^2}{(c\cos\nu)^2} - \frac{y^2}{(c\sin\nu)^2} = 1$$

By assuming a fixed $\nu = \nu_0$ in $x = c\cosh\mu\cos\nu$ and $y = c\sinh\mu\sin\nu$, then

$$\frac{x}{c\cos\nu_0} = \cosh\mu, \qquad \frac{y}{c\sin\nu_0} = \sinh\mu.$$

Square and subtract:

$$\left(\frac{x}{c\cos\nu_0}\right)^2 - \left(\frac{y}{c\sin\nu_0}\right)^2 = \cosh^2\mu - \sinh^2\mu = 1.$$

This is the equation for a hyperbola with $a = c\cos\nu_0$, and $b = c\sin\nu_0$. The focal distance is:

$$f = \sqrt{a^2 + b^2} = \sqrt{c^2(\cos^2\nu_0 + \sin^2\nu_0)} = c$$

so the foci are again at $(\pm c, 0)$.

The relevant parts of the code:

```
def ellipsePlot(c, mu):  # Fixed mu
  xs = []; ys = []
  for i in range(361):
    nu = 2 * math.pi * i / 360   # vary nu to get coords
    x = c * math.cosh(mu) * math.cos(nu)
    y = c * math.sinh(mu) * math.sin(nu)
    xs.append(x); ys.append(y)
  return xs, ys

def drawHyperbola(c, nu): # Fixed nu
  xs = []; ys = []
  for k in range(1, 800):
    mu = 0.01 + k * 0.01   # vary mu to get coords
    x = c * math.cosh(mu) * math.cos(nu)
    y = c * math.sinh(mu) * math.sin(nu)
```

```
    if abs(x) > 10:  # stop if x goes beyond plot range
      break
    xs.append(x); ys.append(y)
  return xs, ys
```

confocalPara.py draws confocal parabolas with the same focus $(f, 0)$ but different directrixes, $x = d$. When $d < f$, the parabolas open to the right, and with $d > f$ they open to the left. Once again, it's 'obvious' from looking at Fig. 16.50 that the opposite-facing parabolas all intersect at 90°.

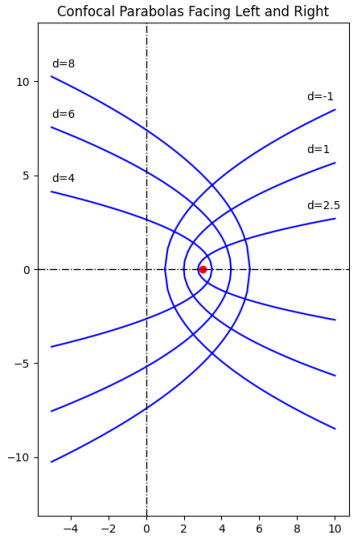


Figure 16.50. Confocal Opposite-facing Parabolas

The parabola's coordinates are generated by a call to `parabolaPlot(xMin, xMax, f, d)` with different values for `d`:

```
f = 3       # focus at (3,0)
ds = [-1, 1, 2.5, 4, 6, 8] # directrix positions

for d in ds:
  xs, ys = parabolaPlot(-5, 10, f, d)
  plt.plot(xs, ys, 'b')
  plt.plot(xs, [-y for y in ys], 'b')
```

The function only returns a list of coordinates for the part of the parabola above the x-axis, which explains the second call to `plt.plot()` to draw the lower half.

The formulation of the parabola in terms of just its focal part $(f, 0)$ and directrix $x = d$ means that any point (x, y) on the parabola satisfies

$$\sqrt{(x-f)^2 + y^2} = |x - d|.$$

Squaring both sides gives

$$(x-f)^2 + y^2 = (x-d)^2.$$

Expanding and simplifying,

$$x^2 - 2fx + f^2 + y^2 = x^2 - 2dx + d^2,$$
$$y^2 = 2(f-d)x + (d^2 - f^2).$$

Let $p = (f-d)/2$ which implies that

$$f + d = 2(p + d), \quad f - p = p + d$$

Also set

$$x_v = f - p = f - \left(\frac{f-d}{2}\right) = \frac{f+d}{2}.$$

x_v is the x-coordinate of the point located at the shortest distance between the focus and directrix.

Then the equation becomes

$$\begin{aligned} y^2 &= 2(f-d)x + (d-f)(d+f) \\ y^2 &= 2 \cdot 2px - 2p \cdot 2(p+d) \\ y^2 &= 4px - 4p(f-p) \\ y^2 &= 4p(x - x_v) \end{aligned}$$

which is the standard form for a parabola opening left or right. This equation style is reflected in `parabolaPlot()`:

```
def parabolaPlot(xMin, xMax, f, d):
  p = (f - d) / 2
  vX = f - p
  if xMax == xMin:
    return [], []
  stepSize = (xMax - xMin) / steps
  xsRange = [xMin + i * stepSize for i in range(steps+1)]
  # Insert vX in sorted order if within range;
  # avoids gaps in the plotted points
  if xMin <= vX <= xMax:
    bisect.insort(xsRange, vX)

  xs =  []; ys = []
  for x in xsRange:
    ySqu = 4 * p * (x - vX)
    if ySqu >= 0:  # only store positive y's
      ys.append(ySqu ** 0.5)
```

```
        xs.append(x)
    return xs, ys
```

One messy aspect is the need to ensure that x_v (vX in the code) is included in the list of plotted coordinates. It may be missed due to the way that the plotting is discretized by stepSize, and because negative y-values are dropped from the coordinates list.

Exercises

(1) The circular waves in Fig. 16.51 were made by touching the surface of a ripple tank, first at A and then at B. As the waves expand, their points of intersection appear to trace a hyperbola. Does they really do that? To find out, we can model the waves with circles centered at A and B.

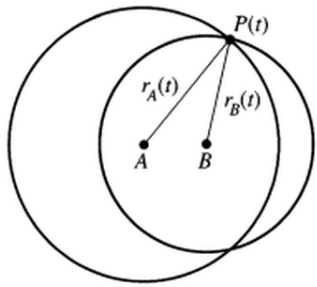


Figure 16.51. Circular Waves

At time t, the point P is $r_A(t)$ units from A and $r_B(t)$ units from B. Since the radii of the circles increase at a constant rate, the speed of the waves is

$$\frac{dr_A}{dt} = \frac{dr_B}{dt}.$$

Work out from this equation that $r_A - r_B$ has a constant value, so that P must lie on a hyperbola with foci at A and B.

(2) *The Trammel of Archimedes.* The mechanical system in Fig. 16.52 consists of a rigid bar of length L, with one end attached to a roller that moves along the y-axis. At a fixed distance R from that end, the bar is attached to a second roller on the x-axis. Let P be the point at the free end of the bar and let θ be the angle the bar makes with the positive x-axis.

a) Find parametric equations for the path of P in terms of θ.

b) Find an equation in x and y whose graph is the path of P, and identify this path.

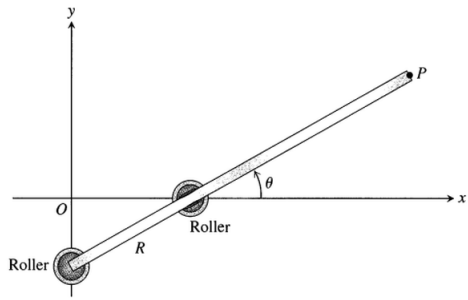


Figure 16.52. The Trammel of Archimedes

(3) *Perihelion and Aphelion.* A planet travels about its sun in an ellipse whose semi-major axis has length a as depicted in Fig. 16.53.

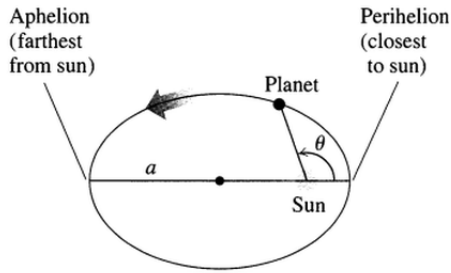


Figure 16.53. A Planet Orbiting a Sun

a) Show that $r = a(1 - e)$ when the planet is closest to the sun and $r = a(1 + e)$ when the planet is farthest away.
b) Use the data in Table 16.2 to determine how close each planet comes to the sun and how far away.

(4) *A satellite orbit.* A satellite passes over the North and South Poles. When it's over the South Pole, it's at the highest point of its orbit, 1000 miles above the surface. Above the North Pole it's at the lowest point, 300 miles.

a) Assuming that the orbit is an ellipse with one focus at the center of the Earth, find its eccentricity. (Take the diameter of the Earth to be 8000 miles.)
b) Using the north-south axis of the Earth as the x-axis and the center of the Earth as the origin, find a polar equation for the orbit.

Planet	Semi-major Axis (AU)	Eccentricity
Mercury	0.3871	0.2056
Venus	0.7233	0.0068
Earth	1.000	0.0167
Mars	1.524	0.0934
Jupiter	5.203	0.0484
Saturn	9.539	0.0543
Uranus	19.18	0.0460
Neptune	30.06	0.0082
Pluto	39.44	0.2481

Table 16.2. Planets Axes and Eccentricities

(5) A comet moves in a parabolic orbit with the sun at the focus. When the comet is 4×10^7 miles from the sun, the line from the comet to the sun makes a 60° angle with the orbit's axis, as shown in Fig. 16.54. How close will the comet come to the sun?

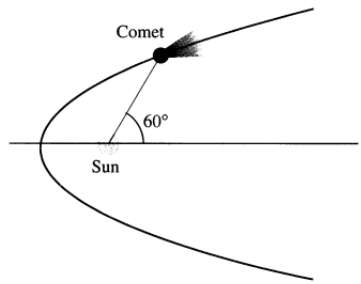


Figure 16.54. A Comet Orbiting the Sun

(6) Perhaps the best mathematical treatment of turtle graphics is *Turtle Geometry: The Computer as a Medium for Exploring Mathematics* by Abelson and Disessa [**AD86**]. The Python version of their simple function for drawing a circle is:

```
def circle(t):
  for _ in range(360):
    t.forward(1)
    t.right(1)
```

while the following functions:

```
def duopoly(t, side1, angle1, side2, angle2):
  c = 0
  for _ in range(200):
    vector(t, c*angle1, side1)
    vector(t, c*angle2, side2)
    c += 1

def vector(t, direction, length):
  t.setheading(direction)
  t.forward(length)
```

are capable of drawing a wide range of patterns, but the specific call:

```
duopoly(t, 4, 2, 8, -2)
```

draws an ellipse. You can try out these functions for yourself in ellipsesTG.py.

a) Represent the (x, y) position of the turtle in terms of parameterized equations involving `length` and `direction`. Then consider how (x, y) changes over time.

b) Use a similar (x, y) parameterization for `duopoly()` and see how the specific call affects (x, y).

Answers

(1)

$$\frac{dr_A}{dt} = \frac{dr_B}{dt} \Rightarrow \frac{d}{dt}(r_A - r_B) = 0 \Rightarrow r_A - r_B = C,$$

So the points $P(t)$ lie on a hyperbola with foci at A and B.

(2) (a) Let C be the point where the vertical line through P meets the x-axis. We have $AB = R$ and $AP = L \Rightarrow PB = L - R$. Then $y = PB\sin\theta \Rightarrow y = (L - R)\sin\theta$, and $BC = PB\cos\theta \Rightarrow BC = (L - R)\cos\theta$. Also, in $\triangle OAB$, $\cos\theta = \frac{OB}{AB} = \frac{OB}{R} \Rightarrow OB = R\cos\theta$. Therefore, $x = OB + BC = R\cos\theta + (L - R)\cos\theta = L\cos\theta \Rightarrow x = L\cos\theta$ and $y = (L - R)\sin\theta$.

(b) Note that

$$\frac{x^2}{L^2} + \frac{y^2}{(L-R)^2} = \frac{L^2\cos^2\theta}{L^2} + \frac{(L-R)^2\sin^2\theta}{(L-R)^2} = 1.$$

Therefore, the points $P(x, y)$ satisfy the equation

$$\frac{x^2}{L^2} + \frac{y^2}{(L-R)^2} = 1,$$

which is an ellipse.

Planet	Perihelion (AU)	Aphelion (AU)
Mercury	0.3075	0.4667
Venus	0.7184	0.7282
Earth	0.9833	1.0167
Mars	1.3817	1.6663
Jupiter	4.9512	5.4548
Saturn	9.0210	10.0570
Uranus	18.2977	20.0623
Neptune	29.8135	30.3065
Pluto	29.6549	49.2251

Table 16.3. Planet Perihelion & Aphelion

(3) (a) Perihelion $= a - ae = a(1-e)$, Aphelion $= ea + a = a(1+e)$

(b) See Table 16.3.

(4) (a) The length of the major axis is 300 miles + 8000 miles + 1000 miles $= 2a$ $\Rightarrow a = 4650$ miles. If the center of the Earth is one focus and the distance from the center to the satellite's low point is 4300 miles (half the diameter plus the distance above the North Pole), then the distance from the center of the ellipse to the focus (center of the Earth) is 4650 miles - 4300 miles = 350 miles $= c$. Therefore

$$e = \frac{c}{a} = \frac{350 \text{ miles}}{4650 \text{ miles}} = \frac{7}{93}.$$

(b) $r = \frac{a(1-e^2)}{1+e\cos\theta} \Rightarrow r = \frac{4650[1-(7/93)^2]}{(1+(7/93)\cos\theta)} = \frac{430000}{93+7\cos\theta}$ miles.

(5) If the vertex is $(0,0)$, then the focus is $(p,0)$. Let $P(x,y)$ be the present position of the comet. Then

$$\sqrt{(x-p)^2+y^2} = 4\times 10^7.$$

Since $y^2 = 4px$ we have

$$\sqrt{(x-p)^2+4px} = 4\times 10^7 \Rightarrow (x-p)^2+4px = 16\times 10^{14}.$$

Also, $x - p = 4\times 10^7 \cos 60^\circ = 2\times 10^7 \Rightarrow x = p + 2\times 10^7$. Therefore

$$\begin{aligned}
(2\times 10^7)^2 + 4p(p+2\times 10^7) &= 16\times 10^{14}\\
4\times 10^{14} + 4p^2 + 8p\times 10^7 &= 16\times 10^{14}\\
4p^2 + 8p\times 10^7 - 12\times 10^{14} &= 0\\
p^2 + 2p\times 10^7 - 3\times 10^{14} &= 0\\
(p+3\times 10^7)(p-10^7) &= 0\\
p = -3\times 10^7 \quad &\text{or} \quad p = 10^7.
\end{aligned}$$

Since p is positive we obtain $p = 10^7$ miles.

(6) (a) The circle's parametric form

$$x(t) = r\cos t, \quad y(t) = r\sin t, \quad t \in [0, 2\pi]$$

In this case, $r = 1$ and $t = 1°$.

Over 360 iterations it will create a circle with a circumference of 360 'turtle' units. Thus the circle's radius will be $360/2\pi \approx 57$ units.

(b) The parametric form for an ellipse:

$$x(t) = a\cos(t), \quad y(t) = b\sin(t)$$

When `duopoly(4, 2, 8, -2)` is called, the turtle draws two vectors:

- vector($2c$, 4): heading $2c$, length 4
- vector($-2c$, 8): heading $-2c$, length 8

In Cartesian coordinates, the position after multiple iterations is:

$$x(c) \approx 4 \cdot \cos(2c) + 8 \cdot \cos(-2c)$$
$$y(c) \approx 4 \cdot \sin(2c) + 8 \cdot \sin(-2c)$$

Since $\cos(-\theta) = \cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$:

$$\begin{aligned} x(c) &= 4 \cdot cos(2c) + 8 \cdot cos(2c) = 12 \cdot cos(2c) \\ y(c) &= 4 \cdot sin(2c) - 8 \cdot sin(2c) = -4 \cdot sin(2c) \end{aligned}$$

This is an ellipse with parameter $t = 2c$:

$$x = 12 \cdot \cos(t), \quad y = -4 \cdot \sin(t)$$

This gives us an ellipse with:

- Semi-major axis $a = 12$ (horizontal)
- Semi-minor axis $b = 4$ (vertical)

In summary: when angle1 = -angle2, the two vectors create a sum and difference pattern: the sum ($4+8 = 12$) determines one axis, while the absolute difference ($|4 - 8| = 4$) determines the other.

17

Solving a Cubic Equation

Why Study and Solve Cubic Equations?

- **They model systems with non-linear behaviour**. Cubic relationships arise in many problems, including beam bending, fluid flow, population models, and optimisation. Being able to solve a cubic gives us information about their turning points, thresholds, and stability boundaries.
- **They're more complicated than quadratics**. Quadratics can always be solved with a simple formula; cubics introduce new algebra such as multiple real roots, cusps, and the need for more complex solution methods.
- **Their solution led to major advances in algebra**. Cardano's solution of the cubic in the 1500s drove the development of complex numbers, symbolic manipulation, and the idea of radicals. Cubics are tied to the birth of modern algebra.

17.1 Introduction

The discovery of a general solution to cubic equations can be traced back to the Italian mathematicians Scipione del Ferro (1465–1526), Niccolò Tartaglia (1500–1557), and Gerolamo Cardano (1501–1576). Cardano's publication of Tartaglia's solution in *Ars Magna* in 1545 laid the groundwork for solving higher-degree polynomial, but his approach was geometrical, involving literal cubes and their volumes, which was standard for a time when algebra was in its infancy [**Dun90**]. However, we'll present his ideas algebraically, which clarifies many aspects of his approach, such as his reliance on square roots of negative numbers (i.e. on

complex numbers). For instance, consider the *depressed* cubic $x^3 - 15x - 4 = 0$ (depressed because there's no x^2 term, not because of any emotional malaise). Applying Cardano's formula will lead to:

$$x = \sqrt[3]{2 + \sqrt{-121}} - \sqrt[3]{-2 + \sqrt{-121}}$$

In an historical period when even negative numbers were still viewed suspiciously, utilizing negative square roots were inconceivable, and there was a strong temptation to dismiss equations of this type as unsolvable. Yet it can easily be checked that the equation has a real solution at $x = 4$.

Cardano took a few half-hearted stabs at investigating this problem, but ultimately dismissed the whole enterprise as being "as subtle as it is useless."

It would be another generation before Rafael Bombelli (1526-1573), in his 1572 treatise *Algebra*, took the bold step of regarding complex numbers as necessary. But their full importance didn't really become evident until the work of Euler, Gauss, and Cauchy more than two centuries later. In the meantime, François Viète's (1540-1603) method emerged as a pleasing alternative to solving cubic equations, offering a solution when all three roots were real which used only 'respectable' trigonometry.

We'll implement Cardano's and Viète's work with the help of Python's complex numbers and its `cmath` module. If you need a quick refresher on those topics, please cast an eye over Appendix J. We'll then change track and consider *visual* ways of finding the roots of cubic equations. We'll look at three approaches: ordinary 2D plots, modulus surfaces, and phase maps with domain coloring and contours, all of which are easy to implement using matplotlib.

17.2 Depressing Cubic Equations

To solve $f(x) = ax^3 + bx^2 + cx + d = 0$, we start by rewriting it as a depressed cubic of the form

$$x^3 + px = q.$$

Substitute $y - h$ for x in the full cubic:

$$\begin{aligned} ax^3 + bx^2 + cx + d &= a(y-h)^3 + b(y-h)^2 + c(y-h) + d \\ &= a(y^3 - 3hy^2 + 3h^2y - h^3) + b(y^2 - 2hy + h^2) + c(y-h) + d \\ &= ay^3 - 3ahy^2 + 3ah^2y - ah^3 + by^2 - 2bhy + bh^2 + cy - ch + d \\ &= y^3 + y^2(-3ah + b)/a + y(3ah^2 - 2bh + c)/a + \\ &(-ah^3 + bh^2 - ch + d)/a \end{aligned}$$

If we set $h = \frac{b}{3a}$, then the y^2 term disappears, and the equation takes on the desired form $x^3 + px = q$, where:

$$\begin{aligned} p &= (3ac - b^2)/(3a^2) \\ q &= -(2b^3 - 9abc + 27a^2d)/(27a^3) \end{aligned}$$

The importance of this manipulation is that it shows that any general cubic can be rewritten in depressed form. In addition, p and q can be used to calculate the cubic's *discriminant* Δ:

$$\Delta = \left(\frac{q}{2}\right)^2 + \left(\frac{p}{3}\right)^3$$

Δ provides information about the nature of the cubic's roots:

- If $\Delta > 0$ then the equation has one real root and two complex conjugate roots.
- If $\Delta = 0$ then all the roots are real, and at least two of them are equal.
- If $\Delta < 0$ then the equation has three distinct real roots.

Later on we'll employ the discriminant to decide whether to use Cardano's or Viète's method for solving a cubic.

17.3 The Tartaglia-Cardano's Method

Consider the depressed cubic: $x^3 + px = q$. The key idea is to rewrite x as a difference of two quantities, $x = t - u$, and then find t and u.

By the binomial theorem, we have

$$(t-u)^3 = t^3 - 3t^2u + 3tu^2 - u^3.$$

Thus

$$\begin{aligned} t^3 - u^3 &= (t-u)^3 + 3t^2u - 3tu^2 \\ &= (t-u)^3 + 3tu(t-u). \end{aligned}$$

The equation becomes $x^3 + px = q$ if we set

$$p = 3tu, \qquad q = t^3 - u^3.$$

The first equation gives

$$u = \frac{p}{3t}.$$

Substitute this into the second equation to obtain

$$q = t^3 - \frac{p^3}{27t^3}.$$

Rearrange and multiply through by t^3 to get

$$t^6 - qt^3 - \frac{p^3}{27} = 0.$$

We can apply the standard quadratic formula by assuming that we're solving for t^3:

$$t^3 = \frac{q \pm \sqrt{q^2 + 4p^3/27}}{2} = \frac{q}{2} \pm \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}.$$

Therefore,

$$t = \sqrt[3]{\frac{q}{2} \pm \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}}.$$

$u^3 = t^3 - q$, so

$$u = \sqrt[3]{-\frac{q}{2} \pm \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}}.$$

Also, $x = t - u$, so

$$x = \sqrt[3]{\frac{q}{2} \pm \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}} - \sqrt[3]{-\frac{q}{2} \pm \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}}.$$

The square root term is the discriminant, which we can refer to directly:

$$x = \sqrt[3]{\frac{q}{2} \pm \sqrt{\Delta}} - \sqrt[3]{-\frac{q}{2} \pm \sqrt{\Delta}}.$$

There are two $\pm$ signs in the expression, which may make you think that we've found four solutions, but two of them are the same. Consider the expression to have the form $x = \sqrt[3]{a \pm b} - \sqrt[3]{-a \pm b}$. The solution with two minus signs inside the radicals equals the solution with two positive signs, because of the effects of factoring:

$$\sqrt[3]{a - b} - \sqrt[3]{-a - b} = (-1)\sqrt[3]{-a + b} - (-1)\sqrt[3]{a + b} = \sqrt[3]{a + b} - \sqrt[3]{-a + b}.$$

17.3.1 Cardano's Example. We'll solve $x^3 + 6x = 20$, which was one of Cardano's examples in *Ars Magna*. $p = 6$ and $q = 20$, so that

$$\Delta = (q/2)^2 + (p/3)^3 = 10^2 + 2^3 = 108$$

This means that there's one real root and two complex conjugate roots.

$$\begin{aligned} x &= \sqrt[3]{\tfrac{q}{2} \pm \sqrt{\Delta}} - \sqrt[3]{-\tfrac{q}{2} \pm \sqrt{\Delta}} \\ &= \sqrt[3]{10 \pm \sqrt{108}} - \sqrt[3]{-10 \pm \sqrt{108}} \end{aligned}$$

Cardano ignored any choice of operations that might lead to negative cube roots, which only leaves

$$x = \sqrt[3]{10 + \sqrt{108}} - \sqrt[3]{-10 + \sqrt{108}}$$

This simplifies to $x = 2$, as we can (almost) confirm with Python:

```
>>> import math
>>> d = math.sqrt(108)
>>> a = (10 + d)**(1/3)
>>> b = (-10 + d)**(1/3)
>>> a-b
1.9999999999999996
```

17.3.2 Finding the Other Roots. Once we've used Cardano's method to find a real root, the other two can be found by factorization. For example, we'd factorize $f(x) = x^3 + 6x - 20$ by dividing by $(x - 2)$, producing

$$f(x) = (x-2)(x^2+2x+10) = (x-2)(x-(-1+3i))(x-(-1-3i)).$$

We can solve $x^2+2x+10$ with the quadratic formula, and choose to give up upon discovering that $b^2 - 4ac < 0$ (where $a = 1, b = 2, c = 10$).

Factoring a cubic into a linear term and a quadratic is quite straightforward. If you divide $a_3x^3 + a_2x^2 + a_1x + a_0$ by $b_1x + b_0$ then the coefficients of $q_2x^2 + q_1x + q_0$ satisfy:

$$\begin{aligned} q_2 &= \frac{a_3}{b_1}, \\ q_1 &= \frac{a_2 - q_2 b_0}{b_1}, \\ q_0 &= \frac{a_1 - q_1 b_0}{b_1}, \\ R &= a_0 - q_0 b_0. \end{aligned}$$

where R is a remainder which we expect to be 0 when we factor by a linear equation using a root. These equalities are obtained by multiplying the linear and quadratic terms, and matching their results with the corresponding coefficients in the cubic:

$$(b_1x + b_0) \cdot (q_2x^2 + q_1x + q_0) = a_3x^3 + a_2x^2 + a_1x + a_0$$

Therefore:

$$\begin{aligned} a_3 &= b_1q_2, \\ a_2 &= b_1q_1 + b_0q_2, \\ a_1 &= b_1q_o + b_0q_1, \\ a_0 &= b_0q_0 + R. \end{aligned}$$

`divideCubicByLinear()` in cubicUtils.py is a translation of this into Python. Let's use it on Cardano's cubic:

```
>>> from cubicUtils import *
>>> divideCubicByLinear([1, 0, 6, -20],[1,-2])
([1.0, 2.0, 10.0], 0.0)
```

This is the quadratic $x^2 + 2x + 10$.

cubicUtils.py also contains a quadratic solver, which utilizes `cmath.sqrt()`. We can supply it with the coefficients of the quadratic returned by `divideCubicByLinear()` to get the complex conjugates:

```
>>> from cubicUtils import *
>>> quadraticSolve(1, 2, 10)
[(-1+3j), (-1-3j)]
```

17.3.3 Bombelli's Example. Now we'll consider $x^3-15x = 4$, one of Bombelli's examples, and be brave enough not be put off when we encounter $\sqrt{-1}$. $p = -15$ and $q = 4$, so that

$$\Delta = (q/2)^2 + (p/3)^3 = 2^2 + (-5)^3 = -121$$

This means that there are three distinct roots, but doesn't mean that we'll avoid complex numbers.

$$\begin{aligned} x &= \sqrt[3]{\tfrac{q}{2} \pm \sqrt{\Delta}} - \sqrt[3]{-\tfrac{q}{2} \pm \sqrt{\Delta}} \\ &= \sqrt[3]{2 \pm 11i} - \sqrt[3]{-2 \pm 11i} \end{aligned}$$

We'd dearly like the cube roots to be positive, so limit ourselves to:

$$\begin{aligned} x &= \sqrt[3]{2 + 11i} - \sqrt[3]{-2 + 11i} \\ &= \sqrt[3]{2 + 11i} + \sqrt[3]{2 - 11i} \\ &= (2 + i) + (2 - i) \\ &= 4 \end{aligned}$$

A real solution via the realm of complex numbers.

The calculation can be checked using Python:

```
>>> (2 + 11j) ** (1/3)
(2+1.0000000000000002j)
>>> (2 - 11j) ** (1/3)
(2-1.0000000000000002j)
```

or we can start at the solution and find integer powers:

```
>>> (2 + 1j) ** 3
(2+11j)
>>> (2 - 1j) ** 3
(2-11j)
```

Taking the cube root of a complex number can lead to some confusion, since only one result is given when using the ** operator. For instance if we take the cube root of $(-2+11i)$, which was the original form of the second root in the x equation above, then Python reports:

```
>>> (-2 + 11j) ** (1/3)
(1.8660254037844388+1.2320508075688772j)
```

It's necessary to obtain all three of the roots and select the most suitable one:

```
>>> import complexUtils
>>> rs = complexUtils.getRoots((-2 + 11j), 3)
>>> for r in rs: print(r)
(1.8660254037844388+1.2320508075688772j)
(-2.0000000000000004+0.9999999999999996j)
(0.1339745962155615-2.232050807568877j)
```

The second root, $(-2 + i)$, is the one we need.

17.4 Viète's Method

Viète's trigonometric method solves a cubic equation when all of its roots are real, which we can detect by checking that its discriminant Δ is negative. It utilizes the identity that relates the cosine of a triple angle to a cubic expression:

$$\cos(3\theta) = 4\cos^3(\theta) - 3\cos(\theta)$$

In the depressed cubic $x^3 + px = q$, assume that $x = k\cos\theta$ and $k^2 = -4p/3$ (making $x = 2\sqrt{(-p/3)}\cos\theta$). Substitute this into $x^3 + px = q$:

$$\begin{aligned}
(-8p/3)\sqrt{(-p/3)}\cos^3\theta + 2p\sqrt{(-p/3)}\cos\theta &= q \\
(4\cos^3\theta - 3\cos\theta)\left((-2p/3)\sqrt{(-p/3)}\right) &= q \\
\cos 3\theta &= q/((-2p/3)\sqrt{(-p/3)}) \\
&= \frac{q}{2\sqrt{(-p/3)^3}}
\end{aligned}$$

For this equation to have real solutions,

$$\begin{aligned}
\left|q/((-2p/3)\sqrt{(-p/3)})\right| &\leq 1 \\
q^2/((4p^2/9)(-p/3)) &\leq 1 \quad \text{(when } p < 0\text{)} \\
q^2/(-4p^3/27) &\leq 1 \\
-p^3/27 &\geq q^2/4 \\
p^3/27 + q^2/4 &\leq 0.
\end{aligned}$$

This means that when $p^3/27 + q^2/4 \leq 0$ (or equivalently when $\Delta \leq 0$), then we can indeed find real roots for $x^3 + px = q$ with $x = 2\sqrt{(-p/3)}\cos\theta$. Returning to the cosine,

$$\cos 3\theta = \frac{q}{2\sqrt{(-p/3)^3}},$$

the three angles corresponding to the roots are:

$$\theta_k = \frac{1}{3}\operatorname{acos}\left(\frac{q}{2\sqrt{(-p/3)^3}}\right) + \frac{2\pi k}{3}, \quad k = 0, 1, 2.$$

Substitute each of these into x to obtain

$$x_k = 2\sqrt{(-p/3)}\cos\theta_k.$$

This gives us three real roots without needing to rely on complex numbers.

17.4.1 Why $k^2 = -4p/3$? What's the reason for choosing $k^2 = -4p/3$ at the start of Viète's method? He began with $x^3 + px = q$, and assigned $x = k\cos\theta$, to produce:

$$k^3\cos^3\theta + pk\cos\theta = q$$

Factoring out a k^3 term gives:

$$k^3(\cos^3\theta + \frac{p}{k^2}\cos\theta) = q$$

Rearrange the identity $\cos 3\theta = 4\cos^3\theta - 3\cos\theta$ as:

$$\cos^3\theta = \frac{1}{4}(\cos 3\theta + 3\cos\theta)$$

and use its right-hand side to simplify the conic:

$$k^3\left[\frac{1}{4}(\cos 3\theta + 3\cos\theta)\right] + pk\cos\theta = q$$

or

$$\frac{k^3}{4}\cos 3\theta + \left(\frac{3k^3}{4} + pk\right)\cos\theta = q$$

Viète wanted this equation to reduce to a simple linear relation, ideally eliminating the $\cos\theta$ term completely. To do that, he set:

$$\frac{3k^3}{4} + pk = 0$$

which gives:

$$k^2 = -\frac{4p}{3}$$

17.4.2 Using the Method. We'll consider $x^3 - 2x = -1$, which makes $p = -3$ and $q = -1$. First we check that the discriminant Δ^2 is ≤ 0, making the equation suitable for Viète's method:

$$(p/3)^3 + (q/2)^2 = -1 + 0.25 = -0.75 \leq 0$$

The three angles for the roots:

$$\begin{aligned}\theta_k &= \tfrac{1}{3}\,\text{acos}(\tfrac{q}{2\sqrt{(-p/3)^3}}) + \tfrac{2\pi k}{3}, \quad k = 0, 1, 2.\\ x_k &= 2\sqrt{(-p/3)}\cos\theta_k\end{aligned}$$

Focusing on θ_0 and x_0:

$$\begin{aligned}\theta_0 &= \frac{1}{3}\operatorname{acos}\left(\frac{-1}{2\sqrt{(-3/3)^3}}\right)\\ &= \frac{1}{3}\operatorname{acos}\frac{-1}{2}\\ x_0 &= 2\sqrt{(3/3)}\cos\left(\frac{1}{3}\operatorname{acos}\frac{-1}{2}\right)\\ &= 2\cos\left(\frac{1}{3}\operatorname{acos}\frac{-1}{2}\right)\end{aligned}$$

Switching to Python:

```
>>> import math
>>> 2 * math.cos ( math.acos(-0.5)/3 )
1.532088886237956
```

The other two values are obtained by adding $2\pi/3$ and $4\pi/3$ (or equivalently $-2\pi/3$) to θ_0:

```
>>> theta0 = math.acos(-0.5)/3
>>> 2 * math.cos( theta0 + (2*math.pi)/3 )
-1.8793852415718166
>>> 2 * math.cos( theta0 - (2*math.pi)/3 )
0.3472963553338613
```

In summary, the three roots (to 4 d.p) are 1.5321, −1.8794, and 0.3473. These can be tested by plugging them back into the original cubic:

```
>>> from cubicUtils import cubic
>>> cubic(1.532088886237956, 1, 0, -3, 1)   # x^3 -3x + 1 = 0
-4.440892098500626e-16
>>> cubic(-1.8793852415718166, 1, 0, -3, 1)
8.881784197001252e-16
>>> cubic(0.3472963553338613, 1, 0, -3, 1)
-1.5543122344752192e-15
```

17.5 Implementing Cardano's and Viète's Methods

The function `cvRoots()` in cubicRoots.py uses either Cardano's or Viète's method to find the roots of a cubic depending on the discriminant of its depressed version.

The following runs of `findCubicRoots()` correspond to the earlier examples:

```
>>> from cubicRoots import *
>>> findCubicRoots((1, 0, 6, -20))
Equation: x^3 + 6x - 20 = 0
Discriminant: 108.0000
One real root and two complex conjugate roots
Roots:
```

```
   2.0000
  -1.0000+3.0000j
  -1.0000-3.0000j

>>> findCubicRoots((1, 0, -15, -4))
Equation: x^3 - 15x - 4 = 0
Discriminant: -121.0000
All three roots are real and distinct
Roots:
   4.0000
  -3.7321
  -0.2679

>>> findCubicRoots((1, 0, -3, 1))
Equation: x^3 - 3x + 1 = 0
Discriminant: -0.7500
All three roots are real and distinct
Roots:
   1.5321
  -1.8794
   0.3473
```

`findCubicRoots()` calls `cvRoots()` to find the roots, and various extra features for printing the discriminant and the roots. The essential structure of `cvRoots()` is shown below:

```
def cvRoots(a, b, c, d):
  if a == 0:
    raise ValueError("Coef 'a' must not be zero")

  # Convert to depressed cubic: x = t - b/(3a)
  p = (3*a*c - b*b)/(3*a*a)
  q = (2*b**3 - 9*a*b*c + 27*a*a*d)/(27*a**3)
  disc = (q/2)**2 + (p/3)**3
  shift = -b/(3*a)

  roots = []
  if abs(disc) < 1e-12:  # disc == 0
    # repeated real roots
    #  :
    roots = ...
  elif disc > 0:
    # Cardano's method
    # One real root and two complex conjugates
    #   :
    roots = ...
  else:  # disc < 0
    # Vieta's method
```

```
    # Three distinct real roots
    #   :
    roots = ...

  return disc, roots
```

p and q are the coefficients of the depressed cubic $x^3 + px = q$, which allows the discriminant (`disc`) to be calculated and used to direct the execution to one of three branches.

17.5.1 Cardano's Method in `cvRoots()`. The code branch that implements Cardano's method:

```
    sqrtD = math.sqrt(disc)
    # t = (q/2 + sqrtD)**(1/3)
    t = complexUtils.getRealRoot((q/2 + sqrtD), 3)
    u = (-q/2 + sqrtD)**(1/3)

    # Ensure t*u = -p/3 relationship;
    # avoids using the wrong cube root
    if t != 0 and isinstance(u, complex):
      vPrev = u
      u = -p/(3*t)

    if a < 0:
      u = -u   # so x1, x2, x3 are additions
    x1 = t - u
    # compute the other two roots using the
    # principal cube root of unity w = e^(2*pi*i/3)
    w = cmath.exp(2*math.pi*1j/3)
    x2 = w*t - w*w*u
    x3 = w*w*t - w*u
    roots = [complex(x1.real+shift, 0), x2+shift, x3+shift]
```

The calculation of t and u are almost as expected, but matters are complicated by having to ensure that the real cube root is returned for t and u. For t this is guaranteed by using `complexUtils.getRoots()` to get all three roots, and then selecting the real one. The correct value for u is determined by making sure that the $tu = -p/3$ relationship is true.

Another adjustment is to check the sign of a to decide if x should be calculated using an addition of t and u.

The last few lines are not part of Cardano's method since they utilize the cube roots of unity to 'rotate' the t and u values to obtain the other two x roots.

17.5.2 Viète's Method in `cvRoots()`. The code branch that implements Viète's method:

```
arg = q/(2*math.sqrt(-(p/3)**3))
theta = math.acos(arg)/3
m = 2*math.sqrt(-p/3)
roots = [ m*math.cos(theta) + shift,
          m*math.cos(theta + 2*math.pi/3) + shift,
          m*math.cos(theta - 2*math.pi/3) + shift ]
```

This is almost unchanged from the maths.

17.6 Visually Finding Roots in a 2D Plot

In the last section we found the roots of $x^3 + 6x - 20 = 0$. One way to check the results is to plot the cubic using `plot2D.py`. For example:

```
> python plot2D.py
Enter cubic coefs (a b c d): 1 0 6 -20
Range (or 3): 4
Range: [-4, 4]
```

This displays the graph in Fig. 17.1.

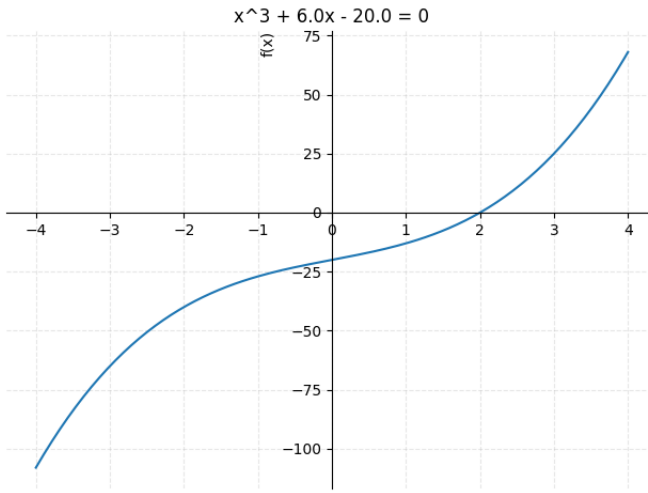


Figure 17.1. A 2D Plot of a Cubic

Fig. 17.1 illustrates the main drawback of visualizing cubics in 2D – only the real roots (in this case $x = 2$) are shown. However, there are ways to locate the complex conjugate pair $(-1 \pm 3i)$ once the real root has been identified. Let the roots of $y = px^3 + qx^2 + rx + s = 0$ be $x = c$ and $x = a \pm ib$, where a, b, c are real numbers. The factorized cubic is

$$y = (x - c)(x^2 - [a + ib])(x - [a - ib]) = 0,$$

or

$$y = (x - c)(x^2 - 2ax + a^2 + b^2) = 0.$$

Our aim is to find a and b by employing the geometric properties shown in Fig. 17.2.

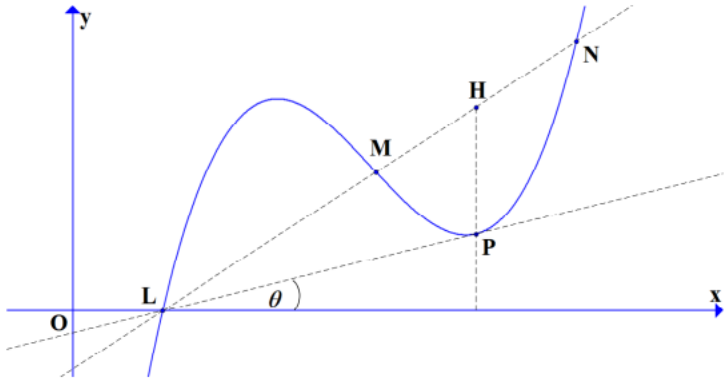


Figure 17.2. Find the Complex Conjugate Roots

We draw a line LMN through the cubic, where L is the real root at $x = c$. The general equation of a line is given by $y - y_1 = m(x - x_1)$, where m is the slope. So for LMN we have $y = m(x - c)$. When it intersects the cubic at M and N we get

$$m(x - c) = (x - c)(x^2 - 2ax + a^2 + b^2),$$

which simplifies to

$$x^2 - 2ax + a^2 + b^2 - m = 0.$$

Solving this gives us

$$x = a \pm \sqrt{m - b^2}.$$

M and N will have the same x value when their x equations are equal, implying that $m = b^2$, and that $x = a$. The line becomes LP, a tangent to the cubic, with LP = the slope = $m = b^2 = \tan\theta$. The real parts of the complex roots $x = a \pm ib$ can now be located:

- a is the x coordinate of the tangent point P. This can be read off by drawing a vertical line from P down to the x-axis;
- $b = \sqrt{\tan\theta}$, where θ is measured from the graph, or alternatively $b = \sqrt{|m|}$ since we're only interested in the magnitude of the slope.

As an example, let's find the roots for $x^3 - 3x^2 + x + 5$. Of course, we can just call `findCubicRoots()`:

```
>>> from cubicRoots import *
>>> findCubicRoots((1, -3, 1, 5))
Equation: x^3 - 3x^2 + x + 5 = 0
Discriminant: 3.7037
One real root and two complex conjugate roots
Roots:
  -1.0000
   2.0000+1.0000j
   2.0000-1.0000j
```

However, let's use the geometric approach on Fig. 17.3, which uses a graph drawn by `plot2D.py`:

```
> python plot2D.py
Enter cubic coefs (a b c d): 1 -3 1 5
Range (or 3):
Range: [-3, 3]
```

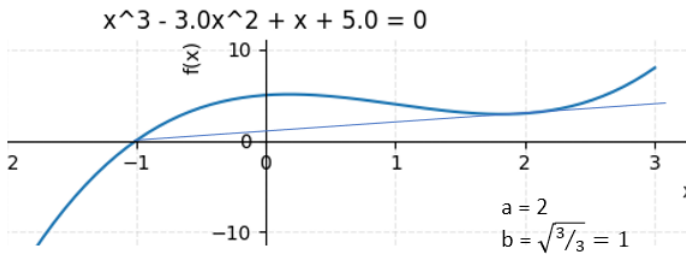


Figure 17.3. Find the Complex Roots for $x^3 - 3x^2 + x + 5$

To be clear, `plot2D.py` only draws the curve; we manually drew the tangent line and calculated a and b in Fig. 17.3. The graph suggests that $a = 2$ and $b \approx 1$, which is in agreement with the actual values for the complex roots.

17.7 Visually Finding Roots with a Modulus Surface

Long and Hern [**LH89**] popularized the use of a *modulus surface* to find the roots of complex functions. This approach can easily be applied to a cubic function f by supplying it with complex numbers rather than reals. Each complex result is converted to a modulus, $|f(z)|$ so the plot can be 3D – the $(x + iy)$ inputs occupy the xy plane and the moduli are plotted up the z-axis. The roots occur when the modulus surface touches the plane.

modulusPlot.py implements this using matplotlib and a small use of numpy so the x and y values can be converted into a mesh. Fig. 17.4 shows the graph generated for $x^3 - 3x^2 + x + 5$:

```
> python modulusPlot.py
Enter cubic coefs (a b c d): 1 -3 1 5
Range (or 3):
Real range: [-3, 3]
Imaginary range: [-3, 3]
```

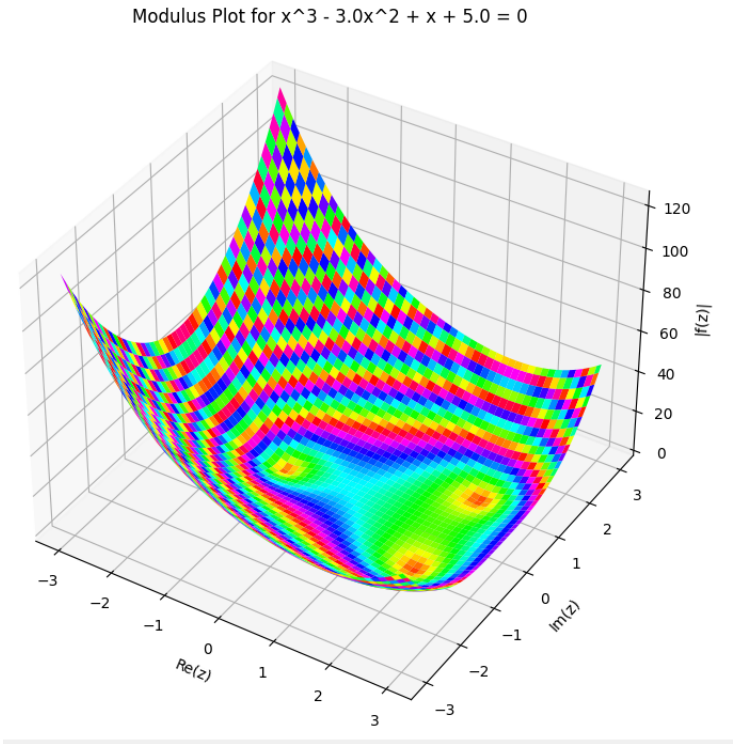


Figure 17.4. Modulus Plot for $x^3 - 3x^2 + x + 5$

The hsv colormap used by `matplotlib.plot_surface()` has been made to cycle through all its colors in steps of 10, starting from $z = 0$. This makes it much easier to see the location of the three roots (the red spots at $(-1, 0)$, and $(2 \pm i)$). The modulus data is generated by:

```
def getMagnitudes(realVs, imagVs):
  mags = []
  for im in imagVs:
```

```
    rowMag = []
    for re in realVs:
      z = complex(re, im)
      try:
        w = cubicUtils.cubic(z, a, b, c, d)
      except (ValueError, ZeroDivisionError):
        w = complex(float('nan'), float('nan'))
      rowMag.append( abs(w))
    mags.append(rowMag)
  return mags
```

a, b, c, and d are the coefficients supplied by the user.

Fig. 17.4 also hints at the weakness of this approach: we needed to rotate the graph so that the roots became easier to see, and 'view improvement' may not always be possible, especially if the surface is very mountainous. The other issue, common to all of these visualizations, is that the roots can only be estimated approximately.

17.8 Visually Finding Roots with Enhanced Phase Maps

By 'enhanced' we mean phase maps with domain coloring and contours, an approach that's explained in Appendix J on complex numbers. Nevertheless, we'll briefly describe the meaning of its color scheme here as well. The cubic is displayed as a phase map by calling phasePlot.py:

```
> python phasePlot.py
Enter cubic coefs (a b c d): 1 -3 1 5
Range (or 3):
Real range: [-3, 3]
Imaginary range: [-3, 3]
```

with the graph in Fig. 17.5 being drawn.

Positive numbers are colored red, negative numbers are colored blue. This is easiest to understand by looking along the real axis from right to left which shows values switching from red to light blue at $(-1, 0)$ which correspond to the way that the real part of the curve behaves in Fig. 17.3.

Zeros and poles occur at the points where all the colors meet. A zero is distinguished by the change from red, yellow, green, to blue in a counter-clockwise order around the point. A pole, which is when the output extends off to infinity, has its colors change in a clockwise order. This means that there are three zeros in Fig. 17.5, at $(-1, 0)$ and $(2 \pm i)$, as expected.

There are two types of contour line drawn over the surface – the black lines are contours of equal phase, and are labeled in degrees. The white lines are contours of equal magnitude (modulus), which pleasingly match the shape of the curve in Fig. 17.4.

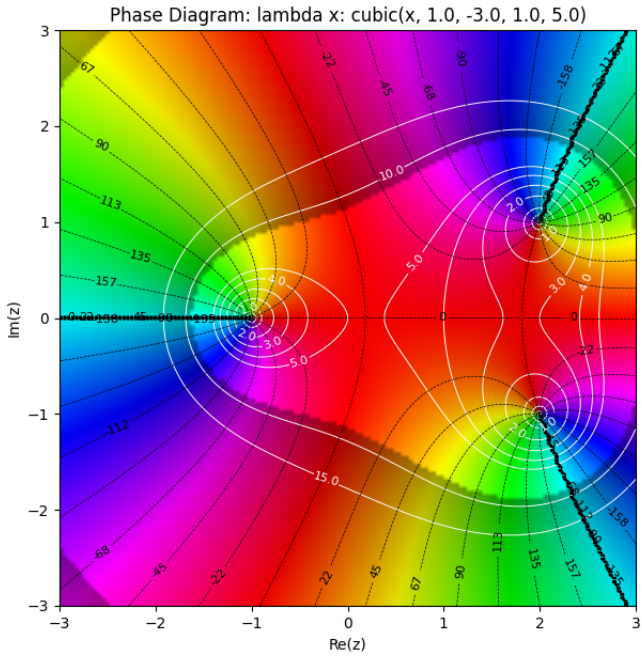


Figure 17.5. Phase Plot for $x^3 - 3x^2 + x + 5$

phasePlot.py is actually very simple, little more than a wrapper around a call to `showPhase()` in complexUtils.py.

18

The Catenary

Why Study the Catenary?

- **Structural Engineering and Architecture**. The catenary is the optimal shape for a hanging cable and, when inverted, forms an arch that carries loads purely in compression, making it strong. Suspension bridge cables, power lines, and railway overhead wires naturally form catenaries, and the catenary is an ideal road surface for bicycles with square wheels.

- **Historical and Mathematical Significance**. The catenary played a key role in the development of early calculus and provides a natural geometric example of hyperbolic functions via $y = a\cosh(x/a)$. The catenary has unusual connections to other important curves such as the parabola and the tractrix.

- **Differential Geometry and Curvature**. Rotating a catenary produces the *catenoid*, the simplest nontrivial minimal surface, connecting the curve to curvature theory. Soap-film catenoids are used to study stability transitions (where a catenoid "snaps" into two disks), and the resulting mathematics can be utilized for investigating surface tension, Plateau's laws, and capillary action.

18.1 The Catenary Chain

The catenary is the shape of a perfectly flexible hanging chain, with its mass uniformly distributed along its length. It looks very much like a parabola, and was thought to be one by Galileo, although some authors argue that he only believed

it was 'approximately' parabolic. In any case, that belief was disproved by the 17-year-old Huygens in 1646, who showed that the parabola was the shape assumed by a flexible cord loaded by weights uniformly distributed in the horizontal direction. Fig. 18.1 demonstrates the difference between the two, and the curves are admittedly very similar.

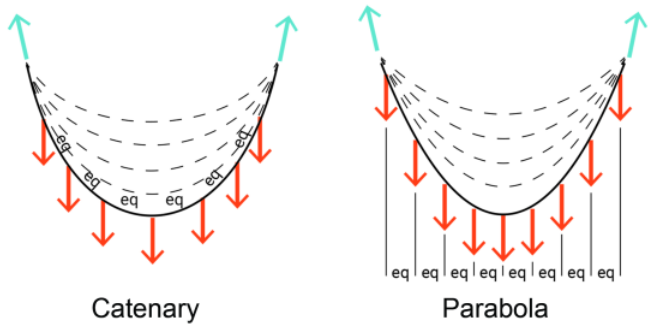


Figure 18.1. The Distribution of Weights in a Catenary and a Parabola

An equation for the catenary was proposed independently by Johann Bernoulli, Huygens, and Leibniz in 1691, in response to a challenge from Jakob Bernoulli the previous year. Johann showed that the curve satisfies the differential equation $dy/dx = s/a$ where a is a constant derived from the chain's weight per unit length, and s is its arc length. He also rewrote it as the integral

$$dx = \frac{a\,dy}{\sqrt{y^2 - a^2}}.$$

This was as far as matters could be taken at the time, since x is a transcendental function of y. Nowadays, it can be solved

$$y = a\cosh(x/a),$$

or written in terms of cosh()'s exponential definition,

$$y = a(e^{x/a} + e^{-x/a})/2.$$

The curve was first called a catenaria by Huygens in a letter to Leibniz, based upon the Latin word *catena* for 'chain', and the anglicized equivalent, 'catenary', is due to the (future) US President Thomas Jefferson, coined in a letter to Thomas Paine in 1788, about the design of a bridge.

Two classic books on curves which discuss the catenary are *A Book of Curves* by Lockwood [**Loc67**] and *A Handbook on Curves and Their Properties* by Yates [**Yat47**]. Details with an engineering maths emphasis can be found in chapter 7

of *How Round Is Your Circle?* by Bryant and Sangwin [**BS24**], and the catenary is placed in its historical context by Stillwell in [**Sti10**].

18.1.1 Why cosh() and sinh()? In 1757, Vincenzo Riccati introduced the use of hyperbolic functions to obtain the roots of certain equations, particularly cubics. Soon after, in 1759, Daviet de Foncenex showed how to interchange circular and hyperbolic functions by using i:

$$\sin\theta = -i\sinh i\theta, \qquad \text{and} \qquad \cos\theta = \cosh i\theta.$$

In a 1768 paper, Johann Heinrich Lambert adopted the notation we use today, including sinh and cosh, and helped popularize hyperbolic trigonometry.

cosh() and sinh() play a similar role for hyperbolas that cos() and sin() satisfy for circles. A unit circle, $x^2 + y^2 = 1$, can be parameterized using $x = \cos t$ and $y = \sin t$ since these trigonometry functions satisfy both

$$\cos^2 t + \sin^2 t = 1,$$

and the differential equation

$$y'' + y = 0.$$

A unit hyperbola has the form $x^2 - y^2 = 1$, and the question arises as to what functions can be used to parameterize it in a similar manner. The hyperbolic functions are the answer:

$$\cosh t = \frac{e^t + e^{-t}}{2} \qquad \text{and} \qquad \sinh t = \frac{e^t - e^{-t}}{2},$$

linked by

$$\cosh^2 t - \sinh^2 t = 1,$$

Their derivatives,

$$\frac{d}{dx}\cosh x = \sinh x \qquad \text{and} \qquad \frac{d}{dx}\sinh x = \cosh x,$$

can be utilized to solve the hyperbolic analogue of the circular differential equation:

$$y'' - y = 0$$

sinhCosh.py plots $\cosh(x)$, $\sinh(x)$, e^x, and e^{-x} (see Fig. 18.2) to illustrate the relationships between the curves: $\sinh(x)$ is half the difference of e^x and e^{-x} and $\cosh(x)$ is their average.

For more on the hyperbolic functions, any good book on calculus, such as Thomas and Finney [**TFW96**] will be helpful. In particular, you may want to refer to it for details on how various hyperbolic derivatives and integrals are derived, which we're going to take as given.

Python's `math` module contains implementations for cosh() and other hyperbolic functions (`https://docs.python.org/3/library/math.html`).

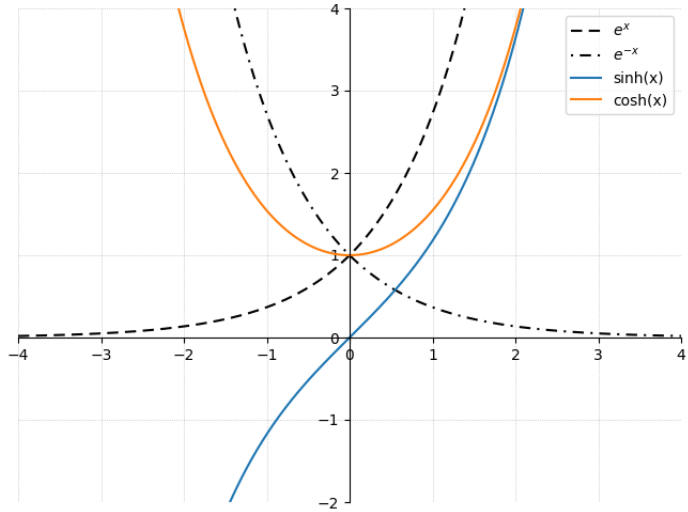


Figure 18.2. Plot of $\cosh(x)$, $\sinh(x)$, e^x, and e^{-x}

However, CPython doesn't implement these functions itself, instead calling the underlying platform's C math library, such as `libm` on Linux and macOS (have a look at CPython's `mathmodule.c` at `https://github.com/python/cpython/blob/main/Modules/mathmodule.c`). A relatively simple version, musl `libm` for embedded systems, can be found at `https://git.musl-libc.org/cgit/musl/tree/src/math`. The C implementation of cosh() is at `https://git.musl-libc.org/cgit/musl/tree/src/math/cosh.c`, and here's a pseudo-Pythonic version which seems very distant from $\cosh(x) = \frac{1}{2}(e^x + e^{-x})$:

```
def cosh(x):
  absX = abs(x)
  if absX < math.log(2):
    # for extremely small x, cosh(x) is approx 1
    if absX < 2**-26:
      return 1
    t = math.exp(absX) - 1
    return 1 + t * t / (2 * (1 + t))

  maxLog = 709.782712893384    # approx log() of MAX FLOAT
  if absX < maxLog:
    t = math.exp(absX)
    return 0.5 * (t + 1 / t)

  # for extremely large x
  return math.exp(absX)
```

Edge cases are dealt with by separate branches, and `1 + t*t/(2*(1+t))` is utilized for the general case rather than $\frac{1}{2}(e^x + e^{-x})$. The trouble with the standard definition is that it can suffer from catastrophic cancellation for small values of x due to the subtraction of nearly equal quantities in

$$\exp(x) \approx 1 + x + \frac{x^2}{2} \qquad \text{and} \qquad \exp(-x) \approx 1 - x + \frac{x^2}{2}.$$

One solution is to use $\exp(x) - 1$ and the identity

$$\cosh(x) = 1 + \frac{(\exp(x) - 1)^2}{2\,\exp(x)}.$$

Employing $t = \exp(x) - 1$, such that $\exp(x) = 1 + t$, produces the equation used in the code:

$$\cosh(x) = 1 + \frac{t^2}{2(1+t)}.$$

This is numerically stable because when x is small:

$$t = \exp(x) - 1 \approx x + \frac{x^2}{2} + \cdots,$$

since it doesn't involve the subtraction of nearly equal numbers.

The final return statement utilizes `math.exp(absX)` because the e^{-x} term is negligible for large x. Also, e^x can overflow, but that's handled by `math.exp()` returning `math.inf`.

18.1.2 Deriving the Catenary. We'll derive equations for the catenary with the help of Fig. 18.3. The tension T_0 at the lowest point A is directed horizontally. s is the length of the chain measured from A to any point P, and the tension at that point is T inclined by ψ to the horizontal. The weight per unit length of the chain is w.

The segment AP will be in equilibrium under the action of three forces: its weight ws, T_0, and the tension T at P. Resolving vertically and horizontally,

$$T \sin\psi = ws, \quad T\cos\psi = T_0.$$

For convenience, we introduce the constant a, such that $T_0 = wa$, which allows us to write

$$T \sin\psi = ws, \quad T\cos\psi = wa.$$

Dividing these equations

$$\tan\psi = \frac{s}{a} \qquad \text{or} \qquad s = a\tan\psi. \tag{18.1}$$

Equ. (19.1) is called the catenary's *intrinsic* equation, and a is its *parameter*.

To find the Cartesian version, start with

$$\tan\psi = \frac{dy}{dx}, \quad \text{and so} \quad \frac{dy}{dx} = \frac{s}{a}. \tag{18.2}$$

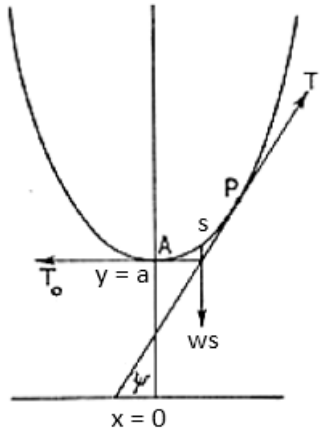


Figure 18.3. A Catenary Curve

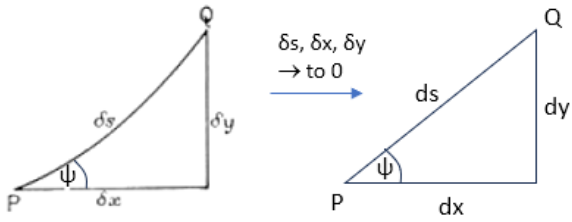


Figure 18.4. A Small Element δs of the Curve

Consider a small element δs of the curve joining two points P and Q, as in Fig. 18.4. Let the coordinates of P and Q be (x, y) and $(x+\delta x, y+\delta y)$ respectively. Then

$$(\delta s)^2 \simeq (\delta y)^2 + (\delta x)^2$$

so that

$$\left(\frac{\delta s}{\delta x}\right)^2 \simeq \left(\frac{\delta y}{\delta x}\right)^2 + 1$$

and

$$\left(\frac{\delta s}{\delta y}\right)^2 \simeq 1 + \left(\frac{\delta x}{\delta y}\right)^2$$

When δs, δx and $\delta y \to 0$, these become

$$\left(\frac{ds}{dx}\right)^2 = \left(\frac{dy}{dx}\right)^2 + 1 \tag{18.3}$$

and

$$\left(\frac{ds}{dy}\right)^2 = 1 + \left(\frac{dx}{dy}\right)^2 \tag{18.4}$$

Rearranging equ. (18.3) with the help of equ. (18.2):

$$\begin{aligned}\left(\frac{ds}{dx}\right)^2 &= 1 + \frac{s^2}{a^2} \\ \frac{ds}{dx} &= \frac{\sqrt{(a^2+s^2)}}{a} \\ dx &= \frac{a\,ds}{\sqrt{(a^2+s^2)}}\end{aligned}$$

Integrating:

$$x = a\sinh^{-1}\frac{s}{a} \qquad \text{or} \qquad s = a\sinh\frac{x}{a} \tag{18.5}$$

provided that $x = 0$ when $s = 0$.

Rearranging equ. (18.4) with the help of equ. (18.2):

$$\begin{aligned}\left(\frac{ds}{dy}\right)^2 &= 1 + \frac{a^2}{s^2} \\ \frac{ds}{dy} &= \frac{\sqrt{(s^2+a^2)}}{s} \\ dy &= \frac{s\,ds}{\sqrt{(s^2+a^2)}}\end{aligned}$$

Integrating:

$$y = \sqrt{(s^2+a^2)} \qquad \text{or} \qquad y^2 = s^2 + a^2 \tag{18.6}$$

provided that $y = a$ when $s = 0$.

Combining Equs. (18.5) and (18.6):

$$\begin{aligned}y^2 &= a^2\left(\sinh^2\frac{x}{a} + 1\right) \\ &= a^2\cosh^2\frac{x}{a}\end{aligned}$$

This gives us the Cartesian equation for a catenary:

$$\boxed{y = a\cosh\frac{x}{a}} \tag{18.7}$$

Since $T\sin\psi = ws$ and $T\cos\psi = wa$, then

$$T^2 = w^2(s^2 + a^2)$$

which, with the help of Equ. (18.6), is

$$\begin{aligned} T^2 &= w^2y^2 \\ T &= wy \end{aligned}$$

Thus the tension at any point in the catenary is proportional to the height of that point above the x-axis.

When a is large, Equ. (18.7) can be expanded as,

$$\begin{aligned} y = a\cosh\frac{x}{a} &= \frac{a}{2}(e^{x/a} + e^{-x/a}) \\ &= \frac{a}{2}\left\{1 + a + \frac{x^2}{2a^2} + \ldots + \left(1 - a + \frac{x^2}{2a^2} + \ldots\right)\right\} \\ &= a + \frac{x^2}{2a} + \ldots \end{aligned}$$

In this case, the curve is approximately a parabola.

18.1.3 Turning the Catenary Upside Down. If we turn Fig. 18.3 upside down to form an arch (producing Fig. 18.5), what was in tension is now in compression. This was first pointed out by Robert Hooke in 1675. The arch is only in compression, with no tension between the links, making an inverted catenary the strongest curve for building an arch.

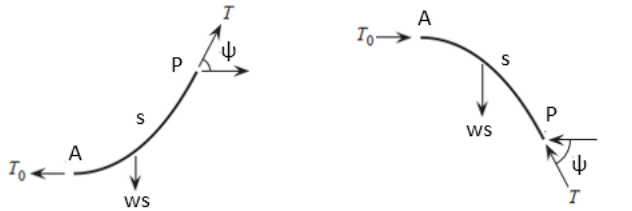


Figure 18.5. From Chain to Arch

18.1.4 The Parametric Form for the Catenary. We can differentiate Equ. (19.1) ($s = a\tan\psi$) to produce

$$\frac{ds}{d\psi} = a\sec^2\psi. \tag{18.8}$$

Also, from Fig. 18.4, we have

$$\frac{dx}{ds} = \cos\psi, \quad \frac{dy}{ds} = \sin\psi.$$

The dx/ds expression can be combined with Equ. (18.8)

$$\begin{aligned}\frac{dx}{d\psi} &= \frac{dx}{ds}\cdot\frac{ds}{d\psi} \\ &= \cos\psi\cdot a\sec^2\psi.\end{aligned}$$

Integrating both sides:

$$\begin{aligned}x &= \int a\sec\psi d\psi \\ &= a\log(\sec\psi+\tan\psi).\end{aligned}$$

Note that if x is measured from the catenary's lowest point (its *vertex*), then no constant term is required in the integral.

We can do a similar manipulation of the dy/ds expression and Equ. (18.8):

$$\begin{aligned}\frac{dy}{d\psi} &= \frac{dy}{ds}\cdot\frac{ds}{d\psi} \\ &= \sin\psi\cdot a\sec^2\psi.\end{aligned}$$

Integrating both sides:

$$\begin{aligned}y &= a\int\frac{\sin\psi}{\cos^2\psi}d\psi \\ &= a\sec\psi+\text{constant}.\end{aligned}$$

If y is measured from the vertex where ψ is zero, then the constant becomes $-a$. However, the usual convention is to place the origin at a distance a below the vertex so that $y=a$ when $\psi=0$, and then a constant isn't needed.

The collected parametric equations for the catenary are:

$$x = a\log(\sec\psi+\tan\psi) \qquad \text{and} \qquad y = a\sec\psi. \tag{18.9}$$

18.2 A Parabolic Chain

Fig. 18.6 depicts a chain with n equal weights (w) uniformly spaced out at horizontal intervals of h. This causes it to take on a parabolic shape rather a catenary, as we'll show below.

The weights are attached in arc length steps of $a_1, a_2, \ldots, a_n$, each inclined at $\theta_1, \theta_2, \ldots, \theta_n$ to the horizontal, and with tensions $T_1, T_2, \ldots, T_n$. The chain is tied to a point at (l,k) relative to the origin at the vertex O.

The equilibrium of the weights means that

$$T_1\cos\theta_1 = T_2\cos\theta_2 = T_3\cos\theta_3 = \ldots = T_n\cos\theta_n = H \text{ (say)}, \tag{18.10}$$

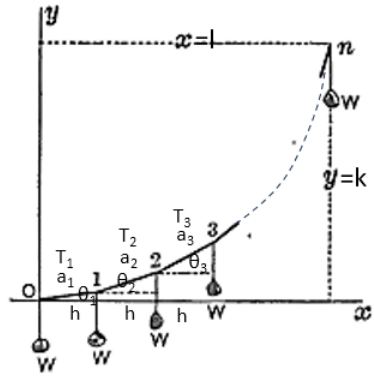


Figure 18.6. A Equally Weighted and Spaced Chain

and

$$\begin{aligned} T_2 \sin\theta_2 - T_1 \sin\theta_1 &= w, \\ T_3 \sin\theta_3 - T_2 \sin\theta_2 &= w, \\ &\vdots \\ T_n \sin\theta_{n+1} - T_{n-1} \sin\theta_n &= w. \end{aligned} \tag{18.11}$$

We have geometrical equations,

$$\begin{aligned} a_1 \cos\theta_1 + a_2 \cos\theta_2 + \ldots + a_n \cos\theta_n &= l, \\ a_1 \sin\theta_1 + a_2 \sin\theta_2 + \ldots + a_n \sin\theta_n &= k. \end{aligned}$$

Since the weights are equally spaced horizontally, we also have

$$a_1 \cos\theta_1 = a_2 \cos\theta_2 = \ldots = a_n \cos\theta_n = h \qquad \text{and} \qquad nh = l.$$

Dividing each of the equations in Equ. 18.11 by the relevant H equality in Equ. 18.10, we get

$$\begin{aligned} \tan\theta_2 - \tan\theta_1 &= \frac{w}{H}, \\ \tan\theta_3 - \tan\theta_2 &= \frac{w}{H}, \\ &\vdots \\ \tan\theta_n - \tan\theta_{n-1} &= \frac{w}{H}. \end{aligned}$$

These can be rearranged and simplified:

$$\begin{aligned}
\tan\theta_2 &= \tan\theta_1 + w/H, \\
\tan\theta_3 &= \tan\theta_2 + w/H \\
&= \tan\theta_1 + 2w/H, \\
\tan\theta_4 &= \tan\theta_3 + w/H \\
&= \tan\theta_1 + 3w/H, \\
&\vdots \\
\tan\theta_n &= \tan\theta_{n-1} + w/H \\
&= \tan\theta_1 + (n-1)w/H.
\end{aligned}$$

The nth coordinate (x, y) in the chain at step a_n and angle θ_n can be expressed as

$$\begin{aligned}
x &= nh \\
y &= h(\tan\theta_1 + \tan\theta_2 + \ldots + \tan\theta_n).
\end{aligned} \tag{18.12}$$

Each θ_r is replaced by its θ_1 expression, to get

$$y = h[n\tan\theta_1 + \frac{1}{2}n(n-1)w/H].$$

Use $x = nh$ (Equ. 18.12) to eliminate n, and express y in terms of x:

$$\frac{2Hh}{w}y = x^2 + \frac{2\tan\theta_1 H - w}{w}x,$$

which is an equation for a parabola.

18.3 Visually Comparing the Catenary and Parabola

sagCatParab.py and arclenCatParab.py plot a catenary and parabola together on the same graph, and include sliders to adjust parameters to see how the curves are affected.

18.3.1 Comparing Curves Using Sag. sagCatParab.py 'hangs' the curves from two endpoints at an equal height above the x-axis, and has them share the same vertex at the origin. This is only possible by allowing the two curves to have different lengths. A third dotted line shows the difference between the catenary's and parabola's y-values. The slider allows the sag (or dip, d) of the curves to be adjusted by changing the heights of the endpoints. Fig. 18.7 shows two screenshots of the plot.

It's apparent that as the sag d decreases that the differences between the two curves also decreases. It's not surprising that early investigators were fooled into thinking they were the same curve.

The subtraction in $y = a\cosh(x/a) - a$ lowers the catenary so its vertex is at the origin. The more difficult problem is to determine a given only the curve's

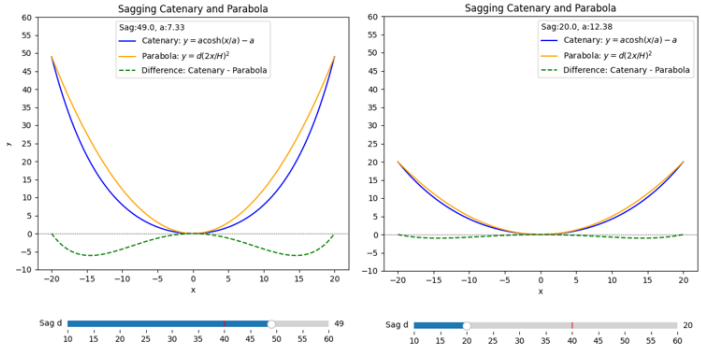


Figure 18.7. Adjusting the Sag of a Catenary and a Parabola

span H and sag d. This is handled by calling `catA()` in catSolver.py, which is explained in the next section.

18.3.2 The Parabola's Equation. If we know that three distinct points lie on the parabola then its equation can be determined by solving a linear system. Let the points be (x_1, y_1), (x_2, y_2), and (x_3, y_3), with x_1, x_2, x_3 pairwise distinct, and the quadratic that passes through those points be $y(x) = ax^2 + bx + c$. The coefficients satisfy

$$\begin{pmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}.$$

which can be solved for (a, b, c) by Gaussian Elimination (see section 6.8).

Fortunately, one of these points is the parabola's vertex, which simplifies matters. Suppose the vertex is (x_v, y_v), and the other two points are (x_1, y_1) and (x_2, y_2). The parabola's equation in terms of the vertex is

$$y = y_v + k(x - x_v)^2,$$

with only one unknown, k. Substitute one of the other points into the equation

$$y_1 = y_v + k(x_1 - x_v)^2,$$

and solve for k,

$$k = \frac{y_1 - y_v}{(x_1 - x_v)^2}.$$

The parabola becomes

$$y = y_v + \frac{y_1 - y_v}{(x_1 - x_v)^2}(x - x_v)^2.$$

In our case, $(x_v, y_v) = (0, 0)$, and so

$$y = \frac{y_1}{x_1^2} x^2.$$

Also, y_1 is equivalent to the curve's sag, d, and due to the symmetry of the two end points y_1 is half of the curve's span, H, away from the vertex. Using these values gives us:

$$y = \frac{4d}{H^2} x^2 \qquad \text{or} \qquad y = d\frac{2x}{H}^2$$

which is used by `sagCatParab.py` for plotting the parabola.

18.3.3 Comparing Curves Using their Length. `arclenCatParab.py` keeps the lengths of both curves the same, and suspends their ends between two fixed points at (-2,0) and (2,0). The slider allows the user to adjust the curves' length, as shown in Fig. 18.8. This time it's the two curve's vertices (lowest points) which vary over time. However, a vertex always remains midway between the two endpoints, at $x = 0$, so its only its y-coordinate that changes.

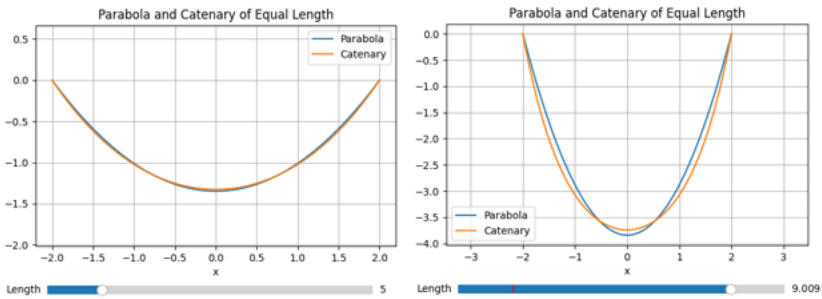


Figure 18.8. Adjusting the Length of a Catenary and a Parabola

The same relationship between the curves seen in `sagCatParab.py` also occurs in Fig. 18.8 – as the lengths decrease the curves become straighter, and the difference between them diminish.

The catenary's equation is derived from the two fixed symmetric endpoints and the arc length. Let the two endpoints be (x_1, y_1) and (x_2, y_1) so they're at the same height, and

$$h = \frac{x_1 + x_2}{2}$$

so the curve is symmetric about the line $x = h$, and the span as $H = x_2 - x_1$. Let $H_2 = H/2$, so the endpoints are at $x = h \pm H_2$.

A symmetric catenary can be written as

$$y = a\cosh\left(\frac{x-h}{a}\right) + k,$$

where k is a vertical shift that moves the catenary up or down so its endpoints remain at the same positions even as the curve's length changes.

The right-hand endpoint at $(h + H_2, y_1)$ satisfies

$$y_1 = a\cosh\left(\frac{H_2}{a}\right) + k.$$

so:

$$k = y_1 - a\cosh\left(\frac{H_2}{a}\right).$$

Substitute this into the catenary equation:

$$y(x) = y_1 + a\left[\cosh\left(\frac{x-h}{a}\right) - \cosh\left(\frac{H_2}{a}\right)\right].$$

This can be simplified a little since in our case $y_1 = 0$ and $h = 0$:

$$y(x) = a\left[\cosh\left(\frac{x}{a}\right) - \cosh\left(\frac{H_2}{a}\right)\right].$$

The length of the catenary between $x_1 = h - H_2$ and $x_2 = h + H_2$ is

$$\begin{aligned} L &= \int_{-H_2}^{H_2} \cosh\left(\frac{u}{a}\right) du \\ &= 2\int_0^{H_2} \cosh\left(\frac{u}{a}\right) du = 2a\sinh\left(\frac{H_2}{a}\right) \\ &= 2a\sinh\left(\frac{H}{2a}\right) \end{aligned}$$

This is a transcendental equation that needs to be solved for a when we have values for the curve's length L and span H. Once again, this is handled by calling `catA()` in catSolver.py, which is explained in the next section.

The code for generating the curve is located in `getCatenary()` in arclenCatParab.py:

```
def genCatenary(arcLen):
  a = getCatA(H=H, L=arcLen)              # H = horizontal width
  k = yCoord - a * math.cosh(H2 / a)      # H2 = H/2
  xs = [xLeft + i*dx for i in range(steps+1)]
  ys = [a*math.cosh(x/a) + k for x in xs]
  return xs, ys
```

18.3.4 The Parabola's Equation. Although the derivation of the parabola's equation starts out simply, it encounters a roadblock. Let the two endpoints be (x_1, y_1) and (x_2, y_1) and define h and H_2 as before so that the parabola's vertex is at (h, k) and the endpoints are at $x = h \pm H_2$. Because the parabola is symmetric about $x = h$, its equation can be written as

$$y = a(x-h)^2 + k.$$

Substituting in $(h + H_2, y_1)$ gives

$$y_1 = aH_2^2 + k,$$

so that

$$k = y_1 - aH_2^2.$$

Thus, the parabola may be written entirely in terms of a:

$$y = a(x - h)^2 + y_1 - aH_2^2.$$

Let $u = x - h$, which makes the derivative

$$y' = 2au.$$

The length L of the parabola between $x = h - H_2$ and $x = h + H_2$ is therefore

$$L = \int_{h-H_2}^{h+H_2} \sqrt{1 + (y')^2}\, dx = 2 \int_0^{H_2} \sqrt{1 + 4a^2u^2}\, du. \tag{18.13}$$

This has a closed form integral, but it's ugly, and it's more interesting to see how a can be obtained by summing the y' integral, as shown below.

Once a is found, the parabola is calculated using

$$y = a(x - h)^2 + y_1 - aH_2^2$$

This can be simplified to $y = a(x^2 - H_2^2)$ since $y_1 = 0$ and $h = 0$.

The code for generating the curve is located in `getParabola()` in arclenCatParab.py:

```
def genParabola(arcLen):
  a = solveParabolaA(arcLen)
  xs = [xLeft + i*dx for i in range(steps+1)]
  ys = [a*(x*x - H2*H2) + yCoord for x in xs]  # H2 = H/2
  return xs, ys
```

The actual hard work is carried out inside `solveParabolaA()`. It relies on `paraArcLen()` which uses a value for a to calculate the curve length by summing the y' integral from equ. 18.13 from $x = h - H_2$ to $x = h + H_2$.

```
def paraArcLen(a):
  length = 0.0
  x = xLeft    # x = h-H2
  for _ in range(steps):   # stops at x = h+H2
    slope = 2*a*x          # the derivative y'
    ds = math.sqrt(1 + slope*slope) * dx
    length += ds
    x += dx
  return length
```

The a passed to `paraArcLen()` is obtained by a binary search through a range of possible values until the length calculated by `paraArcLen()` is close enough to the known length of the curve (`arcLen`):

```
def solveParabolaA(arcLen):
  lowA = 0.0
  highA = 5.0
  for _ in range(40):
    midA = 0.5 * (lowA + highA)
    if paraArcLen(midA) > arcLen:
      highA = midA
    else:
      lowA = midA
  return 0.5 * (lowA + highA)
```

We coded things this way to illustrate another approach to finding a, but in the next section we'll use SciPy's `brentq()` solver for the equations associated with the catenary.

18.4 Determining the Catenary Parameter a

Determining the a parameter for a particular catenary can vary from being very simple to requiring numerical root-finding which can be quite tricky due to rounding and overflow errors. For these more complicated cases, we'll utilize the dimensions in Fig. 18.9. The ends of the catenary are at the same height, H is its span, d the dip (or sag), L the total length, and the vertex is a distance a above the x-axis.

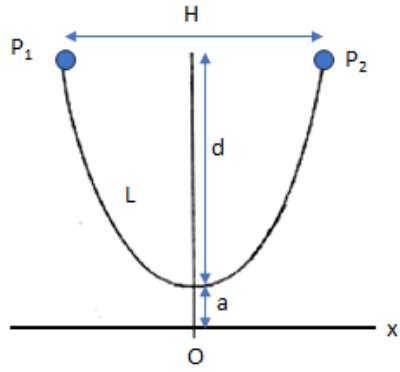


Figure 18.9. The Dimensions of a Catenary

The most common ways to determine a utilize:

(1) **Horizontal span (H) and sag (d).** This requires you to numerically solve

$$d = a\left(\cosh\left(\frac{H}{2a}\right) - 1\right)$$

for a. Unfortunately, it's quite difficult to catch all the edge cases that can produce incorrect answers or cause the solver to fail. One concern is highlighted by the Taylor expansion of cosh():

$$\cosh(u) \approx 1 + \frac{u^2}{2} + \frac{u^4}{24} + \cdots$$

the equation becomes

$$d = a\left(\cosh\left(\frac{H}{2a}\right) - 1\right) \approx a\left(\frac{(H/(2a))^2}{2}\right) = \frac{H^2}{8a}$$

$\cosh(u) - 1$ involves subtracting two nearly equal numbers, potentially causing catastrophic cancellation in the floating-point arithmetic. Also, if the sag (d) is small compared to the span (H), as when the catenary is nearly horizontal, then any slight error in d's measurement will produce huge variations in a.

Another concern is that certain numerical methods, such as Newton-Raphson and bisection, aren't well suited to solving transcendental equations. This drawback is avoided in `catA()` in catSolver.py by using Brent's method imported from the SciPy module (`https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.brentq.html`). However, that solver introduces its own concerns regarding setting the bounds around the root.

(2) **Horizontal span (H) and curve length (L).** Numerically solve:

$$L = 2a\sinh\left(\frac{H}{2a}\right)$$

for a.

(3) **Sag (d) and curve length (L).** This has an explicit solution,

$$a = \frac{L^2/4 - d^2}{2d}$$

18.5 Using a Catenary to Calculate Logarithms

'How to Find the Logarithm of Any Number Using Nothing but a Piece of String' by Viktor Blasjö [**Bla16**] describes how Leibniz utilized the catenary to calculate logarithms. His recipe is summarized in Fig. 18.10

One half of a catenary is extended horizontally so that a line can be drawn from its endpoint M to the chain's top midpoint at H. The line is bisected at B, and a normal to HM constructed through that point. Meanwhile, a perpendicular is dropped from H, and where it and the normal intersect is marked as the origin O. The distance from the origin to the lowest point of the catenary (its vertex at A) is considered to be a unit length. In that case, the catenary's equation will

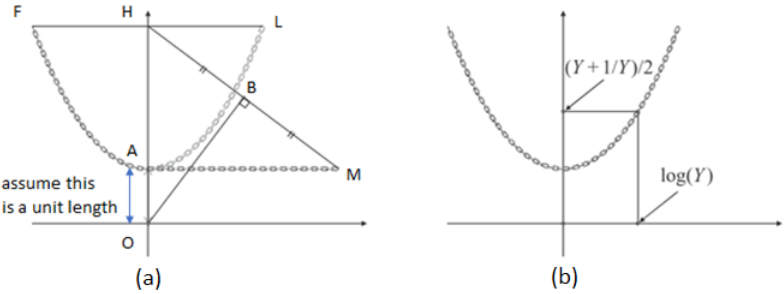


Figure 18.10. Leibniz's Recipe for Determining Logarithms from a Catenary

be $y = (e^x + e^{-x})/2$, and the logarithm tool is ready for use in Fig. 18.10(b). To determine $\log(Y)$, locate $(Y + 1/Y)/2$ on the y-axis and draw a line across to the curve, then down to the x-axis.

Why does this work? We have $y = a\cosh(\frac{x}{a})$, where $a = OA$ in Fig. 18.10(a). Since we've assigned this a value of 1, then $y = \cosh(x) = \frac{1}{2}(e^x + e^{-x})$. We also know that $y^2 = a^2 + s^2$ (see equ. (18.6)), or $y = \sqrt{1 + s^2}$. In Fig. 18.10(a), $y = OH$ and $s = AL$, and the horizontal segment AM has the same length. It follows by Pythagoras that $OM = \sqrt{1 + s^2}$, and $y = \sqrt{1 + s^2}$, which means that $OH = OM$. Thus, $\bigtriangleup OHM$ is isosceles, and the perpendicular bisector of its base HM will pass through O. So the construction does indeed give set up the coordinate system for $y = (e^x + e^{-x})/2$.

It may not be immediately obvious that the relationship in Fig. 18.10(b) is true. Let $x = \log(Y)$, and use it to simplify $y = \frac{1}{2}(e^x + e^{-x})$. It's also possible to test things in Python, as in the following examples:

```
>>> Y = 5
>>> math.acosh((Y + (1/Y))/2)
1.6094379124341003
>>> math.log(Y)
1.6094379124341003

>>> Y = 8
>>> math.acosh((Y + (1/Y))/2)
2.0794415416798357
>>> math.log(Y)
2.0794415416798357
```

18.6 Rolling a Parabola

The catenary and parabola are related in another way apart from looking very similar. If a parabola is rolled along a straight line, its focus traces out a catenary. rollParabola.py animates this relationship: the parabola keeps rolling right and left along the x-axis, reversing direction when the tip of an arm touches the axis. The position of its focus is recorded and its path displayed as a red curve. In addition, $y = \frac{1}{4a}\cosh(4ax)$ is drawn as a dashed gray line, revealing that the focus is following that curve (see Fig. 18.11).

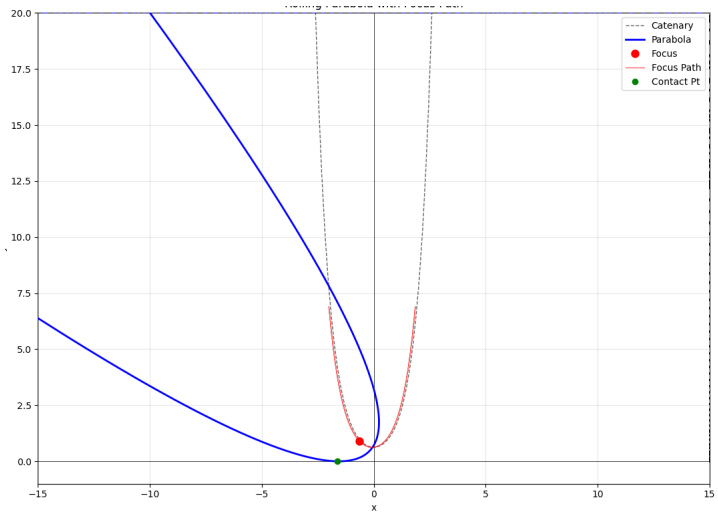


Figure 18.11. An Animated Rolling Parabola Tracing out a Catenary

rollParabola.py starts by generates all of the points for the parabola at the origin. Then the arc length of each of these points from the vertex is calculated: `calcLengths()` measures the distance between adjacent points and increments a total length value.

Matplotlib calls `animate()` each time an animation frame needs to be drawn. The parabola at the origin is rotated around the origin and its new coordinates are searched to find the lowest one. The curve is shifted up the y-axis so this 'contact' point is resting on the x-axis, and the point's corresponding arc length is used to move the curve the correct distance to the right or left.

The rotation and translation of the curve are also applied to the focus, which starts at $(0, 1/(4a))$. Its new position is recorded in a list which is used to draw its path.

The reason for the focal point being at $(0, 1/(4a))$ is due to the parabola having the equation $y = ax^2$. The link between a parabola and its focus is that any point

on the curve will be an equal distance from the focus and the length of a vertical line dropped to the directrix at $y = -f$ (see Fig. 18.12). This is discussed in more detail in section 16 on conics.

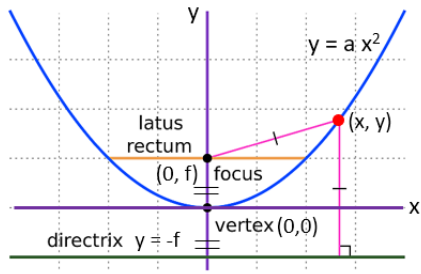


Figure 18.12. Linking f and a for a Parabola

Proving that the path of the focus is a catenary is a bit too involved to be investigated here, but we'll give some pointers to information on that topic at the end of this section. Instead, we'll follow Sánchez-Reyes [**SR12**] and demonstrate the equivalence of the path to the equation for cosh.

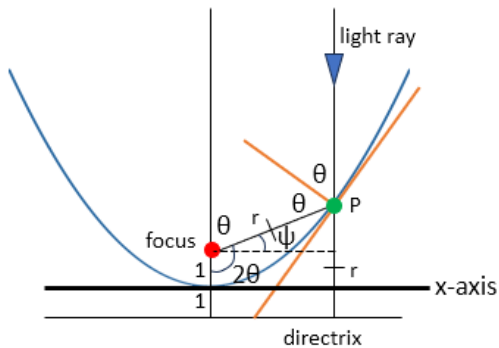


Figure 18.13. The Parabola Before Rolling

The parabola in Fig. 18.13 displays some useful facts – point P is an equal distance r from the focus and directrix, and a beam of light hitting the parabola vertically is reflected through the focus. We've labeled the angle of incidence and reflection as θ, and the focus is fixed to be 2 units above the directrix. $\psi =$

$2\theta - \pi/2$, and so

$$\begin{aligned} r &= 2 + r\sin(2\theta - \pi/2) \\ &= 2 + r(-\cos(2\theta)) \\ r(1 + \cos(2\theta)) &= 2 \end{aligned}$$

With the help of the double-angle identity, $1 + \cos(2\theta) = 2\cos^2\theta$:

$$\begin{aligned} r \cdot 2\cos^2\theta &= 2 \\ r &= \sec^2\theta \end{aligned}$$

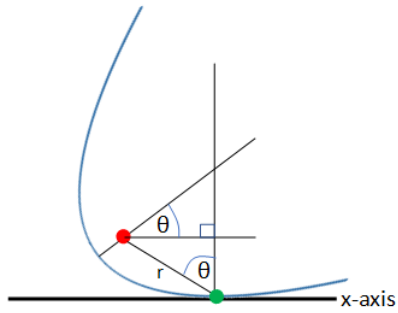


Figure 18.14. The Parabola After Rolling

Fig. 18.14 shows the parabola after it has rolled a short distance to the right. The coordinate axes now pass through the green point, which has become the center of rotation. This allows us to write

$$\frac{dy}{dx} = \tan\theta \qquad \text{and} \qquad y = r\cos\theta.$$

The second equation gives us the height of the focus as it rolls along the x-axis, and can be rewritten using the r equation and the derivative:

$$\begin{aligned} y &= r\cos\theta \\ &= \sec^2\theta \cdot \cos\theta \\ &= 1/\cos\theta \quad = \sec\theta \end{aligned}$$

Recall that $1 + \tan^2\theta = \sec^2\theta$, so

$$\begin{aligned} y &= \sec\theta \\ &= \sqrt{1 + \tan^2\theta} \\ &= \sqrt{1 + (dy/dx)^2} \end{aligned}$$

This equation specifies how the focus moves around the current center of rotation. Is there another equation with a similar relationship between y and dy/dx?

Turning to the hyperbolic functions, and using the identities:

$$\frac{d}{dx}\cosh x = \sinh x \qquad \text{and} \qquad \cosh^2 x - \sinh^2 x = 1$$

then

$$\begin{aligned} \cosh x &= \sqrt{1 + \sinh^2 x} \\ &= \sqrt{1 + \frac{d}{dx}(\cosh x)^2} \end{aligned}$$

If we set $y = \cosh(x)$, then the equation becomes:

$$y = \sqrt{1 + (dy/dx)^2}$$

In other words, the same general form. This is not conclusive proof that the focus' path follows a catenary, but it's certainly suggestive. A complete proof can be found in [**AM10**], where Agarwal and Marengo show that if the parabola is $y = ax^2$ then the path's equation is

$$y = \frac{1}{4a}\cosh 4ax$$

Another proof can be found at
`https://www.anirdesh.com/math/calculus/parabola-rolling-proof.php`, with a lot of additional analysis.

18.7 The Tractrix

In 1676, Frenchman Claude Perrault placed his pocket watch on a table, pulled the end of its chain along the table's edge, and asked: What's the shape of the curve traced out by the watch? The first known solution was given by Christiaan Huygens in a letter to a friend in 1693 where he also coined the name "tractrix", from the Latin *tratus* for something pulled along.

We've discussed the tractrix before, in the context of curves of pursuit in section 6.14.2, and return to it here because it can also be generated by taking the *involute* of a catenary.

Fig. 18.15 shows the geometrical relationships involved. The catenary's equation is $y = a\cosh(x/a)$, with its vertex C located at $(0, a)$. Imagine a string wrapped tightly over the right half of the catenary, ending at C, which is gradually pulled away from the curve so that the untethered string forms a tangent PQ. Q is a point on the tractrix, and P lies on the catenary. PQ has the same arc length as PC, labeled as s in Fig. 18.15, and the line segment of length a normal to PQ always touches the x-axis (currently at R). This can be confirmed by using the catenary equation $y^2 = s^2 + a^2$ (equ. (18.6) from section 18.1). Another equation applicable to Fig. 18.15 is $\tan\psi = s/a$ (equ. 19.1). Q corresponds to Perrault's watch, QR is the chain, and OR is the pulling direction.

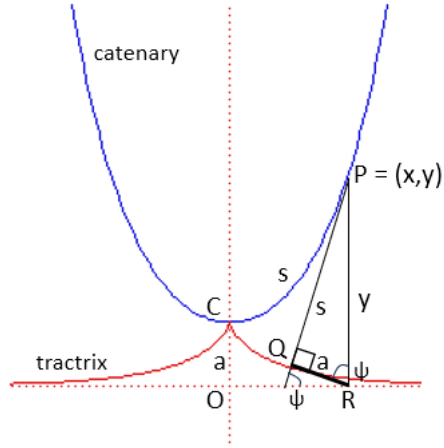


Figure 18.15. The Tractrix as an Involute of a Catenary

The left branch of the tractrix can be constructed in a similar way, but by starting with a string wrapped around the left half of the catenary, ending at C, and then pulling the string to the left.

Section 6.14.2 gives a Cartesian equation for the tractrix:

$$x = a \ln\left(\frac{a + \sqrt{a^2 - y^2}}{y}\right) - \sqrt{a^2 - y^2},$$

assuming that when $x = 0$, $y = a$. If we don't mind using hyperbolic functions, this can be rewritten. Let $y = a \text{ sech } u$ so that

$$\sqrt{a^2 - y^2} = a\sqrt{1 - \text{sech}^2 u} = a \tanh u,$$

and

$$\frac{a + \sqrt{a^2 - y^2}}{y} = \frac{a + a \tanh u}{a \text{ sech } u} = \cosh u + \sinh u = e^u.$$

Hence

$$\ln\left(\frac{a + \sqrt{a^2 - y^2}}{y}\right) = u,$$

which allows us to rewrite

$$\begin{aligned} x &= a \ln\left(\frac{a + \sqrt{a^2 - y^2}}{y}\right) - \sqrt{a^2 - y^2} \\ &= a\, u - a \tanh u \end{aligned}$$

paraTractrix.py plots $x = u - \tanh u$ and $y = \text{sech } u$ as shown in Fig. 18.16.

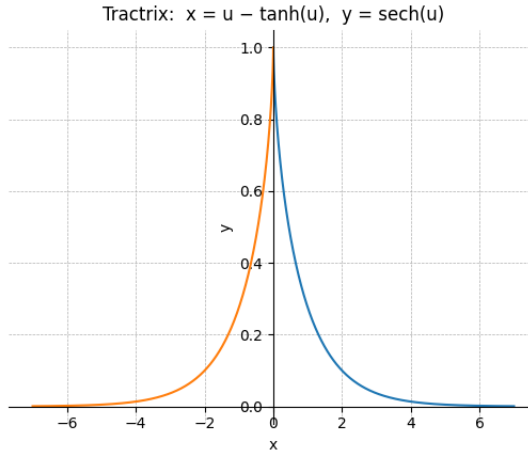


Figure 18.16. Plotting a Parametric Tractrix

18.7.1 Generating a Catenary with Tractrix Normals. Fig. 18.15 relates a point Q on a tractrix to a point P on the catenary by the normal QP to the tangent QR. This gives us a simple way to generate a catenary as an *envelope* of normals as shown in Fig. 18.17. It was drawn by `tractrixNormals.py`, with a little help from our Vector class, `Vec.py` (see section 14.4).

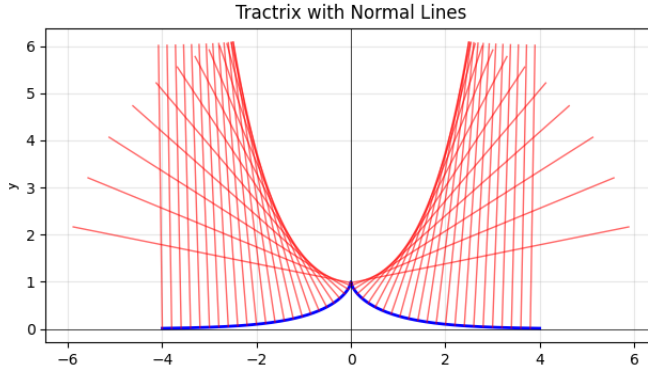


Figure 18.17. Drawing a Catenary with Tractrix Normals

To obtain a normal, we need a tangent, which can be produced by differentiating the parametric equations:

$$x(u) = u - \tanh u, \qquad y(u) = \operatorname{sech} u.$$

Differentiating with respect to u:

$$\begin{aligned}\frac{dx}{du} &= 1 - \frac{d}{du}(\tanh u) \\ &= 1 - \operatorname{sech}^2 u \\ &= \tanh^2 u.\end{aligned}$$

and

$$\begin{aligned}\frac{dy}{du} &= \frac{d}{du}(\operatorname{sech} u) \\ &= -\operatorname{sech} u \tanh u.\end{aligned}$$

These equations are utilized in `tractrix()` and `tractrixDerive()` in tractrixNormals.py to obtain a tractrix point and its tangent:

```
def tractrix(u, sign):
  x = sign * (u - math.tanh(u))
  y = 1 / math.cosh(u)
  return Vec(x, y, 0)

def tractrixDeriv(u, sign=+1):
  duDt = sign * math.tanh(u)**2
  duDt = -1/math.cosh(u) * math.tanh(u)
  return Vec(duDt, duDt, 0)
```

The tangent is converted to a normal in `getNormal()`:

```
def getNormal(tangent, sign):
  tangent = tangent.normalize()
  # For left branch (sign=-1), rotate clockwise;
  # for right branch, counterclockwise
  if sign == -1:
    normal = Vec(tangent.y, -tangent.x, 0)
  else:
    normal = Vec(-tangent.y, tangent.x, 0)
  normal *= NORMAL_LEN      # scale
  return normal
```

One slight quirk is that we have to check the tractrix branch to decide which way the normal vectors should point so they're aimed up the y-axis.

18.7.2 Generating a Tractrix with Circles. Yet another way to generate a tractrix, that doesn't involve a catenary, but is still interesting. Draw a large group of identical circles whose centers all lie along a straight line. Starting at the top of the first circle, draw a line to its center, stopping when your reach the edge of the next circle to the right. The process now repeats but linking the stopping point to its circle's center. The collection of points will *approximate* a tractrix, with the quality of the match improving with the number of circles along the line.

tractrixOrtho.py generates Fig. 18.18. The lower curve is generated in the same manner, but starting from the bottom of the leftmost circle.

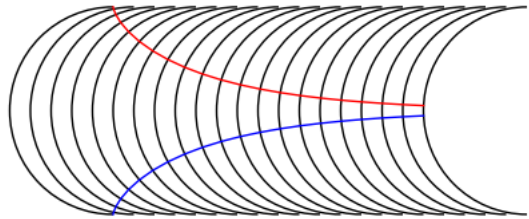


Figure 18.18. Drawing a Tractrix with Circles

The coding problem here is determining the intersection between a line segment and a circle. Inside `lineCircleIntersection()` in tractrixOrtho.py, the line (supplied as two points p_1 and p_2) is written in parametric form:

$$(x, y) = p_1 + t(p_2 - p_1)$$

To find the intersection, this is substituted into the equation for a circle:

$$(x - h)^2 + (y - k)^2 = r^2,$$

where (h, k) is the circle's origin, and r is its radius.

The components are rearranged as a quadratic in terms of t: $at^2 + bt + c = 0$, which is solved. In fact, the values for a, b, and c are hardwired into the code as:

```
a = dx*dx + dy*dy
b = 2 * (fx*dx + fy*dy)
c = fx*fx + fy*fy - r*r
```

where `dx` and `dy` are the coordinates of $p_2 - p_1$, and `fx` and `fy` are the coordinates of $p_1 - (h, k)$. Also, the quadratic's determinant, $D = b^2 - 4ac$, is examined to see if there are 0, 1, or 2 intersections. Each valid t is turned into an intersection point, and `collectPoints()` chooses the correct point if two intersections were found, using the one nearest to the other point forming the line segment.

18.8 Riding a Bike with Square Wheels

Fig. 18.19 shows a bike with square wheels at Macalester College, constructed by mathematician Stan Wagon and bicycle mechanic Loren Kellen in 1997 (`https://www.macalester.edu/mscs/multimedia/squarewheeledbike/squarewheelbike/`).



Figure 18.19. A Bicycle with Square Wheels

The most relevant question here is what road surface is required to make riding this bike a reasonable possibility. Of course, the answer is a catenary, as we'll show below.

Gerson Robison is credited with this discovery in 1960 [**Rob60**], although it became more widely known through the efforts of Stan Wagon and Leon Hall [**Wag99, HW92**]. Often forgotten is the very early work on Hengist Pod, inventor of the square wheel in around 54 BC as documented in *Carry on Cleo* [**Tho64**], and shown in Fig. 18.20.



Figure 18.20. Hengist Pod Demonstrating his Invention

We obviously want the bike's wheels to roll smoothly, which means that the center of each wheel should move horizontally. There should be no slipping, which implies that a wheel will form a tangent to the road's surface at the point of contact, and its center is directly above that contact point.

Fig. 18.21 shows a wheel of side length $2a$ at the top of a crest in the road. As the wheel rolls, the center will always remain a height h above the x-axis.

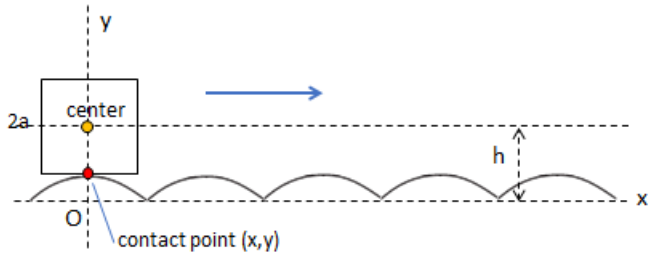


Figure 18.21. The Initial Square Wheel

This height requirement means that even when the wheel has rolled into a valley that its center will still be h above the axis (see Fig. 18.22). Since the wheel has a side length of $2a$, then $h = a\sqrt{2}$.

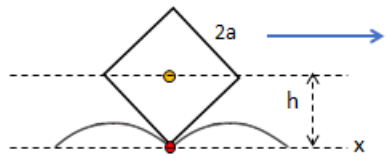


Figure 18.22. The Square Wheel in a Cusp

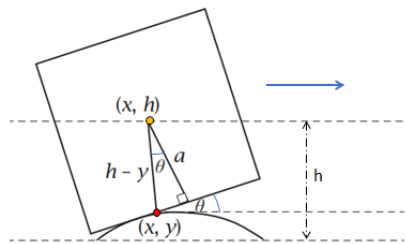


Figure 18.23. The Wheel in a General Position

We're now ready to consider the general case shown in Fig. 18.23. The wheel is inclined at an angle θ to the horizontal, with the point of contact at some (x, y) coordinate, so that:

$$a = (h - y)\cos\theta = \frac{h - y}{\sec\theta} = \frac{h - y}{\sqrt{1 + \tan^2\theta}} = \frac{h - y}{\sqrt{1 + (dy/dx)^2}}.$$

Therefore,

$$\frac{dy}{dx} = \pm\sqrt{((h-y)/a)^2 - 1}. \tag{18.14}$$

We'll use the positive part of this equation, which implies that the wheel is rolling up the incline (as in Fig. 18.23),

$$\begin{aligned} \frac{dy}{dx} &= \sqrt{\left(\frac{(h-y)^2-a^2}{a^2}\right)} \\ \int \frac{1}{\sqrt{(h-y)^2-a^2}}\, dy &= \frac{1}{a}\int dx \end{aligned}$$

After integrating:

$$\begin{aligned} -\cosh^{-1}\frac{y-h}{a} &= \frac{x}{a} + c \\ y &= -a\cosh\left(\frac{x}{a}+c\right) + h. \end{aligned}$$

Fig. 18.21 indicates that when the wheel first starts rolling that its contact point $(x, y) = (0, h-a)$. Substituting this into the equation makes $\cosh(c) = 1$ and so $c = 0$. The equation for the road's surface becomes,

$$y = h - a\cosh\left(\frac{x}{a}\right) \tag{18.15}$$

Since $h = a\sqrt{2}$, then

$$y = a\sqrt{2} - a\cosh\left(\frac{x}{a}\right), \tag{18.16}$$

an inverted catenary.

At the joint between two curves in the road, as in Fig. 18.22, the contact point is touching the x-axis so $y = 0$ and Equ. 18.14 can be written as

$$\frac{dy}{dx} = \sqrt{((a\sqrt{2}-0)/a)^2 - 1} = 1,$$

This implies that $\tan\theta = 1$, so the angle between the wheel and x-axis is $\pi/4$ in Fig. 18.22. Of course, each corner takes up $\pi/2$, so the wheel fits snugly into the gap between the crests.

To determine the length of a curved section of the road, let's measure from $x = 0$ (the crest in Fig. 18.21) to the next valley at $(x_1, 0)$. The total length, s, will be double this distance:

$$\begin{aligned} s &= 2\int_0^{x_1} \sqrt{1 + \left(\frac{dy}{dx}\right)^2}\, dx \\ &= 2\int_0^{x_1} \sqrt{1 + \left(\frac{h-y}{a}\right)^2 - 1}\, dx \\ &= 2\int_0^{x_1} \frac{h-y}{a}\, dx \end{aligned}$$

Use Equ. 18.15 to equate $(h-y)/a = \cosh(\frac{x}{a})$:

$$\begin{aligned} s &= 2\int_0^{x_1} \cosh\frac{x}{a}\,dx \\ &= 2\left[a\sinh\frac{x}{a}\right]_0^{x_1} \\ &= 2a\sinh\frac{x_1}{a}, \end{aligned}$$

Plug the coordinate of the second cusp, $(x_1, 0)$, into Equ. 18.16:

$$\begin{aligned} a\sqrt{2} - a\cosh\left(\frac{x_1}{a}\right) &= 0 \\ \cosh\left(\frac{x_1}{a}\right) &= \sqrt{2} \end{aligned}$$

Using the identity $cosh^2u - sinh^2u = 1$:

$$\begin{aligned} cosh^2(x_1/a) - sinh^2(x_1/a) &= 1 \\ 2 - sinh^2(x_1/a) &= 1 \\ \sinh(x_1/a) &= 1. \end{aligned}$$

Substitute this into the s equation to obtain:

$$s = 2a\sinh\frac{x_1}{a} = 2a$$

So, the length of each curved section is the same as the length of the side of the square, which means that the wheel can rotate between cusps without any slippage.

Hall and Wagon [**HW92**] go into considerably more detail than we have, considering a variety of road surfaces (what Robison terms 'rockers') and wheel shapes ('rollers'). For example, the wheel can be any regular polygon *except* an equilateral triangle. The corner of a triangle hits the road surface before it enters a valley, and scrapes against it again on the way out. This is quite surprising since the angle between the two sides of the adjacent catenaries is 90°, more than enough room for the triangle's corners. As the number of sides in the polygon gets larger, it begins to look more and more like a circle and the line of catenaries will get flatter, looking increasingly like a flat road.

Although a ride over inverted catenaries feels smooth, it's not *quite* as smooth as a regular bicycle moving over a flat surface. For an ordinary bike, the horizontal distance traveled, x, is linearly proportional to the angle θ pedaled – pedal twice as much and you travel twice as far. This is almost true of the square wheel, but not exactly, as shown by Fig. 18.24.

If the distance traveled is $y = f(x)$, then $d\theta/dx = -1/f(x)$. We know $y \approx -\cosh(x)$, and so after integration $\theta(x) \approx 2\arctan(-e^{-x})$. An examination of the Taylor series for arctan() highlights its small discrepancy from linearity.

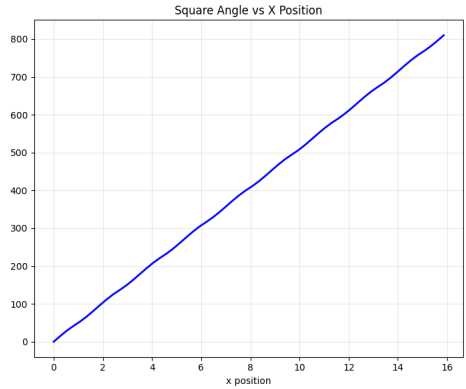


Figure 18.24. θ versus x for the Rolling Wheel

Fig. 18.24 is generated by rollingSquare.py, which also animates the path of a square wheel over a series of inverted catenaries in Fig. 18.25. `rollingSquare.py` was inspired by a much more feature-rich program in Wagon's *Mathematica in Action* [**Wag10**].

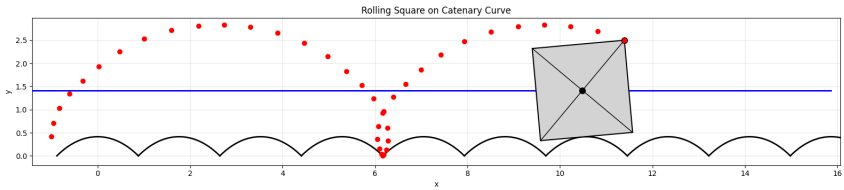


Figure 18.25. Animating the Journey of a Square Wheel

The series of red dots record the journey of one of the corners of the square, while the center of the square never deviates from the blue horizontal line. Note, that the corner occasionally goes backwards, reminiscent of the classic puzzle about the locus of the point at the bottom of the flanged wheel of a train [**Dar04**]. This is a *prolate* cycloid, the path traced out by a fixed point at a radius $b > a$, where a is the radius of a rolling circle; in our case, $b = h = a\sqrt{2}$.

An important point to remember when viewing `rollingSquare.py`'s animation is that no kind of collision detection is being carried out. A square shape is being rotated around its origin as it moves across the window by following the blue line. Its smooth traversal is *not* the result of it rolling over the catenaries. At best, the animation is an indication that the edges of the rotating square always seem to be just above the catenaries that it passes over.

What are the parameters required to produce this effect? There are three: `ir`, `cr`, and `hArch`, as shown in Fig. 18.26.

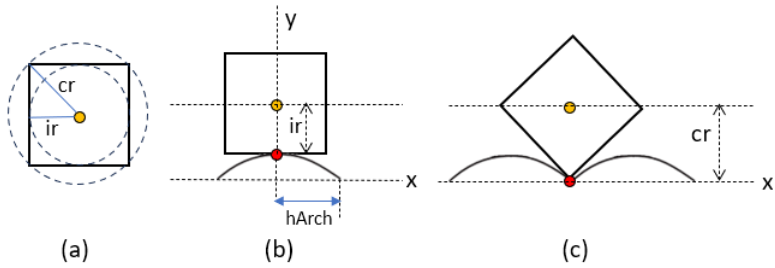


Figure 18.26. The Parameters used by `rollingSquare.py`

The four corners of the square are defined in terms of a circumradius (`cr`) and inradius (`ir`) (see Fig. 18.26(a)), which are set to 1 and $\sqrt{2}$ respectively. These variables are also used to determine `hArch`, the half-width of each catenary arch drawn along the floor (see Fig. 18.26(b)).

Each 'bump' on the floor is an inverted catenary, with an equation based on $y = -a\cosh(x/a)$ like the one in Fig. 18.27.

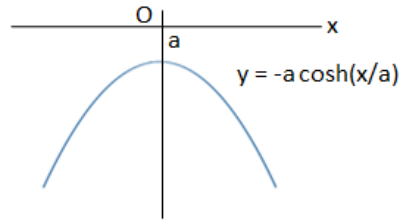


Figure 18.27. An Inverted Catenary

If the x-axis was the horizontal line through the center of the square, then a would equal `ir` as indicated best by Fig. 18.27(b). However, since the x-axis is `cr` units down from that line (as seen in Fig. 18.27(c)) then a corresponding shift must be applied to the catenary, producing:

$$y = \mathrm{cr} - \mathrm{ir} \cdot \cosh(x/\mathrm{ir})$$

The square is shown at the top of a crest in Fig. 18.26(b), when $(x, y) = (0, 0)$. y will next be 0 when $x =$ `hArch`. Plug this y value into the catenary equation, and

see what x is returned:

$$\begin{aligned} \text{cr} - \text{ir} \cdot \cosh(x/\text{ir}) &= 0 \\ \sqrt{2} - \cosh(x) &= 0 \\ \cosh(x) &= \sqrt{2} \end{aligned}$$

Using the identity, $cosh^2 x - sinh^2 x = 1$:

$$\begin{aligned} 2 - \sinh^2(x) &= 1 \\ x &= asinh(1) \end{aligned}$$

and so the code sets `hArch = math.asinh(1)`.

The first inverted catenary is drawn at $x = -\text{hArch}$, and subsequent catenaries are `2*hArch` apart. `catenaryPts()` in rollingSquare.py generates the coordinates for the ith catenary along the x-axis:

```
def catenaryPts(i, cr, ir, hArch, numPts=100):
  xs = []; ys = []
  xMin = -hArch + 2*hArch*i
  xMax = hArch + 2*hArch*i
  for j in range(numPts):
    x = xMin + (xMax - xMin)*j/(numPts - 1)
    y = cr - ir*math.cosh((x - 2*hArch*i)/ir)
        # catenary equation, offset by 2*hArch*i
    xs.append(x)
    ys.append(y)
  return xs, ys
```

18.9 The Catenoid

A catenoid is the surface of revolution produced when a catenary like the one in Fig. 18.3 is rotated about its x-axis. The catenoid, first described by Leonhard Euler in 1740, is the oldest known *minimal surface* – a shape of least area when bounded by a given closed space. This can be demonstrated nicely by a soap film stretched between two rings (see Fig. 18.28).

The catenoid can be defined by the parametric equations:

$$\begin{aligned} x &= a\cosh(\tfrac{v}{a})\cos u \\ y &= a\cosh(\tfrac{v}{a})\sin u \\ z &= v \end{aligned}$$

where $u \in [-\pi, \pi)$, $v \in \mathbb{R}$ and a is a non-zero real constant.

catenoid.py uses these equations to plot the 3D surface shown in Fig. 18.29. The somewhat mysterious a constant can be adjusted using a slider, revealing that it controls the size of the catenoid's 'waist' – reducing a makes it thinner, while a larger a makes the shape more cylindrical.

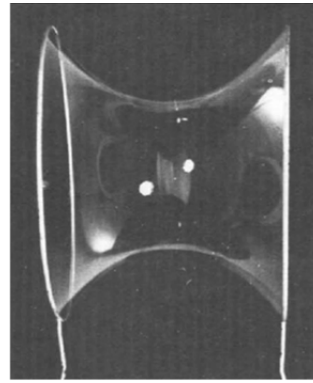


Figure 18.28. A Soap-film Catenoid

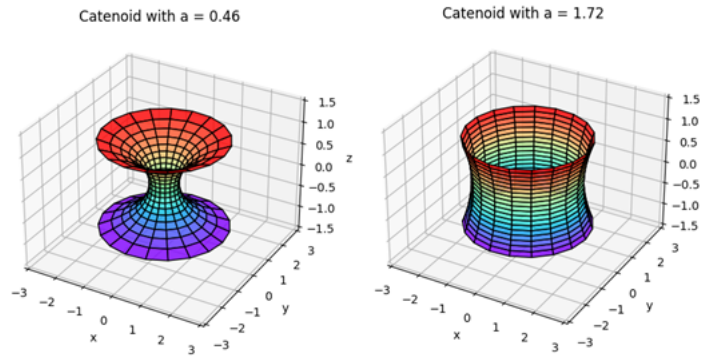


Figure 18.29. An Adjustable Catenoid

18.9.1 The Smallest Waist. Fig. 18.30 is a version of the soap-film catenoid in Fig. 18.28, but with various dimensions labeled. We've also reassigned the main axis to be the z-axis.

As the rings are pulled apart (i.e. H increases), the 'neck' gets longer and the 'waist' gets thinner (i.e. a decreases). However, at a critical distance, which depends on the ring's radius R, the soap film breaks, leaving planar films on each ring.

The catenoid's radius at any point along the z-axis, can be expressed using the equation for a catenary:

$$r(z) = a\cosh\left(\frac{z}{a}\right),$$

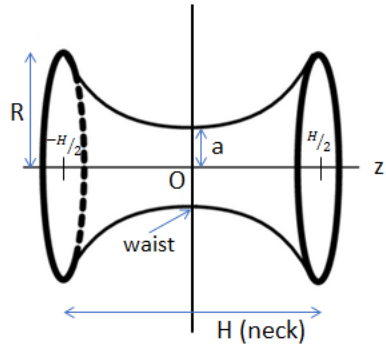


Figure 18.30. The Dimensions of a Catenoid

where a is the minimum neck radius at $z = 0$.

At either ring, say the one at $z = H/2$, the radius must equal the ring radius:

$$r(H/2) = a\cosh(\frac{H}{2a}) = R.$$

This equation allows us to determine a value for a when the values for R and H are known. Unfortunately, the equation is transcendental, so numerically methods (e.g. Brent's method) have to be employed to solve it.

It's also useful to plot the curve, which is simpler if we rearrange it. Multiply through by $2/H$:

$$\frac{2a}{H}\cosh(\frac{H}{2a}) = \frac{2R}{H}$$

and let $t = H/2a$, so that:

$$Y = (2R/H) = \cosh(t)/t$$

coshPlot.py plots $Y = cosh(t)/t$ for positive t values (see Fig. 18.31), and finds the minimum Y using SciPy's `brentq()` root finder (`https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.brentq.html`).

The graph in Fig. 18.31 tells us that there are two soap films that satisfy $Y = 2R/H > 1.5089...$, and nothing for $2R/H < 1.5089$. This is easily demonstrated by starting with a stable soap film, and gradually increasing the H separation between the rings. The rings don't change in size, so R is constant, which means that has H gets bigger that $2R/H$ decreases. At the minimum point, the film collapses into two.

When $2R/H > 1.5089...$, two catenoid shapes are possible – the common, 'stable' version with a thin neck that corresponds to the curve to the left of the minimum point. There's also a small chance of obtaining a bulging catenoid with

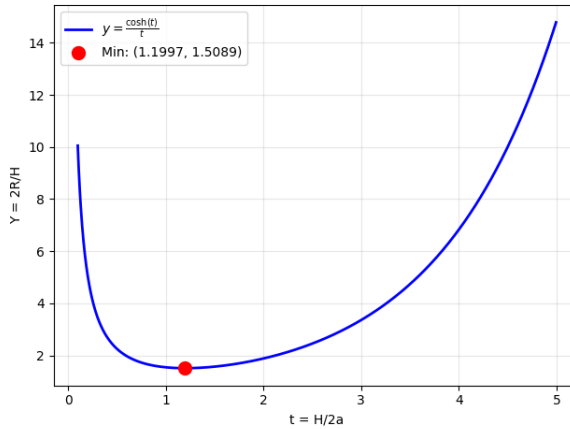


Figure 18.31. Plot of $Y = \cosh(t)/t$

a thick neck, which is unstable and likely to switch to the slim form at any time. This corresponds to the curve to the right of the minimum.

In soap-film demonstrations, R is usually fixed, so its useful to rephrase H and a in terms of R. The minimum point (t^*, Y^*) is

$$t^* = \frac{H}{2a} \approx 1.997, \quad \text{and} \quad Y^* = \frac{2R}{H} \approx 1.5089$$

Let's define H_c, the *critical* neck length, in terms of R:

$$\begin{aligned} 2R/H &= \cosh(t)/t \\ H_c &= (2Rt^*)/\cosh(t^*) \approx (2 \cdot 1.1997)R/\cosh(1.1997) \approx 1.325R \end{aligned}$$

So the critical neck length is about $\frac{4}{3}$ of the ring's radius. Any larger, and the catenoid will become unstable.

In terms of R, the critical waist *diameter*, $2a_c$, is

$$\begin{aligned} H/2a &= t^* \\ 2a_c &= H_c/t^* \approx 1.3255R/1.1997 \approx 1.105R \end{aligned}$$

The critical waist diameter is about the same as the ring radius. Any smaller and the catenoid will become unstable.

18.9.2 A Minimal Surface. The minimal surface property for a catenoid means that the concave surface of our soap film has a smaller area any other shape stretched between the two rings, such as a cylindrical film.

In 1744, Euler obtained a differential equation for determining the minimum area contained by a fixed boundary, although his analysis was later simplified by Lagrange. The Euler-Lagrange equation assumes that the problem surfaces are

continuous and have continuous derivatives. Sadly, most problems don't satisfy these criteria, although our soap film can be solved, but the details are a little advanced requiring concepts from the Calculus of Variations. One fairly approachable derivation can be found in Appendix III of Isenberg's *The Science of Soap Films and Soap Bubbles* [**Ise92**].

18.9.3 Comparing Surface Areas. Although we won't prove that the catenoid possesses the smallest surface area between two rings, we can calculate its area, and compare it to the surface area of a cylinder.

For a radius curve $r(z)$ rotated around the z-axis like the one in Fig. 18.28, the surface area between $z = -H/2$ and $z = H/2$ is:

$$S = 2\pi \int_{-H/2}^{H/2} r(z)\sqrt{1 + \left(\frac{dr}{dz}\right)^2}\, dz$$

$r(z) = a\cosh\left(\frac{z}{a}\right)$ is differentiated:

$$\frac{dr}{dz} = a \cdot \frac{1}{a}\sinh\left(\frac{z}{a}\right) = \sinh\left(\frac{z}{a}\right)$$

which means that:

$$\sqrt{1 + \left(\frac{dr}{dz}\right)^2} = \sqrt{1 + \sinh^2\left(\frac{z}{a}\right)}$$

Using $\cosh^2(x) - \sinh^2(x) = 1$:

$$\sqrt{1 + \sinh^2\left(\frac{z}{a}\right)} = \cosh\left(\frac{z}{a}\right)$$

Plug this and $r(z)$ into the surface area equation:

$$\begin{aligned} S &= 2\pi \textstyle\int_{-H/2}^{H/2} a\cosh\left(\frac{z}{a}\right) \cdot \cosh\left(\frac{z}{a}\right) dz \\ &= 2\pi a \textstyle\int_{-H/2}^{H/2} \cosh^2\left(\frac{z}{a}\right) dz \end{aligned}$$

Using the identity: $\cosh^2(x) = (1 + \cosh(2x))/2$:

$$\begin{aligned} S &= 2\pi a \textstyle\int_{-H/2}^{H/2} \frac{1+\cosh\left(\frac{2z}{a}\right)}{2}\, dz \\ &= \pi a \textstyle\int_{-H/2}^{H/2} \left[1 + \cosh\left(\frac{2z}{a}\right)\right] dz \\ &= \pi a \left[z + \frac{a}{2}\sinh\left(\frac{2z}{a}\right)\right]_{-H/2}^{H/2} \end{aligned}$$

Evaluate this expression between $-H/2$ and $H/2$:

$$S = \pi a\left[\left(\frac{H}{2} + \frac{a}{2}\sinh\left(\frac{H}{a}\right)\right) - \left(-\frac{H}{2} + \frac{a}{2}\sinh\left(-\frac{H}{a}\right)\right)\right]$$

Since $\sinh(-x) = -\sinh(x)$:

$$\begin{aligned} S &= \pi a\left[\frac{H}{2} + \frac{a}{2}\sinh\left(\frac{H}{a}\right) + \frac{H}{2} + \frac{a}{2}\sinh\left(\frac{H}{a}\right)\right] \\ &= \pi a\left[H + a\sinh\left(\frac{H}{a}\right)\right] \end{aligned}$$

areas.py utilizes this expression in `catenoidArea()` after first using SciPy's Brent's method to find a.

```
def catenoidArea(R, H):
  # Solve for catenoid parameter a using brentq()
  z = H/2
  aMin = R * math.exp(-z/R)
  a = brentq(catenoidEq, aMin, R, args=(R, z))
  return math.pi * a * (H + a * math.sinh(H / a))


def catenoidEq(a, R, z):
  return a * math.cosh(z/a) - R
```

These functions are used to generate a sequence of catenoid surface areas, with R fixed at 1, and H varying between 0.1 and 1.32548 (recall that the critical neck length $H_c < 1.325R$, or the soap film will break).

The catenoid's surface area is plotted against a cylinder's area (calculated using $2\pi RH$) in Fig. 18.32. For small H's, the two shapes have a similar surface area, but the catenoid's area becomes smaller as the distance between the rings increases.

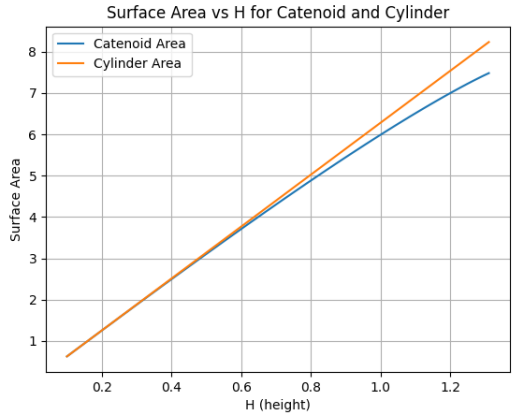


Figure 18.32. Surface Area vs. H for a Catenoid and Cylinder

Exercises

(1) A uniform chain of length $2l$ and weight w per unit length is suspended between two points at the same level and has a maximum sag of d. Prove that the tension at the lowest point is

$$w\frac{l^2-d^2}{2d}.$$

If $l = 50$ m and $d = 20$m, find the distance between the points of suspension.

(2) A uniform chain of length $2l$ has its ends fixed at two points at the same level. The sag at the middle is d. Prove that the span H is

$$\frac{l^2-d^2}{d}\log\frac{l+d}{l-d}.$$

(3) A chain is suspended from two points at the same level and $2l$ apart. The lowest point of the chain is d below the level of the supports. In each half of the chain, the load per unit horizontal distance varies uniformly with the horizontal distance from zero at the lowest point to w at the ends.

Taking the origin at the lowest point, show that the equation of the curve for the chain is

$$y = \frac{dx^3}{l^3}.$$

(4) Find the equation of the tractrix by evaluating the integral

$$-\int \frac{\sqrt{a^2-x^2}}{x}\,dx.$$

There are at least three substitutions which will work: $t^2 = a^2 - x^2$, $x = a\sin(\theta)$, and $x = a\cos(\theta)$.

Do the problem all three ways. Which of these substitutions do you think provides the easiest solution?

(5) In section 18.4, we discussed three equations that are often used to find the a value for a catenary:

(a) $d = a\left(\cosh\left(\frac{H}{2a}\right) - 1\right)$

(b) $L = 2a\sinh\left(\frac{H}{2a}\right)$

(c) $a = \frac{L^2/4-d^2}{2d}$

Derive these equations using the dimensions defined in Fig. 18.9, the catenary equation $y = a\cosh(x/a)$, and other identities from section 18.1.2.

(6) In section 18.6, it is stated without proof that a parabola $y = ax^2$ will have its focus at $(0, \frac{1}{4a})$. Show this to be true by utilizing Fig. 18.12.

(7) In section 18.8, it is stated that the corner of a square wheel fits snugly into the valley between two inverted catenaries. This implies that the angle each catenary makes with the x-axis is $\pi/4$. Show this to be true.

(8) Investigate the statement made near the end of section 18.8 that any polygonal wheel except a triangular one will roll without difficulty over a catenary.

(9) The description of the output of `rollingSquare.py` in section 18.8 mentions that if the distance traveled is $y = f(x)$, then $d\theta/dx = -1/f(x)$. Why is this true?

Answers

(1) For the catenary in Fig. 18.33:

$$y^2 = c^2 + s^2.$$

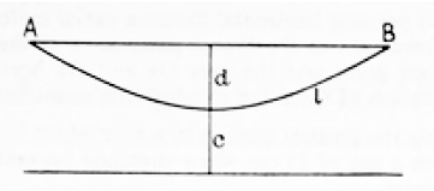


Figure 18.33. The Catenary in Q.1

At point B:

$$y = c + d, \qquad s = l,$$

so

$$(c + d)^2 = c^2 + l^2.$$

Thus,

$$2cd = l^2 - d^2, \qquad c = \frac{l^2 - d^2}{2d}.$$

The tension at the lowest point is therefore

$$wc = w\frac{l^2 - d^2}{2d}.$$

If $l = 50$ and $d = 20$:

$$c = \frac{2500 - 400}{40} = \frac{105}{2}.$$

Now, since

$$s = c\sinh\left(\frac{x}{c}\right),$$

we have if $AB = 2x$,

$$x = \frac{105}{2}\sinh^{-1}\left(\frac{20}{21}\right) = \frac{105}{2}\log\left[\frac{20}{21} + \sqrt{1 + \left(\frac{20}{21}\right)^2}\right].$$

Hence,

$$AB = 105\log\frac{49}{21} = 105 \times 2.303\log_{10}\frac{49}{21} \approx 89 \text{ m}.$$

(5) Deriving Equations:

(a) We'll show how $d = a(\cosh(H/(2a)) - 1)$ applies to a catenary by deriving $H = 2a\,\text{arcosh}((a + d)/a)$, and then rearranging its terms.
Consider the catenary in Fig. 18.9. Its lowest point is $(0, a)$ at $x = 0$, and its attachment points at P_1 and P_2 are at $x = \pm H/2$, height $= a + d$.
At the attachment point $x = H/2$, the height is $y = a + d$:

$$a + d = a\cosh(H/2a)$$

Divide both sides by a:

$$\frac{a + d}{a} = \cosh(H/2a))$$

Take the inverse hyperbolic cosine of both sides:

$$\text{arcosh}\left(\frac{a + d}{a}\right) = \frac{H}{2a}$$

Multiply both sides by $2a$:

$$H = 2a\,\text{arcosh}\left(\frac{a + d}{a}\right)$$

Rearrange the equation so the left side is in terms of d

$$d = a(\cosh(H/(2a)) - 1)$$

(b) $L = 2a\sinh\left(\frac{H}{2a}\right)$. Once again referring to the catenary in Fig. 18.9. The left attachment point P_1 is at $x = -H/2$ and P_2 is $x = H/2$. The arc length of a catenary from its vertex to a point P is $s = a\sinh\frac{x}{a}$, so the length between two points, with $x = x_1$ and $x = x_2$ is:

$$a\left[\sinh(x_2/a) - \sinh(x_1/a)\right]$$

But, we want the total arc length L from $x = -H/2$ to $x = H/2$:

$$L = a\left[\sinh(H/(2a)) - \sinh(-H/(2a))\right]$$

Use the property: $\sinh(-u) = -\sinh(u)$

$$\begin{aligned} L &= a\left[\sinh(H/(2a)) - (-\sinh(H/(2a)))\right] \\ &= a\left[\sinh(H/(2a)) + \sinh(H/(2a))\right] \\ &= 2a\sinh(H/(2a)) \end{aligned}$$

For small sags (large a relative to H), $\sinh(H/(2a)) \approx H/(2a)$, giving $L \approx H$ (nearly a straight line). For large sags (small a), sinh() grows exponentially, reflecting the much longer cable needed

(c) $a = (L^2/4 - d^2)/2d$. Back to Fig. 18.9 one more time. The attachment points, P_1 and P_2 are located at $(-H/2, a+d)$ and $(H/2, a+d)$ respectively. Plugging the latter into the catenary equation:

$$a + d = a\cosh(H/2a), \qquad \text{or} \qquad \cosh(H/2a) = 1 + \frac{d}{a}$$

The total arc length from $x = -H/2$ to $x = H/2$ is:

$$L = 2a\sinh(H/2a), \qquad \text{or} \qquad \sinh(H/2a) = L/(2a)$$

Due to the hyperbolic identity $\cosh^2(u) - \sinh^2(u) = 1$, this means that $\cosh^2(H/2a) - \sinh^2(H/2a) = 1$, and so:

$$\left(1 + \frac{d}{a}\right)^2 - \left(\frac{L}{2a}\right)^2 = 1$$

Expand and simplify;

$$\begin{aligned}
1 + \frac{2d}{a} + \frac{d^2}{a^2} - \frac{L^2}{4a^2} &= 1 \\
\frac{2d}{a} + \frac{d^2}{a^2} - \frac{L^2}{4a^2} &= 0 \\
2da + d^2 - \frac{L^2}{4} &= 0 \\
2da &= \frac{L^2}{4} - d^2 \\
a &= (L^2/4 - d^2)/2d
\end{aligned}$$

19

The Cycloid

Why Study the Cycloid?

- **As a source of interesting problems**. It's the solution to the brachistochrone problem (curve of fastest descent), the tautochrone problem (isochronous motion), and can be used to square the circle.
- **Historical significance**. It played a pivotal role in the development of modern mathematics through the work of Galileo, Huygens, Bernoulli, Euler, and others. Most notably, the solution of the brachistochrone problem 'kicked off' the development of the calculus of variations.
- **Geometric richness**. The cycloid pushed mathematicians to refine tools for cusps, curvature, arc length, involutes, and differentiability.
- **Exact solvability in mechanics**. Motion along a cycloid can be treated analytically, providing rare closed-form results in classical dynamics.

19.1 The Cycloid

A cycloid is the curve generated by a point on the circumference of a circle rolling along a straight line [**Sim92, GH97**]. A cycloid whose generating circle has radius a is defined by the parametric equations

$$x = a(t - \sin t), \qquad y = a(1 - \cos t) \tag{19.1}$$

where t is the circle's angle of rotation, which starts at $t = 0$ at the origin. The equations can be easily obtained by considering the dimensions labeled in Fig. 19.1.

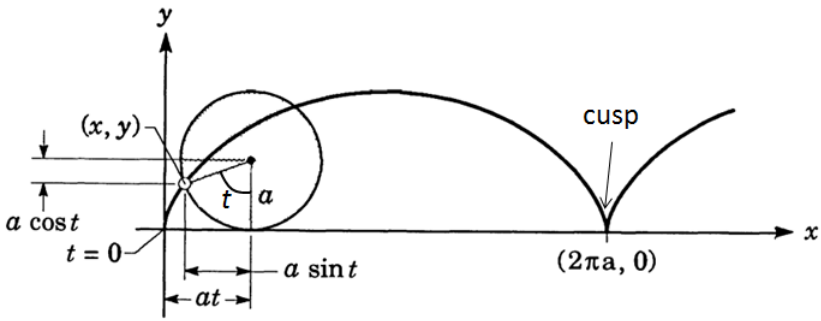


Figure 19.1. A Rolling Circle forms a Cycloid

From Equs. (19.1), we obtain the curve's derivative

$$y' = \frac{dy}{dx} = \frac{a \sin t \, dt}{a(1 - \cos t)\, dt} = \frac{\sin t}{1 - \cos t} = \frac{2 \sin \frac{1}{2}t \, \cos \frac{1}{2}t}{2 \sin^2 \frac{1}{2}t} = \cot \tfrac{1}{2}t.$$

Note that y' isn't defined for $t = 0, \pm 2\pi, \pm 4\pi$, etc, and those rotations correspond to *cusp*s where the cycloid touches the x-axis, and its tangent becomes vertical.

Rather surprisingly, the area under one arch of a cycloid (e.g. between $x = 0$ and $2\pi a$) is three times the area of the rolling circle:

$$A = \int_0^{2\pi a} y \, dx = \int_0^{2\pi} y \frac{dx}{dt} \, dt = \int_0^{2\pi} a(1 - \cos t)\, a(1 - \cos t)\, dt$$
$$= a^2 \int_0^{2\pi} (1 - \cos t)^2 \, dt = a^2 \int_0^{2\pi} (1 - 2\cos t + \cos^2 t)\, dt$$

So we need to evaluate:

$$\int_0^{2\pi} (1 - 2\cos t + \cos^2 t)\, dt$$

Split this into three parts:

$$\int_0^{2\pi} 1 \, dt - 2 \int_0^{2\pi} \cos t \, dt + \int_0^{2\pi} \cos^2 t \, dt$$

Consider each part in turn:

$$\int_0^{2\pi} 1 \, dt = 2\pi$$

and

$$\int_0^{2\pi} \cos t \, dt = [\sin t]_0^{2\pi} = \sin(2\pi) - \sin(0) = 0$$

For the third part, use the power reduction formula, $\cos^2 t = \frac{1+\cos(2t)}{2}$:

$$\int_0^{2\pi} \cos^2 t \, dt = \int_0^{2\pi} \frac{1+\cos(2t)}{2} \, dt$$

$$= \frac{1}{2} \cdot 2\pi + \frac{1}{2}\left[\frac{\sin(2t)}{2}\right]_0^{2\pi}$$

$$= \pi + 0 = \pi$$

Combining the results:

$$\int_0^{2\pi} (1 - 2\cos t + \cos^2 t) \, dt = 2\pi - 2(0) + \pi = 3\pi$$

The complete answer:

$$a^2 \int_0^{2\pi} (1 - \cos t)^2 \, dt = \mathbf{3\pi a^2}$$

This result was first approximated by Galileo in 1599 through the comparison of the weights of circle and cycloid models. It was proved in 1634 by the French mathematician Roberval.

Another unusual result is that the length of one arch of the cycloid is four times the diameter of the rolling circle. Since $dx = a(1 - \cos t)dt$ and $dy = a \sin dt$, the arc length ds is given by

$$\begin{aligned} ds^2 = dx^2 + dy^2 &= a^2\left[(1 - \cos t)^2 + \sin^2 t\right] dt^2 \\ &= 2a^2[1 - \cos t] \, dt^2 = 4a^2 \sin^2 \tfrac{1}{2}t \, dt^2, \\ ds &= 2a \sin \tfrac{1}{2}t \, dt. \end{aligned}$$

Therefore, the length of one arch (let's use $t = 0$ to 2π) is

$$L = \int_0^{2\pi} 2a \sin \tfrac{1}{2}t \, dt = \left[-4a \cos \tfrac{1}{2}t\right]_0^{2\pi} = \mathbf{8a}.$$

This result was first noted in 1658 by the English architect Christopher Wren.

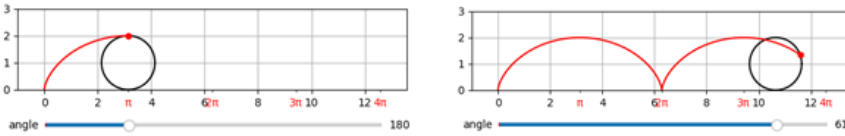


Figure 19.2. Rolling a Circle

19.1.1 Rolling a Circle. rollCycloid.py lets the user adjust the rotation angle of a circle, and the cycloid is drawn from the origin to a trace point located on the circle (see Fig. 19.2).

The circle's radius is $a = 1$ which explains why the cycloid cusps occur at 2π and 4π, and it peaks at $y = 2$.

The animation is updated by `update()` which translates the circle along the x-axis, and plots the cycloid:

```
def update(val):
  theta = math.radians(tSlider.val)
  Circle.set_xdata( xsCircleFn(theta))  # translate the circle
  ts = linspace(0, theta, max(2, int(50 * theta)))
  xs, ys = zip( *[cycloid(t, a) for t in ts])
  Cycloid.set_data(xs, ys)
  x, y = cycloid(theta, a)
  Point.set_data([x], [y])  # the tracer dot on the circle
  fig.canvas.draw_idle()

def cycloid(t, a):
  x = a * (t - math.sin(t))
  y = a * (1 - math.cos(t))
  return (x, y)
```

19.1.2 Examining Area and Arc Length. cycloidA.py lets the user change the rolling circle's radius (a), and draws the resulting cycloid between $t = 0$ and 27 radians. It calculates the area and arc length of a single arch of the curve in two ways – using the expressions $3\pi a^2$ and $8a$, and via numerical integration with Simpson's rule. Two screenshots are shown in Fig. 19.3.

The areas are produced by `calcArea()`:

```
def calcArea(a):
  analyticalArea = 3 * math.pi * a * a

  # For one arch, t goes from 0 to 2 pi
  ts = [i * (2*math.pi)/(nPts-1) for i in range(nPts)]
  xs, ys = cycloid(ts, a)
  # We need dx values for numerical integration
```

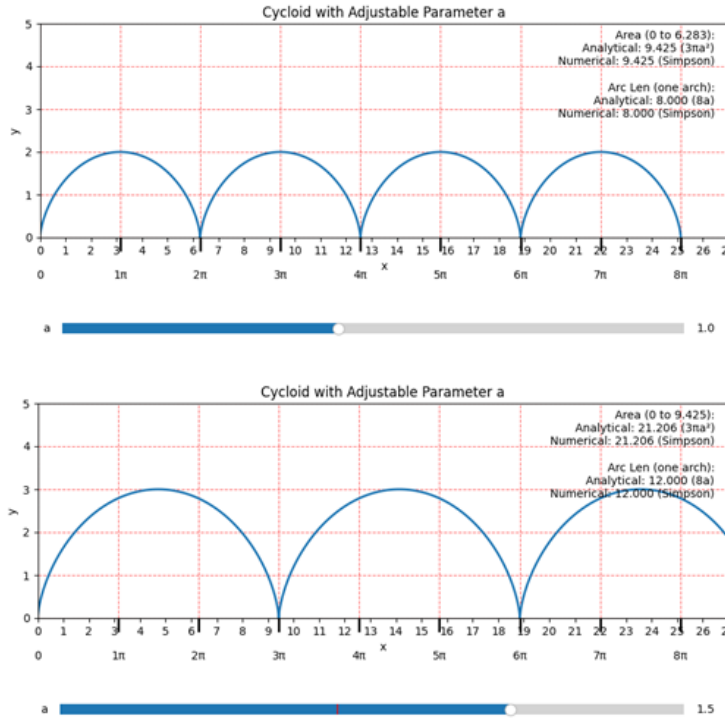


Figure 19.3. Cycloid Areas and arc lengths for Different Radii

```
  dxVals = [xs[i+1] - xs[i] for i in range(len(xs)-1)]
  # integrate y * (dx/dt) * dt, where dx/dt = a(1 - cos(t))
  dxdt = [a * (1 - math.cos(t)) for t in ts]
  integrand = [ys[i] * dxdt[i] for i in range(len(ys))]
  dt = (2 * math.pi) / (nPts - 1)
  numericalArea = simpson(integrand, dx=dt)
  return analyticalArea, numericalArea
```

Rather than code Simpson's rule ourselves (see section 6.11.4), we've imported `simpson()` from the SciPy module. It employs `integrand` to obtain the area under the curve as a collection of thin rectangular strips of dimension $y \times dx/dt$.

The arc lengths are calculated by `calcArcLen()`:

```
def calcArcLen(a):
  # arc length of one arch
  analyticalLen = 8 * a
```

```
  ts = [i * (2*math.pi)/(nPts-1) for i in range(nPts)]
  # Arc length numerical integral
  # Use cycloidDf() to get values for the derivatives
  integrand = []
  for t in ts:
    dx_dt, dy_dt = cycloidDf(t, a)
    integrand.append(math.sqrt(dx_dt**2 + dy_dt**2))
  dt = (2 * math.pi) / (nPts - 1)
  numericalLen = simpson(integrand, dx=dt)
  return analyticalLen, numericalLen


def cycloidDf(t, a):
  # Derivative of the cycloid
  dx_dt = a * (1 - math.cos(t))
  dy_dt = a * math.sin(t)
  return (dx_dt, dy_dt)
```

The numerical arc length is obtained by summing multiple small steps along the curve using $\sqrt{(dx/dt)^2 + (dy/dt)^2}$.

19.1.3 Treating the Cycloid Like a Function. Since the cycloid is defined parametrically:

$$x = a(t - \sin t), \qquad y = a(1 - \cos t)$$

there's no explicit Cartesian form $y = f(x)$. Instead, we recover t from the x equation by root finding, and plug it into the y equation.

cycloidYT.py implements this approach, assuming that $a = 1$:

```
> python cycloidYT.py
Enter x value: 3.14159
y = 2.000000; t = 180.00 deg

> python cycloidYT.py
Enter x value: 1.570796
y = 1.673612; t = 132.35 deg

> python cycloidYT.py
Enter x value: 6.283185
y = 0.000075; t = 359.30 deg
```

Internally, the x equation is rearranged into zero form

$$a(t - \sin t) - x = 0$$

and SciPy's Brent's method is used to find the positive root for t. Brent is utilized instead of Newton–Raphson because it doesn't require derivatives and handles cycloid cusps where $dx/dt = 0$.

The relevant code is in `cycloidYTfromX()`:

```
def cycloidYTfromX(x, a):
  archIndex = math.floor(x / archWidth)
  if archIndex != 0:
    xArch = x - archIndex * archWidth
      # handles negative and large x's
  else:
    xArch = x
  if abs(xArch) < 1e-12 or abs(xArch - archWidth) < 1e-12:
    return 0, 0      # cusp handling
  else:
    tMin = 0
    tMax = 2*math.pi
    t = brentq(cycloidT, tMin, tMax, args=(xArch, a))
    y = a * (1 - math.cos(t))
    return y, t + archIndex*2*math.pi


def cycloidT(t, x, a):
  return a * (t - math.sin(t)) - x
```

19.2 The Tautochrone

A tautochrone is the curve for which the time taken by a bead sliding without friction in uniform gravity along the curve to its lowest point is independent of its starting point.

The curve was first studied by Huygens after he realized that a pendulum, which obviously swings in a circular arc, keeps different times depending on how far the pendulum is initially pulled away from the vertical. He investigated what would happen if the arc was changed to an inverted cycloid, and discovered that the pendulum bob would swing down from any starting point to the bottom in the same amount of time. In other words, an inverted cycloid is a tautochrone.

One way to formalize this is to turn Fig. 19.1 upside down, as in Fig. 19.4. This points the y-axis in the direction of the gravitational force but makes the downward y-coordinates positive, conveniently leaving the cycloid equations unchanged.

The kinetic energy of the bead is initially zero, since it's at rest. The work done by gravity in moving the bead from $(0, 0)$ to some point (x, y) is mgy, and this must equal the change in kinetic energy. That is,

$$mgy = \frac{1}{2}mv^2 - \frac{1}{2}m(0)^2.$$

Thus, the velocity of the bead when it reaches (x, y) is

$$v = \sqrt{2gy}.$$

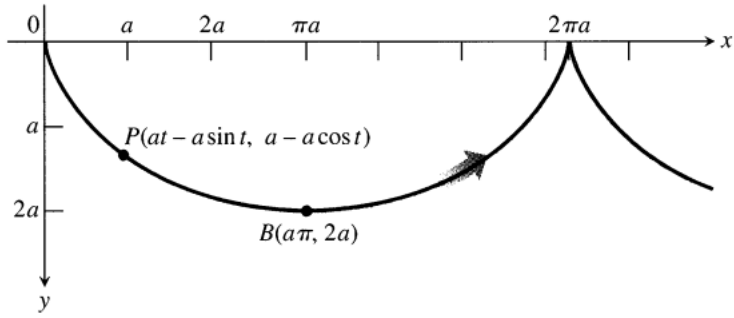


Figure 19.4. An Inverted Cycloid

That is,

$$\frac{ds}{dt} = \sqrt{2gy},$$

where ds is the arc length differential along the bead's path, or

$$dt = \frac{ds}{\sqrt{2gy}} = \frac{\sqrt{1 + (dy/dx)^2}\, dx}{\sqrt{2gy}}.$$

The time T_f it takes the bead to slide along a curve $y = f(x)$ from O to $B(a\pi, 2a)$ is

$$T_f = \int_{x=0}^{x=a\pi} \sqrt{\frac{1 + (dy/dx)^2}{2gy}}\, dx.$$

What curve $y = f(x)$, if any, minimizes the value of this integral? For the cycloid, T_f takes the form

$$T_{\text{cycloid}} = \int_{x=0}^{x=a\pi} \sqrt{\frac{dx^2 + dy^2}{2gy}}.$$

From Equs. (19.1):

$$dx = a(1 - \cos t)\, dt, \qquad dy = a \sin t\, dt,$$

and so

$$\begin{aligned} T_{\text{cycloid}} &= \int_{t=0}^{t=\pi} \sqrt{\frac{a^2(2 - 2\cos t)}{2ga(1 - \cos t)}}\, dt \\ &= \int_0^{\pi} \sqrt{\frac{a}{g}}\, dt = \pi\sqrt{\frac{a}{g}}. \end{aligned}$$

Thus, the amount of time it takes the friction-less bead to slide down the cycloid from O to the bottom at B is $\pi\sqrt{a/g}$.

Suppose that instead of O we start the bead at some lower point, (x_0, y_0). The bead's velocity at (x, y) is now

$$v = \sqrt{2g(y - y_0)} = \sqrt{2ga(\cos t_0 - \cos t)}$$

since $y = a(1 - \cos t)$. Accordingly, the time required for the bead to slide from (x_0, y_0) down to B is

$$\begin{aligned} T &= \int_{t_0}^{\pi} \sqrt{\frac{a^2(2 - 2\cos t)}{2ga(\cos t_0 - \cos t)}}\, dt = \sqrt{\frac{a}{g}} \int_{t_0}^{\pi} \sqrt{\frac{1 - \cos t}{\cos t_0 - \cos t}}\, dt \\ &= \sqrt{\frac{a}{g}} \int_{t_0}^{\pi} \sqrt{\frac{2\sin^2(t/2)}{(2\cos^2(t_0/2) - 1) - (2\cos^2(t/2) - 1)}}\, dt \\ &= \sqrt{\frac{a}{g}} \int_{t_0}^{\pi} \frac{\sin(t/2)\, dt}{\sqrt{\cos^2(t_0/2) - \cos^2(t/2)}} \end{aligned}$$

Apply the substitution $u = \cos(t/2)$ (and $-2\,du = \sin(t/2)\,dt$), and set $c = \cos(t_0/2)$:

$$T = \sqrt{\frac{a}{g}} \int_{t_0}^{\pi} \frac{-2\,du}{\sqrt{c^2 - u^2}}$$

Integrating:

$$\begin{aligned} T &= 2\sqrt{\frac{a}{g}} \left[-\sin^{-1} \frac{u}{c} \right]_{t_0}^{\pi} \\ &= 2\sqrt{\frac{a}{g}} \left[-\sin^{-1} \frac{\cos(t/2)}{\cos(t_0/2)} \right]_{t_0}^{\pi} \\ &= 2\sqrt{\frac{a}{g}} (-\sin^{-1} 0 + \sin^{-1} 1) = \pi\sqrt{\frac{a}{g}}. \end{aligned}$$

This is the same amount of time that it took the bead to slide from O to B, so beads starting simultaneously from different points on the inverted cycloid in Fig. 19.4 will all reach the bottom of the curve at B concurrently.

19.2.1 Simple Harmonic Motion (SHM). T is the descent time from O to the bottom of the cycloid at B. This implies that the oscillation time (or period), from O to the other extreme, and back to O, is $4 * T = 4\pi\sqrt{a/g}$, which is often written as

$$T_{\text{osc}} = 2\pi\sqrt{\frac{4a}{g}}.$$

In general, any motion that is simple harmonic relates its period T_p to its angular frequency ω by

$$T_p = \frac{2\pi}{\omega}.$$

So the cycloid exhibits SHM, and we can equate

$$\omega = \sqrt{\frac{g}{4a}}.$$

19.2.2 Animating the Cycloid as a Tautochrone. SHM is also characterized by the differential equation:

$$\frac{d^2s}{dt^2} + \omega^2 s = 0,$$

where s is the object's displacement from its equilibrium position. With initial conditions $s(0) = s_0$ and $s'(0) = 0$, this equation has the solution:

$$s(t) = s_0 \cos \omega t$$

This gives us a simple way to implement the movement of multiple beads down an inverted cycloid in tautochrone.py, as shown in Fig. 19.5.

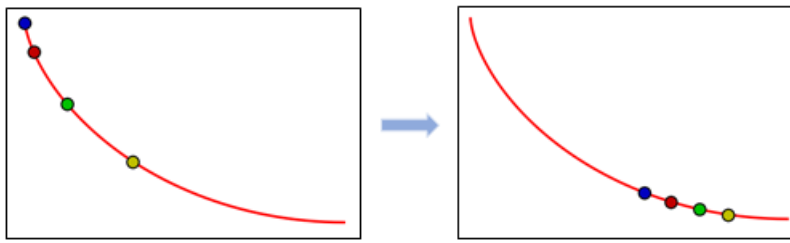


Figure 19.5. A Cycloid as a Tautochrone

Each bead is initialized with a starting t value and color.

```
beads = [
  {'theta':1.0, 'color':'#0000c0'},
  {'theta':0.8, 'color':'#c00000'},
  {'theta':0.6, 'color':'#00c000'},
  {'theta':0.4, 'color':'#c0c000'}
]
```

Inside `animate()`, which is called for each animation frame, each bead's 'theta' is combined with the current `w` to calculate its (x, y) position.

```
def animate(fno):
  w = fno/(nFrames-1)  # 0 to 1
  for i, bead in enumerate(beads):
    theta = -bead['theta']*math.pi * math.cos(w * math.pi/2)
    xc, yc = cycloid(theta, a)
    x = h/2*xc + x0
```

```
    y = h/2*yc + y0
    # (x0,y0) is the top point; h is the window height
    beadPts[i].set_data([x], [y])
  return beadPts
```

19.3 A Cycloid's Involute

Once Huygens had established the cycloid's tautochrone property, how could he arrange for a clock's pendulum to move along a cycloidal, rather than a circular, path? One solution is to suspend the pendulum from the cusp between two inverted cycloids with its rope's length equal to the cycloid's semi-arch (see Fig. 19.6). This constrains the pendulum's path to follow a cycloid.

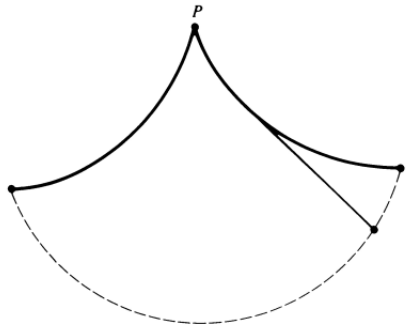


Figure 19.6. An Cycloidal Pendulum

Huygens termed this curve an *involute*: the locus of the end of a taut string as it unwraps itself from a cycloid surface (or wraps itself over such as surface). The equation for the involute is obtained as follows. Start with the parametric form of the surface curve

$$\mathbf{C}(t) = \big(x(t),\, y(t)\big),$$

where $\mathbf{C}'(t) \neq \mathbf{0}$, and define its arc length between a starting angle t_0 and t:

$$s(t) = \int_{t_0}^{t} \|\mathbf{C}'(u)\|\, du,$$

The unit tangent vector is

$$\mathbf{T}(t) = \frac{\mathbf{C}'(t)}{\|\mathbf{C}'(t)\|},$$

which allows the involute to be expressed as

$$\mathbf{I}(t) = \mathbf{C}(t) - s(t)\,\mathbf{T}(t).$$

This is depicted in Fig. 19.7. The end of the (purple) string initially touches the surface curve $\mathbf{C}()$ at $x = \pi a$, the peak of the cycloid's arch, and is attached at $x = 2\pi a$, in the cusp between two arches. Currently, the string has unwrapped so it's tangent is at angle t, and the tangent segment's length is $s(t)$. The involute traced out by the end of the string is $\mathbf{I}()$ (the red curve).

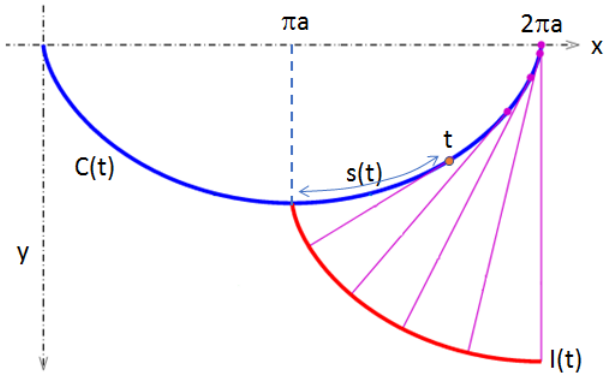


Figure 19.7. An Involute of a Cycloid

We can plug cycloid-specific details into $\mathbf{I}(t)$:

$$\mathbf{C}(t) = \big(a(t - \sin t),\ a(1 - \cos t)\big), \qquad \mathbf{C}'(t) = a\big(1 - \cos t,\ \sin t\big),$$

and

$$\|\mathbf{C}'(t)\| = 2a \sin \frac{t}{2}.$$

The arc length of the unwrapped string, $s(t)$, from $t = \pi$ to the current angle, is

$$\begin{aligned} s(t) &= \int_{\pi}^{t} 2a \sin \frac{u}{2}\, du = 2a\Big[-2\cos \frac{u}{2}\Big]_{u=\pi}^{u=t} \\ &= -4a \cos \frac{t}{2} \ - \ \big(-4a \cos \frac{\pi}{2}\big) = -4a \cos \frac{t}{2}. \end{aligned}$$

The unit tangent is

$$\mathbf{T}(t) = \frac{\mathbf{C}'(t)}{\|\mathbf{C}'(t)\|} = \frac{a(1 - \cos t, \sin t)}{2a \sin \frac{1}{2}} = \big(\sin \frac{t}{2},\ \cos \frac{t}{2}\big),$$

by utilizing half angles identities to rewrite the numerator.

We formulate $\mathbf{I}(t) = \mathbf{C}(t) - s(t)\mathbf{T}(t)$, by considering its x and y parts separately:

$$I_x(t) = a\big(t - \sin t\big) - \big(-4a\cos\frac{t}{2}\big)\sin\frac{t}{2} = a\big(t - \sin t\big) + 4a\cos\frac{t}{2}\sin\frac{t}{2}$$
$$= a\big(t - \sin t\big) + 2a\sin t = a\big(t + \sin t\big),$$

$$I_y(t) = a\big(1 - \cos t\big) - \big(-4a\cos\frac{t}{2}\big)\cos\frac{t}{2} = a\big(1 - \cos t\big) + 4a\cos^2\frac{t}{2}$$
$$= a\big(1 - \cos t\big) + 2a\big(1 + \cos t\big) = a\big(3 + \cos t\big).$$

So:

$$\mathbf{I}(t) = \big(a(t + \sin t),\ a(3 + \cos t)\big)$$

How does this compare to the surface cycloid equation, $\mathbf{C}(t)$, with t replaced by ϕ:

$$\mathbf{C}(\phi) = \big(a(\phi - \sin\phi),\ a(1 - \cos\phi)\big).$$

Set $\phi = t + \pi$, and simplify $\mathbf{C}()$:

$$\mathbf{C}(t + \pi) = a\big((t + \pi) - \sin(t + \pi),\ 1 - \cos(t + \pi)\big)$$
$$= a\big(t + \pi + \sin t,\ 1 + \cos t\big).$$

Equate the components of $\mathbf{I}()$ and $\mathbf{C}()$:

$$I_x(t) = a(t + \sin t) = a(t + \pi + \sin t) - a\pi = C_x(t + \pi) - a\pi,$$

$$I_y(t) = a(3 + \cos t) = a(1 + \cos t) + 2a = C_y(t + \pi) + 2a$$

or,

$$\mathbf{I}(t) = \mathbf{C}(t + \pi) + (-a\pi,\ 2a)$$

This shows that the involute $\mathbf{I}(t)$ is a cycloid, with its t angle is rotated by π and translated by $(-a\pi,\ 2a)$) away from $\mathbf{C}(t)$.

19.3.1 Drawing a Cycloid Involute. involuteCycs.py unwraps two strings from the crests of adjacent cycloids towards a center line above the intermediate cusp (see Fig. 19.8). As the strings move, line segments are drawn at their current tangents with lengths equal to the arc distance from the crests to those tangents. The coordinates of the ends of the moving segments are collected and plotted, forming two halves of a new cycloid.

During the animation, `update()` draws each frame by calling `tangentSeg()` twice to get the coordinates for the ends of the segments. To explain `tangentSeg()`, we'll focus on how (xEnd, yEnd) is calculated for the left-hand string, as shown in Fig. 19.9.

It's a little hard to see but `tangentSeg()` implements a version of

$$\mathbf{I}(t) = \mathbf{C}(t) - s(t)\,\mathbf{T}(t)\ :$$

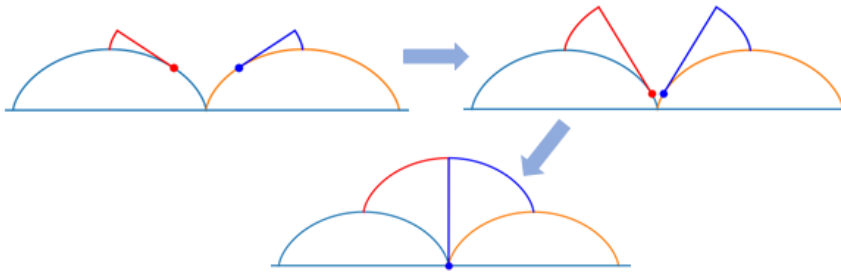


Figure 19.8. Drawing a Cycloid Involute

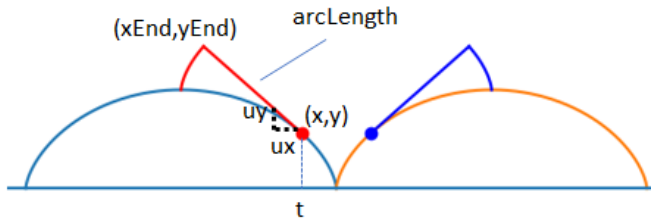


Figure 19.9. Calculating a Segment End Coordinate

```
def tangentSeg(t, a, xOffset, dir):
  x, y = cycloid(t, a, xOffset)    # C(t)
  dxdt, dydt = cycloidDf(t, a)     # C'(t)
  speed = math.hypot(dxdt, dydt)   # |C'(t)|
  if speed == 0.0:
    return [x, x], [y, y], x, y

  ux = dxdt / speed     # T(t) for x and y
  uy = dydt / speed
  if dir < 0:
    ux = -ux
    uy = -uy
  arcLength = 4.0 * a * abs(math.cos(t / 2.0))  # s(t)
  xEnd = x + arcLength * ux    # I(t) for x and y
  yEnd = y + arcLength * uy
  return [x, xEnd], [y, yEnd], xEnd, yEnd
```

$\mathbf{C}(t)$ is implemented by `cycloid()`, while $\mathbf{T}(t) = \mathbf{C}'(t)/\|\mathbf{C}'(t)\|$ employs `cycloidDf()` and math.hypot(). $s(t)$, the arc length, must be $4a$ or less since the unwinding

string only extends half-way around the cycloid (between $x = \pi a$ and $2\pi a$ on the left hand side).

`xOffset` and `dir` are used to generalize the function so it can be employed for the string wrapped over the right-hand cycloid. The relevant lines in `update()` are:

```
lx, ly, lxEnd, lyEnd = tangentSeg(tLeft, a, 0.0, dir=-1)
rx, ry, rxEnd, ryEnd = tangentSeg(tRight, a, 2.0*math.pi, dir=1)
```

19.4 The Brachistochrone

In 1696, Johann Bernoulli conceived and solved the brachistochrone problem, and published the problem (but not the solution) as a challenge to other mathematicians. The problem is this: among all smooth curves in a vertical plane that join a point P_0 to a lower point P_1, not directly below it, find the curve along which a particle will slide from P_0 to P_1 in the shortest possible time.

Following Huygens' approach, we can think of the particle as a bead of mass m sliding down a friction-less wire, with mg the only force acting upon it. Locate P_0 at the origin and set $P_1 = (x_1, y_1)$ as in Fig. 19.10. Note that P_1 does not need to be the bottom of the cycloid (i.e. point B in Fig. 19.4).

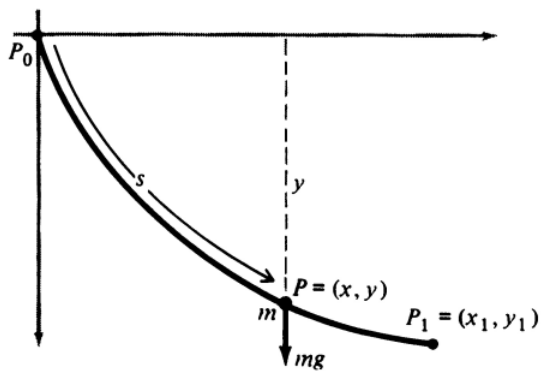


Figure 19.10. A Brachistochrone Bead at P

The bead is released from rest at P_0, so its initial velocity and kinetic energy are zero. The work done by gravity in pulling it down to an arbitrary point $P = (x, y)$ is mgy, which must equal the increase in its kinetic energy. So $\frac{1}{2}mv^2 = mgy$, and therefore

$$v = \frac{ds}{dt} = \sqrt{2gy}.$$

This can be written as

$$dt = \frac{ds}{\sqrt{2gy}} = \frac{\sqrt{dx^2 + dy^2}}{\sqrt{2gy}} = \frac{\sqrt{1 + (dy/dx)^2}\, dx}{\sqrt{2gy}}.$$

The total time T_1 required for the bead to slide down the wire from P_0 to P_1, will depend on the wire's shape as specified by $y = f(x)$, so that

$$T_1 = \int dt = \int_0^{x_1} \sqrt{\frac{1 + (y')^2}{2gy}}\, dx.$$

The brachistochrone problem therefore amounts to finding a curve $y = f(x)$ that passes through P_0 and P_1 that minimizes this integral.

We start by considering an apparently unrelated problem in optics. Fig. 19.11 illustrates a ray of light traveling from A to P with constant velocity v_1. Upon entering a denser (gray) medium, it travels from P to B with a smaller velocity v_2.

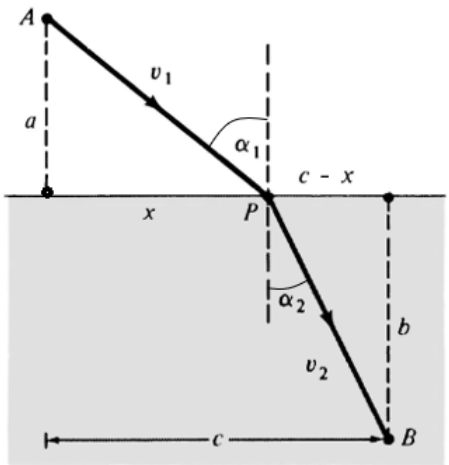


Figure 19.11. The Refraction of Light

The total time T required for the journey is

$$T = \frac{\sqrt{a^2 + x^2}}{v_1} + \frac{\sqrt{b^2 + (c - x)^2}}{v_2}.$$

If we assume that the ray of light is able to select its path from A to B in such a way as to minimize T, then $dT/dx = 0$. Then the two parts of that derivative can be equated:

$$\frac{x}{v_1\sqrt{a^2 + x^2}} = \frac{c - x}{v_2\sqrt{b^2 + (c - x)^2}},$$

By referring to Fig. 19.11, the distances ratios can be replaced by sines involving α_1 and α_2:

$$\frac{\sin\alpha_1}{v_1} = \frac{\sin\alpha_2}{v_2}.$$

This is *Snell's law of refraction*, and the assumption that light travels from one point to another along the path requiring the shortest time is *Fermat's principle of least time*.

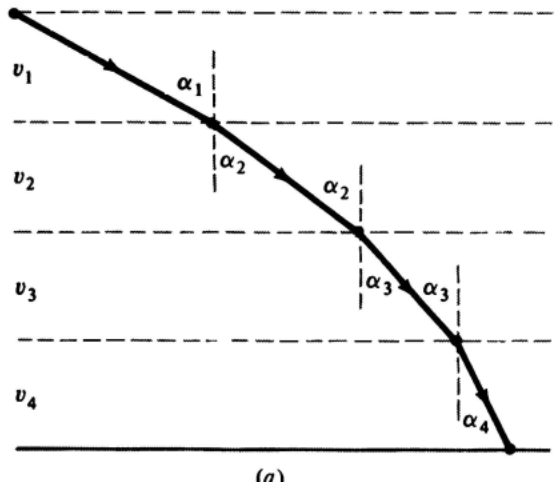


Figure 19.12. Refraction Angles

The velocity of light is constant within a layer in Fig. 19.12, but decreases as it passes down through each subsequent denser layer, and is refracted more and more towards the vertical. When Snell's law is applied at the boundaries between the layers, we obtain

$$\frac{\sin\alpha_1}{v_1} = \frac{\sin\alpha_2}{v_2} = \frac{\sin\alpha_3}{v_3} = \frac{\sin\alpha_4}{v_4}.$$

As we make these layers thinner and more numerous, in the limit the velocity of light decreases continuously as the ray descends, and we get

$$\frac{\sin\alpha}{v} = c$$

Returning to the brachistochrone problem, let's introduce the coordinate scheme shown in Fig. 19.13 and assume that the bead (like the ray of light) is capable of selecting the path from P_0 to P_1 that requires the shortest possible travel

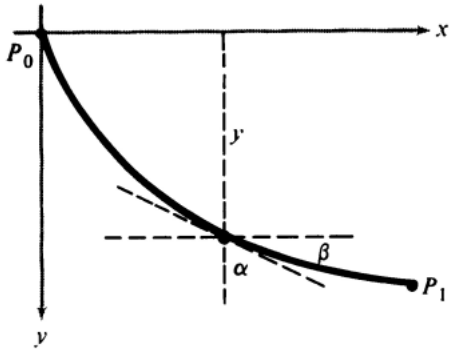


Figure 19.13. Refraction for the Brachistochrone

time. The argument given above yields the same result for the sliding bead

$$\frac{\sin\alpha}{v} = c$$

We also have

$$\sin\alpha = \cos\beta = \frac{1}{\sec\beta} = \frac{1}{\sqrt{1+\tan^2\beta}}$$

$$= \frac{1}{\sqrt{1+(y')^2}}.$$

Combining these equations, and the earlier velocity equality $v = \sqrt{2gy}$, produces

$$y[1+(y')^2] = k.$$

Replace y' by dy/dx, and separate the variables

$$dx = \sqrt{\frac{y}{k-y}}\,dy,$$

so

$$x = \int \sqrt{\frac{y}{k-y}}\,dy.$$

Employ the substitution $u^2 = y/(k-y)$ so that

$$y = \frac{ku^2}{1+u^2} \quad \text{and} \quad dy = \frac{2ku}{(1+u^2)^2}\,du,$$

resulting in

$$x = \int \frac{2ku^2}{(1+u^2)^2}\,du.$$

The trigonometric substitution $u = \tan\phi$, and its derivative $du = \sec^2\phi\, d\phi$, let us write this as

$$\begin{aligned} x &= \int \frac{2k\,\tan^2\phi\,\sec^2\phi}{(1+\tan^2\phi)^2}\,d\phi \\ &= 2k\int \frac{\tan^2\phi}{\sec^2\phi}\,d\phi = 2k\int \sin^2\phi\,d\phi \\ &= k\int (1-\cos 2\phi)\,d\phi \\ &= \frac{1}{2}k(2\phi - \sin 2\phi). \end{aligned}$$

The constant of integration is zero because $y = 0$ when $\phi = 0$, and since P_0 is at the origin, we also have $x = 0$ when $\phi = 0$.

The formula for y is

$$y = \frac{k\tan^2\phi}{\sec^2\phi} = k\sin^2\phi = \frac{1}{2}k(1-\cos 2\phi).$$

Simplifying these equations by writing $a = \frac{1}{2}k$ and $\theta = 2\phi$, leads to

$$x = a(\theta - \sin\theta), \quad y = a(1-\cos\theta).$$

In other words, the curve that produces the shortest travel time for the Brachistochrone problem is the cycloid

19.4.1 Animating the Brachistochrone. `brachAnim.py` visualizes and compares three different paths (a straight line, an arc of a circle, and a cycloid) for a bead sliding under gravity from point (0,0) to an endpoint.

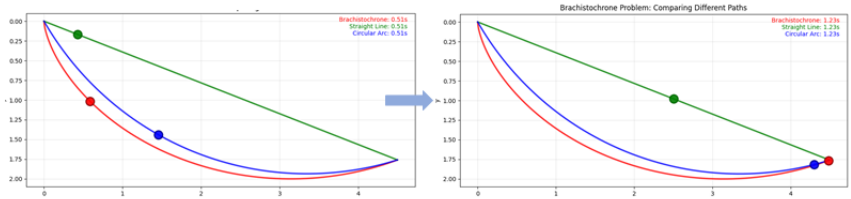


Figure 19.14. The Brachistochrone Race

The animation demonstrates that the cycloid, despite being longer than a circular arc between the same two points, causes the bead to reach the endpoint faster than both the straight line and arc. However, the cycloid bead only *just* beats the bead sliding along the arc, so it's not surprising that early investigators, such as Galileo, concluded that the arc was fastest.

It's possible to pause/resume the animation by pressing the space bar, which allows the user to compare the positions of the beads during their journey. One interesting aspect is that the circular bead (the blue dot) is ahead of the cycloid bead (the red dot) for much of the trip (see Fig. 19.14).

It's important to remember that `brachAnim.py` is not a real-world experiment (e.g. unlike the one at `https://mathsmodels.co.uk/2021/06/01/Branchistochrone2/`). For example, no account is made of friction or air resistance, and the course of the bead's path along the arc is only approximate because the exact time equation is an elliptic integral of the form

$$t(\theta) = \sqrt{\frac{R}{2g}} \int_{\theta_0}^{\theta} \frac{d\phi}{\sqrt{\cos\phi - \cos\theta_0}},$$

where R is the circle's radius, and the bead travels from angular position θ_0 to θ. There's no closed form solution for this expression, so numerical methods must be used. My code employs time = distance / velocity, where the total distance is the sum of small segments along the arc, and the current velocity is $\sqrt{2gh}$ where h is the vertical drop from the starting point. The details can be found in `calculateCircularArc()` in `brachAnim.py`.

Another issue is what curvature should be used for the arc since it's only constrained by two points. My code employs a fixed relationship to transform the chord length between the start and end points into the circle's radius.

The bead's journey time along the cycloid is much simpler; it's based on

$$T = \pi\sqrt{\frac{a}{g}},$$

resulting in:

```
_, thetaEnd = cycloidYTfromX(xEnd, a)
t = thetaEnd * math.sqrt(a / G)
brCurveX, brCurveY = cycloidPts(a, thetaEnd, 500)
```

The difficult part is determining the angle `thetaEnd` of the endpoint (which is not necessarily π) but I reuse `cycloidYTfromX()` from section 19.1.3 for that task.

19.5 Epicycloids

An epicycloid is a curve generated by a point on the circumference of a circle which is rolling on the *outside* of a fixed circle (see Fig. 19.15). In other words, it's the cycloid again but with the flat surface replaced by a circle [**Mao20**].

More complicated curves can be generated by having a third circle roll on the second, a fourth on the third, etc. Epicycloids were introduced by the ancient Greeks and, until the 16th century, formed the basis for ideas about the paths of the Moon and planets.

An epicycloid whose fixed and rolling circles have radii R and r respectively is described by the parametric equations:

$$x = (R+r)\cos t - r\cos(\frac{R+r}{r}t) \qquad y = (R+r)\sin t - r\sin(\frac{R+r}{r}t).$$

t is the angle formed by the x-axis and the ray from the center of the fixed circle to the point of contact with the rolling circle, as depicted in Fig. 19.15.

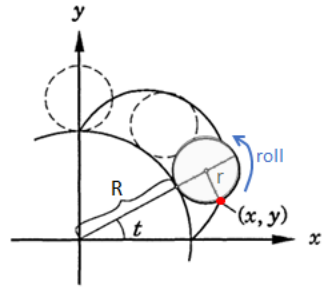


Figure 19.15. The Epicycloid Circles

If the radii of the fixed circle and rolling circle are equal ($R = r$) then a complete tour by the rolling circle around the fixed one will form a single arc. If $R = 2r$ then the tour will include two arcs, and when $R = nr$ the tour will contain n arcs. Between the arcs, the curve will form cusps when the rolling circle's tracer point touches the fixed circle.

epicycloid.py asks the user to enter a float representing R/r, which is used to calculate the radius of the small circle (r) since the large circle is a fixed size ($R = 4$).

The plotting uses a (verbose) translation of the maths:

```
def epicycloidPt(bigR, smallR, t):
  x = (bigR + smallR) * math.cos(t) \
               - smallR * math.cos((bigR + smallR)/smallR * t)
  y = (bigR + smallR) * math.sin(t) \
               - smallR * math.sin((bigR + smallR)/smallR * t)
  return x, y
```

Fig. 19.16 shows four runs of `epicycloid.py`, including a cardioid (one cusp; $R/r = 1$) and a nephroid (two cusps; $R/r = 2$).

19.5.1 From Epicycloid to Epicycle. The parametric equations can be better visualized by increasing the size of the fixed circle, as in Fig. 19.17, so the rotating tracer point (the red dot) on the circumference of the rolling circle can be specified relative to the circumference of the fixed circle. In this version, the smaller circle is usually called the *epicycle* and the larger circle the *deferent*.

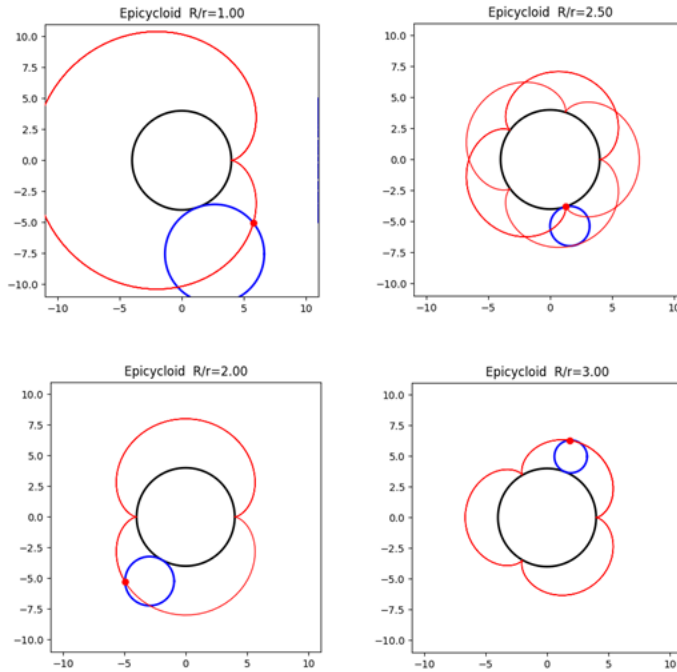


Figure 19.16. Four Epicycloids

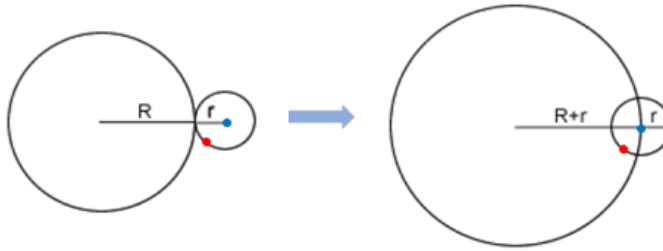


Figure 19.17. From Epicycloid to Epicycle

It's also useful to redefine the epicycle's equations in terms of angular speed. Let

$$\mathbf{r}(t) = (R+r)\begin{pmatrix}\cos t\\ \sin t\end{pmatrix} + r\begin{pmatrix}-\cos\left(\frac{R+r}{r}t\right)\\ -\sin\left(\frac{R+r}{r}t\right)\end{pmatrix}$$

or:

$$\mathbf{r}(t) = R_{\text{def}} \begin{pmatrix} \cos \omega t \\ \sin \omega t \end{pmatrix} + r_{\text{epi}} \begin{pmatrix} \cos(\Omega t) \\ \sin(\Omega t) \end{pmatrix}$$

where:

- R_{def} is the radius of the deferent,
- r_{epi} is the radius of the epicycle,
- ω is the angular speed of the deferent,
- Ω is the angular speed of the epicycle relative to the deferent,

The angular velocities are related by:

$$\frac{\Omega}{\omega} = -\frac{R + r}{r}$$

By adding more epicycles, with varying speeds and sizes, we can formulate increasingly complex curves:

$$x(t) = \sum_i R_i \cos(\omega_i t + \phi_i)$$
$$y(t) = \sum_i R_i \sin(\omega_i t + \phi_i)$$

ϕ_i is how much circle i is initially rotated, and is called a **phase offset**. If we don't utilize phase offsets, then we can only describe epicycles where all the circles start neatly aligned.

19.5.2 From Epicycles to Discrete Fourier Transforms. Instead of thinking about points along the x and y axes, imagine them as complex numbers using the real and imaginary axes:

$$p_j = x_j + iy_j$$
$$z(t) = x(t) + iy(t) = \sum_j^N R_j(\cos(\omega_j t + \phi_j) + i \sin(\omega_j t + \phi_j))$$

Now make use of Euler's formula:

$$z(t) = \sum_j^N R_j e^{i\omega_j t + \phi_j}$$

Even better, permit X_j be a complex number:

$$z(t) = \sum_j^N X_j e^{i\omega_j t}$$

This form almost matches the *Discrete Fourier Transform* (DFT) [**Kre10**]. This shouldn't be too surprising since DFTs essentially allows us to split any periodic function into a series of connected sinusoidal functions.

19.6 Hypocycloids

A hypocycloid is a curve generated by a point on the circumference of a circle rolling on the *inside* of a fixed circle [**Mao20**]. A hypocycloid whose fixed and rolling circles have the radii R and r respectively has the parametric equations

$$x = (R-r)\cos t - r\cos(\frac{R-r}{r}t) \qquad y = (R-r)\sin t - r\sin(\frac{R-r}{r}t).$$

Once again, t is the angle formed by the x-axis and the ray from the center of the fixed circle to the point of contact with the rolling circle, as in Fig. 19.18.

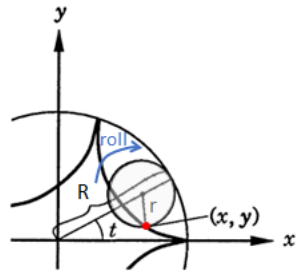


Figure 19.18. The Hypocycloid Circles

In a similar manner to the epicycloid, the hypocycloid has a cusp where its tracer point meets the fixed circle, and describes two arcs when the radius of the fixed circle is twice the length of the radius of the rolling circle ($R = 2r$), and n arcs when $R = nr$.

hypocycloid.py asks the user to enter a float representing R/r, which is used to calculate the radius of the small circle (r) since the large circle is a fixed size ($R = 5$).

The code based on the maths:

```
def hypocycloidPt(bigR, smallR, t):
  x = (bigR - smallR) * math.cos(t) \
      + smallR * math.cos((bigR - smallR) / smallR * t)
  y = (bigR - smallR) * math.sin(t) \
      - smallR * math.sin((bigR - smallR) / smallR * t)
  return x, y
```

Fig. 19.19 shows six runs of `hypocycloid.py`, and illustrates a few interesting cases: a straight line when $R/r = 2$, and a deltoid and astroid. Note that an astroid appears when $R/r = 4$ (as expected) *and* also when $R/r = 4/3$.

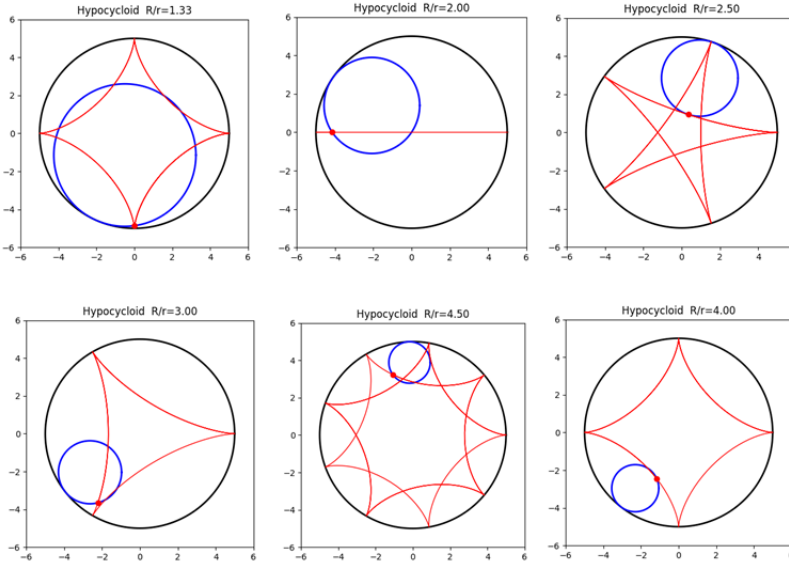


Figure 19.19. Six Hypocycloids

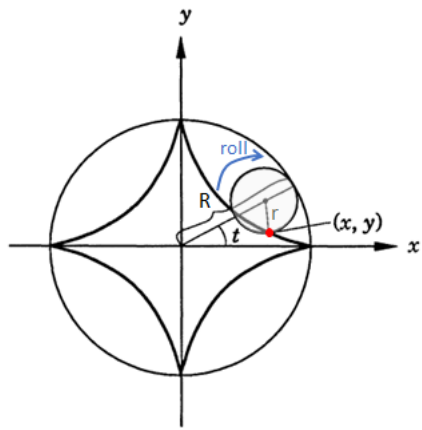


Figure 19.20. The Astroid

19.6.1 The Astroid. The astroid is a hypocycloid whose rolling circle has a diameter one-fourth that of the fixed circle (see Fig. 19.20).

The formula for the astroid sets $R = 4r$ in the equations for the hypocycloid:

$$x = (4r - r)\cos t + r\cos\left(\frac{4r - r}{r}t\right) = r(3\cos t + \cos 3t),$$
$$y = (4r - r)\sin t - r\sin\left(\frac{4r - r}{r}t\right) = r(3\sin t - \sin 3t).$$

With the help of

$$\cos 3t = 4\cos^3 t - 3\cos t; \quad \sin 3t = 3\sin t - 4\sin^3 t,$$

we find

$$x = r(3\cos t + \cos 3t) = 4r\cos^3 t$$
$$y = r(3\sin t - \sin 3t) = 4r\sin^3 t,$$

or

$$x = R\cos^3 t, \qquad y = R\sin^3 t.$$

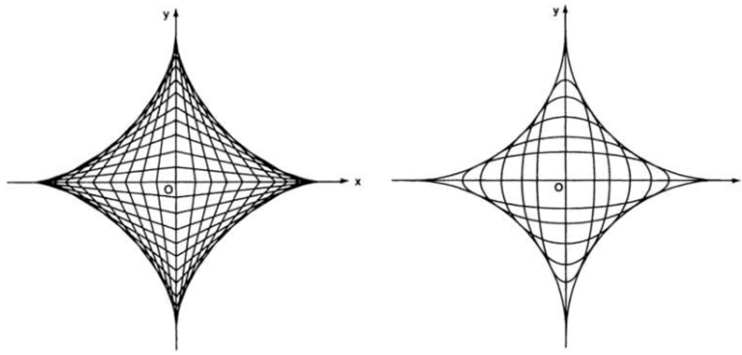
These equations are implemented in astroid.py as:

```
def astroid(a, nPts):
  ts = [2 * math.pi * i / nPts for i in range(nPts + 1)]
  xs = [a * (math.cos(t) ** 3) for t in ts]
  ys = [a * (math.sin(t) ** 3) for t in ts]
  return xs, ys
```

`astroid.py` generates the same shape as the one shown in the bottom right of Fig. 19.19. But what about the top-left example, where $R/r = 4/3$? This only changes the very last step of the equation's derivation given above since now $3R = 4r$. This affects the drawing behavior, but its only noticeable in the animation – the rolling circle for $R/r = 4/3$ takes 3 tours around the fixed circle to complete the astroid, whereas when $R/r = 4$ the much smaller rolling circle needs just a single tour. In general, a hypocycloid with ratio R/r (in reduced form) completes its pattern after the rolling circle has made r tours around the fixed circle.

The astroid has some remarkable properties. For example, all of its tangent lines are the same length R between the x and y axes. Conversely, if a line segment of fixed length R with its endpoints on the x- and y-axes is allowed to assume all possible positions, then the envelope formed by all of the line segments is an astroid (see Fig. 19.21(a)).

The astroid is also the envelope of the family of ellipses of the form $\frac{x^2}{a^2} + \frac{y^2}{(R-a)^2} = 1$, where the sum of their semi-major and semi-minor axes is R (see Fig. 19.21(b)).



(a) Using Tangent Lines (b) Using Ellipses

Figure 19.21. Drawing Astroids

19.7 Linking Hypocycloids and Epicycloids

The parametric equations for the hypocycloid and the epicycloid only differ in the sign associated with r. When the two radii are combined using $R + r$ then a epicycloid results. When its $R-r$, then an hypocycloid is created. The connection becomes clear is we think of the meaning of $-r$ in terms of the position of the rolling circle. A positive radius places it on the outside of the fixed circle while a negative moves it inside.

ehcycloid.py relaxes the constraints imposed by `epicycloid.py` and `hypocycloid.py` that R/r be positive, which makes it possible for one program to produce both types of cycloid. To keep the code short, only the cycloid curve is drawn without the circles or the animation (see Fig. 19.22).

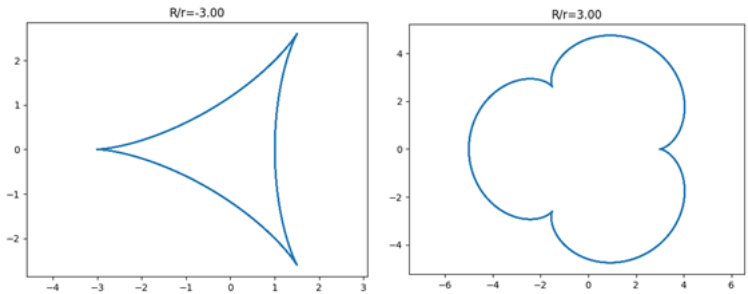


Figure 19.22. Hypocycloids and Epicycloids

19.8 Squaring the Circle

The problem of squaring the circle, namely constructing a square with the same area as a given circle using a straight edge and compass alone, is one of the classic problems of Greek mathematics. In 1882, Ferdinand von Lindemann proved that it wasn't possible if only those tools were available. However, it's quite straightforward if other mechanisms are permitted, such as a circle rolling along a line, to produce a cycloid as in Fig. 19.23.

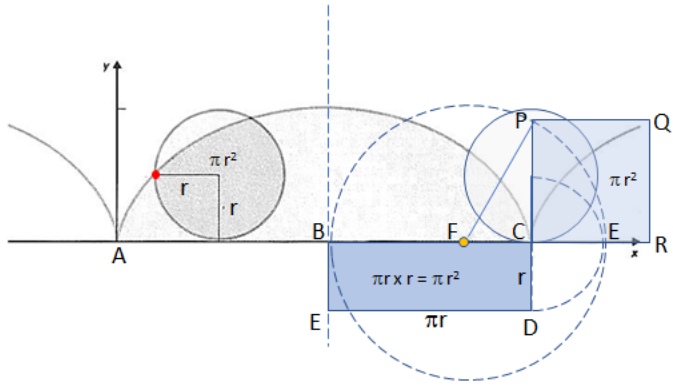


Figure 19.23. Squaring the Circle with a Cycloid

The (red) tracer point on the circle's circumference, moving from A to C describes a cycloid, meaning that the length of the straight line from A to C is equal to the circumference of the circle – that is, $2\pi r$.

Thus, if B is the midpoint of AC, then $BC = \pi r$. Hence, if $CD = r$, the area of the rectangle $BCDE$ is $\pi r \times r = \pi r^2$, which is also the area of the rolling circle. 'Squaring' this rectangle yields the square with the side PC. Thus the circle is also squared.

How is rectangle $BCDE$ created? Extend BC rightwards. Construct a circle with center C and radius r through D; it intersects BC at E. Find the midpoint F of BE. Construct a circle with diameter BE and center F. Extend the line CD until it meets that big circle at P. Construct a square on the segment CP, which we'll call $CPQR$.

How do we know that Area($BCDE$) = Area($CPQR$)? Consider:

$$\begin{aligned}
\text{Area}(CPQR) &= CP \cdot CR \\
&= CP^2 = FP^2 - FC^2 \\
&= (FP - FC)(FP + FC) \\
&= (FE - FC)(BF + FC) \quad \text{because } FP = FE = BF \\
&= CE \cdot BC = CD \cdot BC \\
&= \text{Area}(BCDE).
\end{aligned}$$

This means that the rolling circle, rectangle, and square in Fig. 19.23 all have the same area, πr^2, and we have successfully squared the circle.

Exercises

(1) Determine a $x = f(y)$ equation for the cycloid by eliminating θ from the parametric equations in Equs. 19.1.

(2) Show that the second derivative of the cycloid is $y'' = dy'/dx = -a/y^2$. Observe that this implies that the cycloid is concave down between the cusps.

(3) Show that the tangent to the cycloid at the point P in Fig. 19.24 passes through the top of the rolling circle.

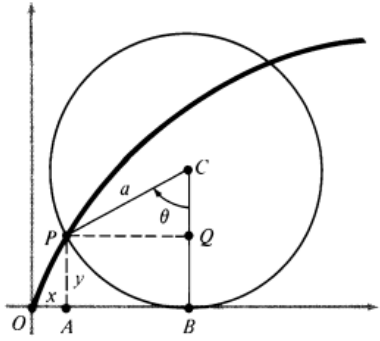


Figure 19.24. Cycloid Normal

(4) Assume that the circle in Fig. 19.24 rolls to the right along the x-axis at a constant speed, with the center C moving at v_0 units per second.

(a) Find the rates of change of the coordinates x and y of the point P.

(b) What is the greatest rate of increase of x, and where is P when this occurs?

(c) What is the greatest rate of increase of y, and for what value of θ is this attained?

(5) Find the area inside an astroid.

(6) Find the total length of an astroid.

(7) A hypocycloid with three cusps, $R = 3r$, is called a deltoid (see the lower left of Fig. 19.19). Find its parametric equations, and the total length of the curve.

Answers

(1) $P = (x, y)$ is $x = a(\theta - \sin\theta)$ and $y = a(1 - \cos\theta)$. Expressing θ and $\sin\theta$ in terms of y:

$$\cos\theta = 1 - \frac{y}{a}$$

$$\sin\theta = \sqrt{1 - \left(1 - \frac{y}{a}\right)^2}$$

$$= \sqrt{\frac{2y}{a} - \frac{y^2}{a^2}} = \frac{\sqrt{2ay - y^2}}{a}$$

$$\theta = \sin^{-1}\left(\frac{\sqrt{2ay - y^2}}{a}\right)$$

Substituting for θ and $\sin\theta$ in the expression for x:

$$x = a\left(\sin^{-1}\left(\frac{\sqrt{2ay - y^2}}{a}\right) - \frac{1}{a}\sqrt{2ay - y^2}\right)$$

$$a\sin^{-1}\left(\frac{\sqrt{2ay - y^2}}{a}\right) = x + \sqrt{2ay - y^2}$$

(2) The slope of the tangent to the cycloid: $y' = \cot\frac{\theta}{2}$.

$$\implies \frac{dy'}{dx} = \frac{d}{d\theta}(\cot\frac{\theta}{2})/\frac{dx}{d\theta} = -\frac{1}{2}\csc^2\frac{\theta}{2}/\frac{dx}{d\theta}$$

$$= \frac{-\frac{1}{2}\csc^2\frac{\theta}{2}}{a(1 - \cos\theta)} = \frac{-1/(2\sin^2\frac{\theta}{2})}{a(1 - \cos\theta)}$$

$$= \frac{-1/(1 - \cos\theta)}{a(1 - \cos\theta)} = -\frac{1}{a(1 - \cos\theta)^2}$$

$$= -\frac{a}{y^2} \quad \text{because } y = a(1 - \cos\theta)$$

As $y \geq 0$ throughout, then $y' < 0$ wherever $y \neq 0$, which is at the cusps.

(3) The point at the top of the circle has coordinates $(a\theta, 2a)$. The slope of the tangent at P is given by $y' = \cot \frac{1}{2}t$. The equation of the tangent at P is therefore

$$y - a(1 - \cos\theta) = \frac{\sin\theta}{1 - \cos\theta}(x - a\theta + a\sin\theta).$$

We substitute $x = a\theta$ in this equation and solve for y, which gives

$$y = a(1 - \cos\theta) + \frac{\sin\theta}{1 - \cos\theta} \cdot a\sin\theta = \frac{a(1 - \cos\theta)^2 + a\sin^2\theta}{1 - \cos\theta} = 2a.$$

This shows that the tangent at P does indeed pass through the point $(a\theta, 2a)$ at the top of the circle.

(4) (a) The rate of change of x and y can be expressed as:

$$\frac{dx}{dt} = v_0(1 - \cos\theta) \qquad \frac{dy}{dt} = v_0\sin\theta$$

where θ is the angle turned by C after time t.

Let the center of C be O. Without loss of generality, let P be at the origin at time $t = t_0$. By definition, P traces out a cycloid. $P = (x, y)$ is

$$x = a(\theta - \sin\theta), \qquad y = a(1 - \cos\theta)$$

Let (x_c, y_c) be the coordinates of O at time t. We have that $y_c = a$, which means that $x_c = v_0 t$. x_c is equal to the length of the arc of C that has rolled along the x-axis, so $x_c = a\theta$. So $\theta = \frac{v_0 t}{a}$, which implies $\frac{d\theta}{dt} = \frac{v_0}{a}$. Thus:

$$x = a\left(\frac{v_0 t}{a} - \sin\theta\right)$$

substituting for θ:

$$x = v_0 t - a\sin\theta$$

$$\begin{aligned} \implies \frac{dx}{dt} &= v_0 - a\cos\theta\frac{d\theta}{dt} = v_0 - a\cos\theta\frac{v_0}{a} \\ &= v_0(1 - \cos\theta) \end{aligned}$$

and:

$$y = a(1 - \cos\theta)$$

$$\begin{aligned} \implies \frac{dy}{dt} &= a\sin\theta\frac{d\theta}{dt} = a\sin\theta\frac{v_0}{a} \\ &= v_0\sin\theta \end{aligned}$$

(b) The maximum rate of change of x is $2v_0$, which happens when P is at the top of the circle C. The rate of change of x is given by:

$$\frac{dx}{dt} = v_0(1 - \cos\theta)$$

This is a maximum when $1 - \cos\theta$ is a maximum, so when $\cos\theta$ is at a minimum. That happens when $\cos\theta = -1$, so $\theta = \pi, 3\pi, \ldots$. That is, when $\theta = (2n+1)\pi$ where $n \in \mathbb{Z}$. That is, when P is at the top of the circle C. When $\cos\theta = -1$ we have:

$$\frac{dx}{dt} = v_0(1 - (-1)) = 2v_0$$

(c) The maximum rate of change of y is v_0, which happens when $\theta = \frac{\pi}{2} + 2n\pi$ where $n \in \mathbb{Z}$. The rate of change of y is given by $\frac{dy}{dt} = v_0 \sin\theta$.
This is a maximum when $\sin\theta$ is a maximum, when $\sin\theta = 1$. That happens when $\theta = \frac{\pi}{2} + 2n\pi$ where $n \in \mathbb{Z}$. When $\sin\theta = 1$ we have $\frac{dy}{dt} = v_0$.

(5) We'll show that the area inside an astroid constructed within a circle of radius a (see Fig. 19.25) is $A = \frac{3\pi a^2}{8}$. Locate the astroid H with its center at the origin and its cusps positioned on the axes.

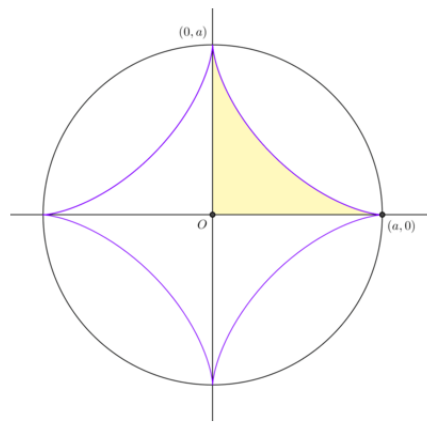


Figure 19.25. Astroid Area

By symmetry, it's sufficient to evaluate the area shaded yellow and to multiply it by 4. The astroid equation is:

$$x = a\cos^3\theta, \qquad y = a\sin^3\theta.$$

Thus:

$$A = 4\int_0^a y\,dx \quad = 4\int_{x=0}^{x=a} y\frac{dx}{d\theta}\,d\theta$$

Differentiate x with respect to θ:

$$\frac{dx}{d\theta} = 3a\cos^2\theta(-\sin\theta)$$

Substitute into A to give:

$$A = 4\int_{x=0}^{x=a} a\sin^3\theta \cdot 3a\cos^2\theta(-\sin\theta)\,d\theta$$

When $x = a$, $a\cos^3\theta = a$; when $x = 0$, $\theta = \frac{\pi}{2}$; when $x = a$, $\theta = 0$. Simplifying

$$\begin{aligned} A &= 4\int_{\theta=0}^{\theta=\pi/2} a\sin^3\theta \cdot 3a\cos^2\theta(-\sin\theta)\,d\theta \\ &= 12a^2\int_0^{\pi/2} \sin^4\theta\cos^2\theta\,d\theta \end{aligned}$$

Simplifying the integrand:

$$\begin{aligned} \sin^4\theta\cos^2\theta = \frac{(2\sin\theta\cos\theta)^2}{4}\sin^2\theta &= \frac{\sin^2 2\theta}{4}\sin^2\theta \\ &= \frac{\sin^2 2\theta}{4}\cdot\frac{1-\cos 2\theta}{2} \\ &= \frac{\sin^2 2\theta - \sin^2 2\theta\cos 2\theta}{8} \\ &= \frac{1-\cos 4\theta}{16} - \frac{\sin^2 2\theta\cos 2\theta}{8} \end{aligned}$$

Thus:

$$
\begin{aligned}
A &= 12a^2 \int_0^{\pi/2} \left(\frac{1-\cos 4\theta}{16} - \frac{\sin^2 2\theta \cos 2\theta}{8} \right) d\theta \\
&= \frac{3a^2}{4} \int_0^{\pi/2} (1-\cos 4\theta)\, d\theta - \frac{3a^2}{2} \int_0^{\pi/2} \sin^2 2\theta \cos 2\theta \, d\theta \\
&= \frac{3a^2}{4} \left[\theta - \frac{\sin 4\theta}{4} \right]_0^{\pi/2} - \frac{3a^2}{2} \int_0^{\pi/2} \sin^2 2\theta \cos 2\theta \, d\theta \\
&= \frac{3a^2}{4} \left[\theta - \frac{\sin 4\theta}{4} \right]_0^{\pi/2} - \frac{3a^2}{2} \left[\frac{\sin^3 2\theta}{6} \right]_0^{\pi/2} \\
&= \frac{3a^2}{4} \left(\frac{\pi}{2} - \frac{\sin 2\pi}{4} \right) - \frac{3a^2}{2} \cdot \frac{\sin^3 \pi}{6} \\
&= \frac{3\pi a^2}{8} - \frac{3a^2}{16} \sin 2\pi - \frac{3a^2}{12} \sin^3 \pi = \mathbf{\frac{3\pi a^2}{8}}
\end{aligned}
$$

(6) We'll show that the total length of the four arcs of an astroid constructed within a deferent of radius a (see Fig. 19.26) is $L = 6a$. Once again the astroid H has its center at the origin and its cusps positioned on the axes.

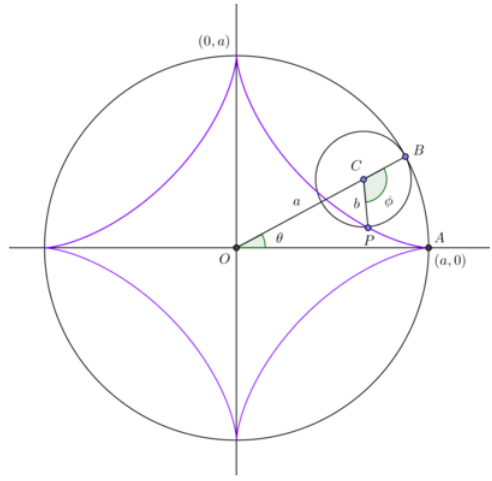


Figure 19.26. Length of an Arc of an Astroid

L is 4 times the length of one arc of the astroid, and the arc length is defined as:

$$L = 4\int_{\theta=0}^{\theta=\pi/2} \sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2}\, d\theta$$

where, from the equation for the astroid:

$$x = a\cos^3\theta, \qquad y = a\sin^3\theta$$

we have:

$$\frac{dx}{d\theta} = -3a\cos^2\theta\sin\theta, \qquad \frac{dy}{d\theta} = 3a\sin^2\theta\cos\theta$$

Thus:

$$\begin{aligned}\sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2} &= \sqrt{9a^2(\sin^4\theta\cos^2\theta + \cos^4\theta\sin^2\theta)} \\ &= 3a\sqrt{\sin^2\theta\cos^2\theta(\sin^2\theta + \cos^2\theta)} \\ &= 3a\sqrt{\sin^2\theta\cos^2\theta} = 3a\sin\theta\cos\theta \\ &= \frac{3a\sin 2\theta}{2}\end{aligned}$$

Thus:

$$\begin{aligned}L &= 4\int_0^{\pi/2} \frac{3a}{2}\sin 2\theta\, d\theta \\ &= 6a\left[-\frac{\cos 2\theta}{2}\right]_0^{\pi/2} = 6a\left(-\frac{\cos\pi}{2} + \frac{\cos 0}{2}\right) \\ &= 6a\left(-\frac{(-1)}{2} + \frac{1}{2}\right) = \mathbf{6a}\end{aligned}$$

(7) By definition, a deltoid is a hypocycloid with 3 cusps.

Let H be the deltoid generated by the epicycle C_1 of radius b rolling without slipping around the inside of a deferent C_2 of radius $a = 3b$ (see Fig. 19.27). Let C_2 have its center located at the origin. Let P be a point on the circumference of C_1. Let C_1 be initially positioned so that P is its point of tangency to C_2, located at point $A = (a, 0)$ on the x-axis. Let (x, y) be the coordinates of P as it travels over the plane. The point $P = (x, y)$ is described by the parametric equation:

$$x = 2b\cos\theta + b\cos 2\theta, \qquad y = 2b\sin\theta - b\sin 2\theta$$

where θ is the angle between the x-axis and the line joining the origin to the center of C_1.

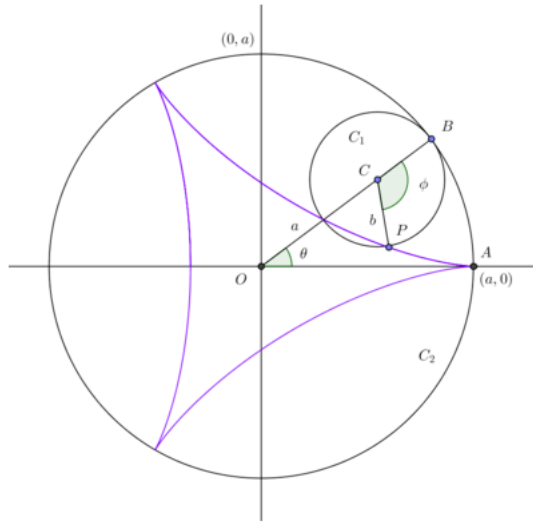


Figure 19.27. A Deltoid

Using the hypocycloid equation, H is given by:

$$x = (a-b)\cos\theta + b\cos\left(\frac{a-b}{b}\theta\right)$$
$$y = (a-b)\sin\theta - b\sin\left(\frac{a-b}{b}\theta\right)$$

This can be generated by an epicycle C_1 of radius $\frac{1}{3}$ the radius of the deferent. Thus $a = 3b$ and the equation of H is now given by:

$$x = 2b\cos\theta + b\cos 2\theta, \qquad y = 2b\sin\theta - b\sin 2\theta$$

We'll now show that the total length of the arcs of a deltoid constructed within a deferent of radius a is $L = \frac{16a}{3}$.

Let one of H's cusps be positioned at $(a, 0)$, and note that L is 3 times the length of one arc of the deltoid. The arc length equation is:

$$L = 3\int_{\theta=0}^{\theta=2\pi/3} \sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2}\, d\theta$$

where, the deltoid is:

$$x = 2b\cos\theta + b\cos 2\theta, \qquad y = 2b\sin\theta - b\sin 2\theta$$

and so:

$$\frac{dx}{d\theta} = -2b\sin\theta - 2b\sin 2\theta, \qquad \frac{dy}{d\theta} = 2b\cos\theta - 2b\cos 2\theta$$

Thus:

$$\begin{aligned}\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2 &= (-2b\sin\theta - 2b\sin 2\theta)^2 + (2b\cos\theta - 2b\cos 2\theta)^2 \\ &= 4b^2\left((-\sin\theta - \sin 2\theta)^2 + (\cos\theta - \cos 2\theta)^2\right) \\ &= 4b^2(\sin^2\theta + 2\sin\theta\sin 2\theta + \sin^2 2\theta + \cos^2\theta - 2\cos\theta\cos 2\theta + \cos^2 2\theta)\end{aligned}$$

Using the sum of squares for sine and cosine:

$$\begin{aligned}&= 4b^2(2 + 2\sin\theta\sin 2\theta - 2\cos\theta\cos 2\theta) \\ &= 8b^2(1 + \sin\theta\sin 2\theta - \cos\theta\cos 2\theta)\end{aligned}$$

and the double angle formula for sine

$$= 8b^2(1 + 2\sin^2\theta\cos\theta - \cos\theta\cos 2\theta)$$

and then the double angle formula for cosine:

$$= 8b^2(1 + 2\sin^2\theta\cos\theta - \cos\theta(1 - 2\sin^2\theta))$$

Simplifying

$$\begin{aligned}&= 8b^2(1 - \cos\theta + 4\sin^2\theta\cos\theta) \\ &= 8b^2(1 - \cos\theta + 4\cos\theta(1 - \cos^2\theta))\end{aligned}$$

Use the sum of squares for sine and cosine again, and the difference of two squares:

$$\begin{aligned}&= 8b^2(1 - \cos\theta + 4\cos\theta(1 + \cos\theta)(1 - \cos\theta)) \\ &= 8b^2(1 - \cos\theta)(1 + 4\cos\theta(1 + \cos\theta)) \\ &= 8b^2(1 - \cos\theta)(1 + 4\cos\theta + 4\cos^2\theta) \\ &= 8b^2(1 - \cos\theta)(1 + 2\cos\theta)^2\end{aligned}$$

Use the half angle formula for sine:

$$\begin{aligned}&= 8b^2\left(2\sin^2\frac{\theta}{2}\right)(1 + 2\cos\theta)^2 \\ &= 16b^2\sin^2\frac{\theta}{2}(1 + 2\cos\theta)^2\end{aligned}$$

Thus:

$$\sqrt{\left(\frac{dx}{d\theta}\right)^2 + \left(\frac{dy}{d\theta}\right)^2} = 4b\sin\frac{\theta}{2}|1 + 2\cos\theta|$$

In the range 0 to $\frac{2\pi}{3}$, $1 + 2\cos\theta$ is not less than 0, and so:

$$L = 3\int_0^{2\pi/3} 4b \sin\frac{\theta}{2}(1 + 2\cos\theta)\, d\theta$$

Put $u = \cos\frac{\theta}{2}$ so $2\frac{du}{d\theta} = -\sin\frac{\theta}{2}$. As θ increases from 0 to $\frac{2\pi}{3}$, u decreases from 1 to $\frac{1}{2}$. Then with the half angle formula for cosine

$$1 + 2\cos\theta = 1 + 2\left(2\cos^2\frac{\theta}{2} - 1\right) = 4u^2 - 1$$

Substituting $2\frac{du}{d\theta} = -\sin\frac{\theta}{2}$ and change the limits of integration from $\theta = 0$ – $\frac{2\pi}{3}$, to $u = 1 - u = \frac{1}{2}$, and after dealing with the sign:

$$\begin{aligned} L &= 12b\int_1^{1/2} (1 - 4u^2)(2)\, du \\ &= 24b\left[u - \frac{4}{3}u^3\right]_1^{1/2} \\ &= 24b\left(\left(\frac{1}{2} - \frac{4}{3}\cdot\frac{1}{8}\right) - \left(1 - \frac{4}{3}\right)\right) \\ &= 24b\left(\frac{2}{3}\right) = 16b = \mathbf{\frac{16a}{3}} \end{aligned}$$

20

Function Approximation

Why Study Function Approximation?

- **Solvability**. Many functions are too complex or too 'expensive' to compute in their original form; approximations offer a way to reexpress them in ways that employ simpler and faster arithmetic.

- **Accuracy vs. Efficiency**. Determining a function approximation involves a balance between precision and speed. There's no single right answer but approximations can be easily adjusted to investigate the trade-offs.

- **Range/Operation Restrictions**. The suitability of an approximation greatly depends on the range of data that it must handle, and upon hardware rounding errors.

- **Polynomial Accuracy**. Polynomial approximations, typified by Taylor series and the Lagrange and Chebyshev approximations, combine simple implementations with precision. The Taylor series works best with data close to its expansion point, and is more error-prone at greater distances. However, Lagrange and Chebyshev spread the approximation error evenly across the entire data range.

- **Modeling Singularities**. Rational approximations are a better choice than polynomials for handling curves containing sharp spikes, vertical asymptotes, or functions that flatten out at infinity.

20.1 Dirty Functions as Motivation

Function approximation investigates how complex functions can be approximated by polynomials. 'Complex' may mean that the original function doesn't have a known definition or that it's computationally costly to evaluate. Polynomials may be 'simpler' in the sense of having desirable properties such as continuity and known error bounds, or by being less expensive to calculate. An excellent textbook which covers this topic is *Numerical Analysis* by Burden *et. al* [**BFB16**]. One major area of function approximation, which we won't be discussing here (but will perhaps in the future) are Fourier series.

The second author's interest in this topic started while he was reading up on the early days of microcomputer BASICs. Many of them lacked math functions, and the first ones only offered integer arithmetic. With the arrival of floating-point capabilities, Dennis Allison (the designer of Tiny BASIC) saw the need for better math support, and published "Quick and Dirty Functions for BASIC" in *Dr. Dobb's Journal* in 1978 [**All78**].

The article briefly describes five short but efficient subroutines for $\sqrt{x}$, $\log_{10} x$, 10^x, $\arctan x$, and $\cos x$. The functions utilize various range reducing formulae, and most employ polynomial approximations borrowed from Hart's *Computer Approximations* text [**Har68**]. The approximations are accurate to around six decimal places, which was more than adequate enough for the microcomputer hardware of the time, and the underlying motivation was that multiplication/-division was expensive, available memory was tiny, and floating-point precision was limited. Low-order approximations or truncated power series were a valid alternative to the high-precision algorithms used on mainframes.

This highlights the question of the relevance of function approximation *to-day*, when computing power is plentiful and cheap. One answer can be seen in the current uses of languages similar to BASIC (e.g. MicroPython `https://micropython.org/`) as coding tools on micro-controllers such as the Arduino and low-end Raspberry Pi's. In addition, functions with no simple closed definition can still only be represented approximately.

Allison's article was republished in Volume 3 of the *Dr. Dobb's Journal* reprints, which is available at the Internet Archive at `https://archive.org/details/dr_dobbs_journal_vol_03/`, in issue No. 9. I've also saved a copy of the article locally in `dirtyAllison.pdf`.

`dirty.py` is our translation of Allison's BASIC into Python, with the addition of a test-rig that calls each function with different inputs and compares their results to Python's math library. The results are much as you'd expect – Allison's code is accurate to around 6 or 7 decimal places. In the next four subsections, we'll look in detail at Allison's $\sqrt{x}$, $\log_{10} x$, $\arctan x$, and $\cos x$.

20.2 Dirty Square Root

Allison's pseudo-code:

(1) Report $x < 0$ as an error.

(2) Reduce the input range:
$t = 1; repeat\{x = x/100; t = t * 10\}$ until $x \leq 1$.
By the end, $0 \leq x \leq 1$ and $x * t * t =$ the input number.

(3) Make an initial guess for $\sqrt{x}$ with

$$y_0 = 0.115442 + 1.15442x$$

(4) Iterate until $|y_{k+1} - y_k| < 10^{-6}$ using the Newton-Raphson method:

$$y_{k+1} = \frac{1}{2}\left(y_k + \frac{x}{y_k}\right)$$

(5) The final answer is: $y_{k+1} * t$

It's possible to replace (4) by a for-loop that loops 4 or 5 times since we know that each pass will roughly double the number of correct digits in the result.

The Python code:

```
def sqrtD(x):
  if x < 0:
    raise ValueError("Math domain error")
  if x == 0:
    return 0

  # Scale x down by powers of 100
  t = 1.0
  while x > 1:
    x = x / 100
    t *= 10

  # A linear guess at sqrt(x)
  y = 0.115442 + 1.15442*x

  # Newton-Raphson until convergence
  count = 0
  while True:
    oldY = y
    y = 0.5 * (y + (x / y))
    count += 1
    if abs(y - oldY) < 1e-6:
      break
  return y * t
```

The *Newton–Raphson* method starts with a function f, its derivative f', and an initial guess y_0 for a root of f. If f satisfies certain assumptions and the guess is close to the root, then

$$y_1 = y_0 - \frac{f(y_0)}{f'(y_0)}$$

will be a better approximation of the root than y_0. Geometrically, $(y_1, 0)$ is the intercept of the tangent to f at $(y_0, f(y_0))$, as depicted in Fig. 20.1.

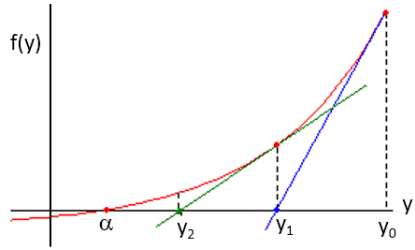


Figure 20.1. Newton-Raphson Approaching a Root at α.

The process repeatedly update y:

$$y_{k+1} = y_k - \frac{f(y_k)}{f'(y_k)}$$

until y_{k+1} is close enough to the root at α.

We use Newton-Raphson to approximate $y = \sqrt{x}$ by rephrasing it as a root-finding problem. Specify f as

$$f(y) = y^2 - x$$

Since $f'(y) = 2y$, the iterator becomes

$$\begin{aligned} y_{k+1} &= y_k - \frac{f(y_k)}{f'(y_k)} \\ &= y_k - \frac{y_k^2 - x}{2y_k} \\ &= \frac{1}{2}(y_k + \frac{x}{y_k}). \end{aligned}$$

The number of correct digits in y_k roughly doubles with each step. To see this, consider each y_k as the root we seek (α) plus some error term e_k, which is its distance from the root (e.g. as pictured in Fig. 20.1). Consider how the error changes from one iteration, $y_k = \alpha + e_k$, to the next, $y_{k+1} = \alpha + e_{k+1}$. These expressions, and the fact that $\alpha^2 = x$, can be substituted into the Newton-Raphson formula to give:

$$\alpha + e_{k+1} = \frac{1}{2}\left(\alpha + e_k + \frac{\alpha^2}{\alpha + e_k}\right).$$

After some rearranging, this becomes:

$$e_{k+1} = \frac{e_k^2}{2(\alpha + e_k)} = \frac{e_k^2}{2(\sqrt{x} + e_k)}.$$

When y_k gets close to the root α, the error term becomes small enough so that it can be omitted from the denominator, yielding a quadratic approximation

$$e_{k+1} \approx \frac{e_k^2}{2\sqrt{x}}.$$

Since the error is being *squared* in each iteration, the number of correct decimal digits doubles. For example, if y_k possesses k correct digits then $e_k \approx 10^{-k}$, and

$$e_{k+1} \approx \frac{(10^{-k})^2}{2\sqrt{x}} \approx 10^{-2k},$$

resulting in roughly $2k$ correct digits. The constant factor $1/(2\sqrt{x})$ affects this progression but will become negligible after the first few steps. This is confirmed by the results of `errDigits.py` described below.

Stage (2) of the pseudo-code reduces the initial error by constraining x between 0 and 1. The range-reduction loop keeps dividing x by 100 and multiplying a scale factor t by 10 until $x \leq 1$. The domain is therefore the interval $(1/100, 1]$.

Stage (3) starts the Newton-Raphson process with a value obtained from $y_0 = a + bx$. A good initial value should fix a and b in order to minimize the maximum relative error:

$$\epsilon(x) = \frac{(a + bx) - \sqrt{x}}{\sqrt{x}}.$$

20.2.0.1 Determining a and b. Deciding on suitable values for a and b is a *minimax* approximation problem.

The *Chebyshev equioscillation theorem* states that a minimax approximation of a curve of degree n is characterized by *at least* $n + 2$ points where the relative error $\epsilon(x)$ has a maximum magnitude E with alternating signs. So for our linear polynomial, we need at least three points to determine suitable values for a and b.

Three suitable points within $(1/100, 1]$ are the left boundary $x_L = 1/100$, the *interior critical* point x^*, and the right endpoint at $x = 1$. The equioscillation condition means that:

$$\epsilon(x_L) = +E, \qquad \epsilon(x^*) = -E, \qquad \epsilon(1) = +E.$$

The interior critical point satisfies $\epsilon'(x) = 0$, and so for our square root problem:

$$\frac{d\epsilon}{dx} = \left(b\sqrt{x} - \frac{a + bx}{2\sqrt{x}} \right) / x = 0.$$

Multiplying through by $x\sqrt{x}$, and focusing on the numerator:

$$bx - \frac{a+bx}{2} = 0 \implies bx = a \implies x^* = \frac{a}{b}.$$

Now we substitute x_L, x^*, and $x = 1$ into the relative error function $\epsilon(x)$:

$$\epsilon(x_L) = \epsilon(1/100) = \frac{a+(b/100)-(1/10)}{1/10} = 10a + (b/10) - 1 = +E. \quad (1)$$

$$\epsilon(x^*) = \frac{a+b\cdot(a/b)}{\sqrt{a/b}} - 1 = \frac{2a}{\sqrt{a/b}} - 1 = 2\sqrt{ab} - 1 = -E. \quad (2)$$

$$\epsilon(1) = a + b - 1 = +E. \quad (3)$$

Solve for a and b in $y_0 = a + bx$ by adding equs. (3) and (2):

$$(a+b-1) + (2\sqrt{ab} - 1) = 0 \implies a + b + 2\sqrt{ab} = 2$$
$$\implies \left(\sqrt{a} + \sqrt{b}\right)^2 = 2,$$

or

$$\sqrt{a} + \sqrt{b} = \sqrt{2}. \quad (4)$$

Also, let's equate equ. (1) and equ. (3) since they both equal $+E$:

$$10a + \frac{b}{10} - 1 = a + b - 1 \implies 9a = b - \frac{b}{10} = \frac{9b}{10}$$
$$\implies a = \frac{b}{10}.$$

Substituting $a = b/10$ into equ. (4):

$$\sqrt{\frac{b}{10}} + \sqrt{b} = \sqrt{2} \implies \sqrt{b}\left(\frac{1}{\sqrt{10}} + 1\right) = \sqrt{2}$$
$$\implies \sqrt{b} = \frac{\sqrt{2}}{1 + 1/\sqrt{10}} = \frac{\sqrt{20}}{\sqrt{10}+1}.$$

Let's assume that $\sqrt{10} = 3.16228$ in order to get some simple values for a and b:

$$\sqrt{b} = \frac{4.47214}{4.16228} = 1.07441, \qquad b = 1.15436, \qquad a = \frac{b}{10} = 0.115436.$$

The linear polynomial is therefore

$$y_0 = 0.115436 + 1.15436\,x,$$

which agrees with Allison's equation to five significant figures.

Aside from the equation, we can also substitute $a = 0.115442$ and $b = 1.15442$ into equ. (2) to give us a value for the peak relative error E:

$$E = 1 - 2\sqrt{0.115442 \times 1.15442} = 1 - 2\sqrt{0.133284}$$
$$= 1 - 2(0.36509) = 0.2698.$$

Therefore, the number of correct decimal digits is

$$d \;=\; -\log_{10}(E) \;=\; -\log_{10}(0.2698) \;\approx\; 0.569 \text{ digits},$$

which agrees with the 0.56 digits of accuracy mentioned in Allison's description of the square-root function.

20.2.0.2 Why Bother with $y_0 = a + bx$? The previous subsection seems to be a lot of work just to set the Newton-Raphson starting value somewhere between 1/100 and 1. To see what advantage it gives use, let's compare this linear approximation with simply fixing $y_0 = 1$ or 0.5 in the relative error equation:

$$\epsilon(x) \;=\; \frac{y_0 - \sqrt{x}}{\sqrt{x}}.$$

Case 1. When $y_0 = 1$:

$$\epsilon_0(x) = \frac{1 - \sqrt{x}}{\sqrt{x}} = \frac{1}{\sqrt{x}} - 1.$$

This is a decreasing function of x, so the worst case is at the left boundary:

$$\epsilon_0\Big(\tfrac{1}{100}\Big) \;=\; \frac{1}{1/10} - 1 \;=\; 9.0, \qquad d_0 \;=\; -\log_{10}(9.0) \;\approx\; -0.954 \text{ digits}.$$

The negative indicates that the worst-case relative error exceeds 100%, meaning that y_0 doesn't contain even one correct digit.

Case 2. $y_0 = 0.5$:

$$\epsilon_0(x) \;=\; \frac{0.5}{\sqrt{x}} - 1.$$

The worst case is again at the left boundary:

$$\epsilon_0\Big(\tfrac{1}{100}\Big) \;=\; \frac{0.5}{0.1} - 1 \;=\; 4.0, \qquad d_0 \;=\; -\log_{10}(4.0) \;\approx\; -0.602 \text{ digits}.$$

Although $y_0 = 1$ and 0.5 are both worse than using $y_0 = a + bx$ in terms of the number of correct digits in the starting value of Newton-Raphson, how does this affect its subsequent performance? In particular, how many more iterations does it take for the method to reach a good level of digit accuracy?

Recall that the recursive error definition is:

$$e_{k+1} = \frac{e_k^2}{2(\sqrt{x} + e_k)}.$$

which models how the error in Newton-Raphson reduces as the method closes in upon an answer.

For $y_0 = 0.115436 + 1.15436\,x$, we found that $e_0 = 0.2698$, and for the fixed starting values, we got $e_0 = 9$ for $y_0 = 1$ and $e_0 = 4$ for $y_0 = 0.5$. Evaluate the

recursive error equation starting with each of these e_0's, and keep going until the number of correct digits d exceeds some threshold (recall that $d = -log_{10}(e)$). Report the number of iterations required to reach that value.

Table 20.1 does all of that; its data was obtained by running `errDigits.py` with the different e_0's.

Iteration n	Linear y_0 $e_0 = 0.27$	Fixed $y_0 = 0.5$ $e_0 = 4.0$	Fixed $y_0 = 1$ $e_0 = 9.0$
0	0.569	−0.602	−0.954
1	1.542	−0.204	−0.607
2	3.398	0.308	−0.211
3	7.096	1.090	0.299
4	14.494	2.516	1.075
5	29.288	5.334	2.487
6	—	10.969	5.277
7	—	22.239	10.854
8	—	—	22.009

Table 20.1. Iterations to achieve a digit accuracy of $d > 15$ for different e_0's

Newton-Raphson's quadratic convergence only kicks in after e_k has become positive. Without $y_0 = a + bx$, the number of iterations increases from about 4 to 6 or 7. So, employing a linear approximation reduces the running time of the square-root function quite significantly.

20.3 Dirty Log Base 10

Allison's pseudo-code:

(1) $x \leq 0$ is treated as an error.

(2) If $x < 1$ then $\{\text{sign} = -1; x = 1/x\}$ else sign $= 1$, relying on the the fact that $\log(1/x) = -\log(x)$.

(3) Reduce the range of x by dividing by 10^n, such that $1/10 \leq \frac{x}{10^n} < 1$, and redefine x as $\frac{x}{10^n}$.

(4) Introduce the variable $s = \sqrt{10}\,x$, which fixes the input range to $1/\sqrt{10} \leq s < \sqrt{10}$.

(5) Change variables yet again so that $z = (s-1)/(s+1)$, and calculate the rational polynomial, $r(z)$, using the supplied P and Q coefficients, with z^2 as the argument, and multiplying by z:

$$r(z) = z\,\frac{P(z^2)}{Q(z^2)}$$

The coefficients:

P_0	3.1878	22082	024
P_1	−2.6558	08794	66
P_2	0.26686	32700	47
Q_0	3.6701	15625	115
Q_1	−4.2809	73292	83
Q_2	1.0		

(6) The $log_{10}(x)$ result is calculated using $(n + r - 0.5) \cdot \text{sign}$.

The corresponding code in dirty.py:

```
SQRT10 = 3.162278

def log10D(x):
  if x <= 0:
    raise ValueError("Math domain error")

  sign = 1
  if x < 1:
    sign = -1
    x = 1/x

  # range reduction to [0.1, 1.0]
  n = 0
  while x >= 1:
    x = x/10
    n += 1     # count powers of 10

  # set to range [1/sqrt(10, sqrt(10)]
  s = SQRT10 * x
  z = (s-1)/(s+1)
  z2 = z*z

  # Coeffs for the rational approx
  p = 3.187822 + z2*(-2.655809 + z2*0.2668633)
  q = 3.670116 + z2*(-4.280973 + z2)
  y = (n + z*p/q - 0.5) * sign
  return y
```

Step (5) in the pseudo-code depends on the hyperbolic arctangent (atanh) identity:

$$\ln(s) = 2 \cdot \operatorname{atanh}\left(\frac{s-1}{s+1}\right)$$

See section 20.3.1 to see why this identity is true.

By defining $z = \frac{s-1}{s+1}$, the Maclaurin series for $\operatorname{atanh}(z)$ is simply:

$$\ln(s) = 2\left(z + \frac{z^3}{3} + \frac{z^5}{5} + \cdots\right) = 2z\left(1 + \frac{z^2}{3} + \frac{z^4}{5} + \cdots\right).$$

If you're unsure of how $\operatorname{atanh}(z)$ is expanded into a series, look at section 20.3.2.

We want base-10 logarithms, but that's only a change of base:

$$\log_{10}(s) = \frac{\ln(s)}{\ln(10)} = \frac{2}{\ln(10)} z\left(1 + \frac{z^2}{3} + \frac{z^4}{5} + \cdots\right).$$

Since the rational polynomial approximates the series:

$$\frac{P(w)}{Q(w)} = \frac{2}{\ln(10)}\left(1 + \frac{w}{3} + \frac{w^2}{5} + \frac{w^4}{7} + \cdots\right),$$

then it is called like so:

$$\log_{10}(z) = z\frac{P(z^2)}{Q(z^2)}.$$

The advantage of evaluating z^2 instead of z, is that it halves the number of multiplications within the series calculation.

20.3.1 Deriving the $\ln(x) = \operatorname{atanh}(x)$ Identity.

Why does

$$\ln(s) = 2 \cdot \operatorname{atanh}\left(\frac{s-1}{s+1}\right)?$$

Let's assume the exponential identities for the hyperbolic cosine and sine:

$$\cosh(t) = \frac{e^t + e^{-t}}{2}, \qquad \sinh(t) = \frac{e^t - e^{-t}}{2}.$$

So,

$$\tanh(t) = \frac{\sinh(t)}{\cosh(t)} = \frac{e^t - e^{-t}}{e^t + e^{-t}}.$$

Set $y = \tanh(t)$, and multiply the numerator and denominator by e^t:

$$y = \tanh(t) = \frac{e^{2t} - 1}{e^{2t} + 1}.$$

Rearrange:

$$e^{2t} = \frac{1+y}{1-y}.$$

Take the natural logarithm of both sides:

$$2t = \ln\left(\frac{1+y}{1-y}\right) \implies t = \frac{1}{2}\ln\left(\frac{1+y}{1-y}\right).$$

But $t = \operatorname{atanh}(y)$, so:

$$\operatorname{atanh}(y) = \frac{1}{2}\ln\left(\frac{1+y}{1-y}\right), \qquad |y| < 1 \qquad (1)$$

The constraint $|y| < 1$ is due to $\tanh(t)$'s range.

We want the equality to be in terms of $\ln(x)$, which requires some variable renaming. Let $y = (x-1)/(x+1)$, and use it to simplify $(1+y)/(1-y)$:

$$\text{The numerator:} \quad (1+y) = 1 + \frac{x-1}{x+1} = \frac{(x+1)+(x-1)}{x+1} = \frac{2x}{x+1}$$

$$\text{The denominator:} \quad (1-y) = 1 - \frac{x-1}{x+1} = \frac{(x+1)-(x-1)}{x+1} = \frac{2}{x+1}$$

Plug these back into the logarithm's ratio:

$$\frac{1+y}{1-y} = \frac{2x/(x+1)}{2/(x+1)} = x.$$

Now we can rewrite the y arguments in equ. (1)'s atanh = ln equality in terms of x:

$$\operatorname{atanh}\left(\frac{x-1}{x+1}\right) = \frac{1}{2}\ln(x).$$

So:

$$\ln(x) = 2\,\operatorname{atanh}\left(\frac{x-1}{x+1}\right), \qquad x > 0.$$

The $x > 0$ domain restriction follows from $|y| < 1$.

20.3.2 Deriving the $\operatorname{atanh}(x)$ Series.

Begin with

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots, \qquad |x| < 1.$$

Integrate term-by-term from 0 to x,

$$\int_0^x \frac{dt}{1-t} = \int_0^x (1 + t + t^2 + t^3 + \cdots)\,dt,$$

to get

$$-\ln(1-x) = x + \frac{x^2}{2} + \frac{x^3}{3} + \frac{x^4}{4} + \cdots.$$

Therefore

$$\ln(1-x) = -\left(x + \frac{x^2}{2} + \frac{x^3}{3} + \frac{x^4}{4} + \cdots\right).$$

Now replace x by $-x$ in the original series:

$$\frac{1}{1+x} = 1 - x + x^2 - x^3 + x^4 - \cdots.$$

Once again integrate term-by-term,

$$\int_0^x \frac{dt}{1+t} = \int_0^x (1 - t + t^2 - t^3 + t^4 - \cdots)dt,$$

which gives us

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots.$$

Substitute these two series into the identity:

$$\operatorname{atanh}(y) = \frac{1}{2}\ln\left(\frac{1+y}{1-y}\right),$$

producing

$$\operatorname{atanh}(x) = \frac{1}{2}\left[\left(x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots\right) - \left(-x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \frac{x^5}{5} - \cdots\right)\right].$$

The terms on the right-hand side either cancel or double up, leaving:

$$\operatorname{atanh}(x) = \frac{1}{2}\left(2x + \frac{2x^3}{3} + \frac{2x^5}{5} + \frac{2x^7}{7} + \cdots\right),$$

so

$$\operatorname{atanh}(x) = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \cdots.$$

20.3.3 Checking the Polynomial's Coefficients. The rational polynomial uses P_0, P_1, P_2, Q_0, Q_1, and Q_2:

$$r(x) = \frac{P_0 + P_1 x + P_2 x^2}{Q_0 + Q_1 x + Q_2 x^2}$$

This polynomial models the power series:

$$r(x) = \frac{2}{\ln(10)}\left(1 + \frac{x}{3} + \frac{x^2}{5} + \frac{x^4}{7} + \ldots\right).$$

One simple way to test if the rational and series are equivalent is to consider their values at $x = 0$. The rational becomes

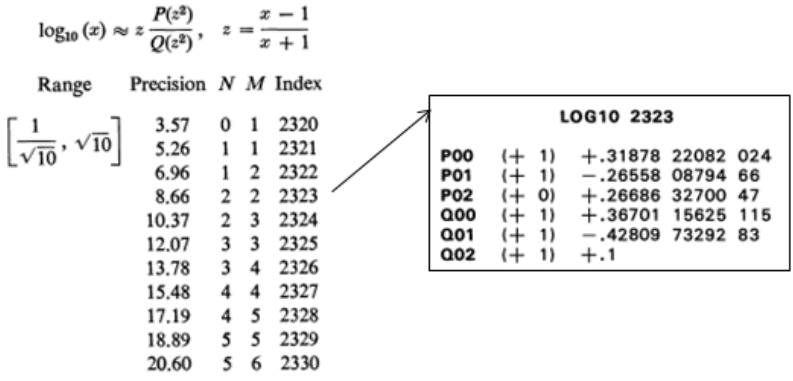
$$\frac{P_0}{Q_0} = \frac{3.187822082024}{3.670115625115} \approx 0.86859$$

while the series is:

$$\frac{2}{\ln(10)} \approx \frac{2}{2.302585093} \approx 0.86859$$

So the same (approximate) result, which is some indication that the rational does approximate the power series.

20.3.4 Why use a Rational Polynomial? Allison obtained his coefficients from Hart's *Computer Approximations* [**Har68**], but why these particular ones, especially when Hart offers a range of approximations for every function, in the form of polynomials and as rationals? Allison chose the rational listed on p.110 of Hart's text, and coefficients that offer a precision of 8.66 (8 or 9 correct decimal digits). Hart's coefficients table 2323 is printed on p.221, but Fig. 20.2 shows the rational and coefficients together.



$$\log_{10}(x) \approx z\,\frac{P(z^2)}{Q(z^2)}, \quad z = \frac{x-1}{x+1}$$

Range	Precision	N	M	Index
$\left[\frac{1}{\sqrt{10}}, \sqrt{10}\right]$	3.57	0	1	2320
	5.26	1	1	2321
	6.96	1	2	2322
	8.66	2	2	2323
	10.37	2	3	2324
	12.07	3	3	2325
	13.78	3	4	2326
	15.48	4	4	2327
	17.19	4	5	2328
	18.89	5	5	2329
	20.60	5	6	2330

Figure 20.2. A Hart Rational for $\log_{10} x$

One reason for choosing a rational is that it gives fairly good answers even near asymptotes in the function. This is an issue for atanh(x) near $x = -1$ and 1, as shown in Fig. 20.3.

However, such concerns are avoided in Allison's code by restricting the input range in

$$r = z\,\frac{P(z^2)}{Q(z^2)}.$$

$z = \frac{s-1}{s+1}$ and s is in the interval $[\frac{1}{\sqrt{10}}, \sqrt{10}]$, meaning that:

- when $s = \frac{1}{\sqrt{10}}$, $z = \frac{(1/\sqrt{10})-1}{(1/\sqrt{10})+1} = \frac{1-\sqrt{10}}{1+\sqrt{10}} \approx -0.51949$;
- when $s = 1$, $z = 0$;
- when $s = \sqrt{10}$, $z = \frac{\sqrt{10}-1}{\sqrt{10}+1} \approx +0.51949$.

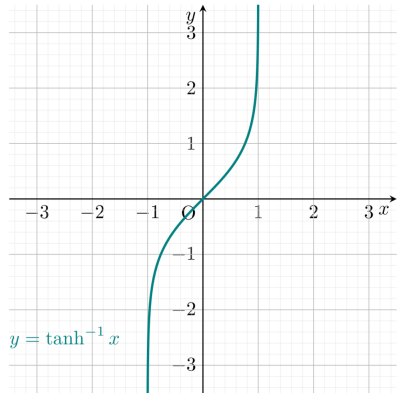


Figure 20.3. The atanh(x) Curve

20.3.5 Scaling the Result in Step (6). Step (6) specifies that the function returns $(n + r - 0.5) \cdot \text{sign}$. r is the polynomial result, so equivalent to $\log_{10}(s)$, and s is a scaled version of the x input, such that $s = (\sqrt{10}/(10^n))x$. Putting these together:

$$\begin{aligned} r &= \log_{10}(s) \\ &= \log_{10}(\frac{\sqrt{10}}{10^n}x) \\ &= \log_{10}(\sqrt{10}) - \log_{10}(10^n) + log_{10}(x) \\ &= 0.5 - n + log_{10}(x) \end{aligned}$$

So $(r + n - 0.5)$ undoes the scaling to produce $\log_{10}(x)$, with the additional step of applying the correct sign.

20.3.6 `dirty.py` in Action. dirty.py includes a test-rig which compares the results when calling `log10D(x)` versus `math.log10(x)` for $x = 0.1$, 1, 10, 50, and 1000. The output is shown below:

```
Func   | Input      | Dirty        | Python       | Error
---------------------------------------------------------------
log10  |  0.100000  | -1.000000    | -1.000000    | 1.64e-07
log10  |  1         |  0.000000    |  0.000000    | 1.64e-07
log10  |  10        |  1.000000    |  1.000000    | 1.64e-07
log10  |  50        |  1.698970    |  1.698970    | 1.61e-08
log10  |  1000      |  3.000000    |  3.000000    | 1.64e-07
```

Errors start to appear at around the seventh decimal place, within the expected range.

20.3.7 Comparison with a Modern Approach. Although Allison's article dates from 1978, its scaling of inputs, followed by the use of a polynomial approximation, is still popular, albeit in a more complicated form. One fairly easy comparison is to look at MicroPython (`https://micropython.org/`) which runs on a range of micro-controllers, including its own development board. MicroPython's math library is documented at `https://docs.micropython.org/en/latest/library/math.html`, and includes a $\log_{10}(x)$ function:

```
#include <math.h>

static const double _M_LN10 = 2.302585092994046;

double log10(double x)
{   return log(x) / (double)_M_LN10;  }
```

The code for $\log(x)$ is at `https://github.com/micropython/micropython/blob/master/lib/libm_dbl/log.c`, and is a bit too complex to describe here in detail. It essentially follows the same steps as Allison's code: it scales its input, then employs a 7-term polynomial approximation of $\log(1+x)$ constrained to the range $\sqrt{2}/2 < x < \sqrt{2}$. It sets $s = x/(2+x)$, which allows $log(1+x)$ to be expressed as $log(1+s) - log(1-s)$, which let's it employ the atanh() series.

Amazingly, Hart's textbook (from 1968) contains a very similar polynomial on p.111 and in table 2665 (p.227), as shown in Fig. 20.4.

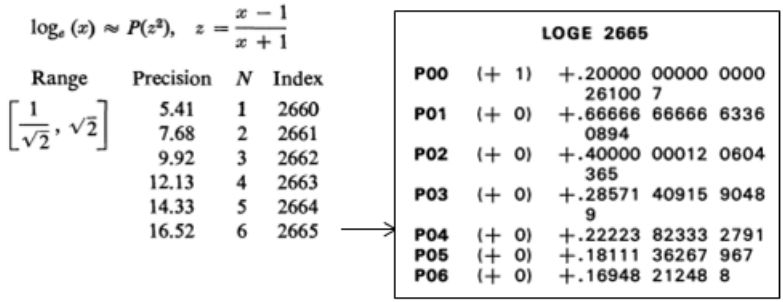


Figure 20.4. Hart Rational Polynomial for log

MicroPython runs on 32-bit architectures and, as explained in Appendix E, 32-bit IEEE 754 floating-point employ 1 sign bit, 8 exponent bits, and 23 stored fraction bits. This fractional part supports 24 bits of precision (after including the hidden leading 1), equivalent to $\log_{10}(2^{24}) \approx 7$ decimal digits. However, the

choice of coefficients in the MicroPython code means that it offers a precision of around 16 digits, making it suitable for 64-bit architectures as well.

20.4 Dirty Arctangent

Allison's pseudo-code:

(1) If $x < 0$ then return $-\arctan(|x|)$ since $\arctan(-x) = -\arctan(x)$.

(2) If $x > 1$ then return $(\pi/2 - \arctan(1/x))$ since $\arctan(|x|) = (\pi/2 - \arctan(1/|x|))$ and $x > 0$.

(3) Compute $\arctan(x)$ in the range $[0, \tan(\pi/4) = 1]$ by setting $z = x * x$ and calling the polynomial:

$$\arctan(x) = x\frac{P_0 + P_1 z + P_2 z^2}{Q_0 + Q_1 z + Q_2 z^2 + Q_3 z^3}$$

The coefficients are:

P_0	80.78998484078
P_1	72.5814378046
P_2	11.11774163116
Q_0	80.78998647615
Q_1	99.5112677199
Q_2	28.13285868067
Q_3	1.0

The corresponding code in `dirty.py`:

```
PI = 3.141593
HALF_PI = PI/2

def atanD(x):
  sign = 1
  if x < 0:
    sign = -1
    x = -x
  isInvert = (x > 1)
  if isInvert:
    x = 1/x

  # range reduction to [0, tan pi/4 = 1]
  x2 = x * x
  # Rational approximation coefficients
```

```
# Hart ARCTN 5092
p = 80.78998 + x2 * (72.58144 + x2 * 11.11774)
q = 80.78999 + x2 * (99.51128 + x2 * (28.13286 + x2))
y = x * p/q

if isInvert:
  y = HALF_PI - y    # atan(x) = pi/2 - atan(1/x)
return y * sign
```

Step (2) is used to restrict the range of x to $[0, \tan(\pi/4) = 1]$, and is best understood geometrically via Fig. 20.5.

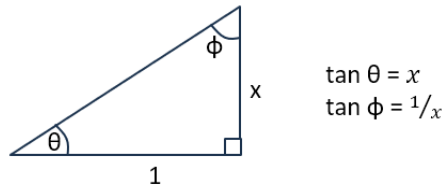


Figure 20.5. Complementary Tangents in a Right-angled Triangle

The right triangle in Fig. 20.5 has legs x and 1. If θ is the angle opposite the side of length x, then $\tan\theta = x$. The other angle ϕ satisfies $\tan\phi = 1/x$. Since the two angles of a right triangle add to $\pi/2$ radians, then

$$\theta + \phi = \pi/2,$$

which gives us

$$\arctan(x) = \pi/2 - \arctan(1/x).$$

Also, since $\theta > \pi/4$, then $\phi < \pi/4$, which allows us to constrain the polynomial's range by only passing it input equivalent to $\tan(\phi) < 1$.

The Maclaurin series expansion for $\arctan(x)$ is

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots,$$

and if we factor out an x, the terms all become even powers:

$$\arctan(x) = x\left(1 - \frac{x^2}{3} + \frac{x^4}{5} - \frac{x^6}{7} + \ldots\right).$$

If we apply the substitution $z = x^2$ to the bracketed sequence, then only sequential powers of z are needed inside it:

$$\arctan(x) = x\left(1 - \frac{z}{3} + \frac{z^2}{5} - \frac{z^3}{7} + \ldots\right)$$

It's this inner sequence that's modeled by the rational polynomial:

$$\begin{aligned}\frac{P(z)}{Q(z)} &= 1 - \frac{z}{3} + \frac{z^2}{5} - \frac{z^3}{7} + \ldots \\ &= \frac{P_0+P_1z+P_2z^2}{Q_0+Q_1z+Q_2z^2+Q_3z^3}\end{aligned}$$

We can obtain some evidence for this by considering the $z = 0$ case. The series will equal 1 while the polynomial will be:

$$\frac{P_0}{Q_0} = \frac{80.78998484078}{80.78998647615} \approx 1.0$$

There's also the question of why a rational polynomial is being used at all. It's for the same reason as atanh() utilization in the previous section – arctan() has asymptotes, as shown in Fig. 20.6, and a rational can better deal with inputs near those extremes.

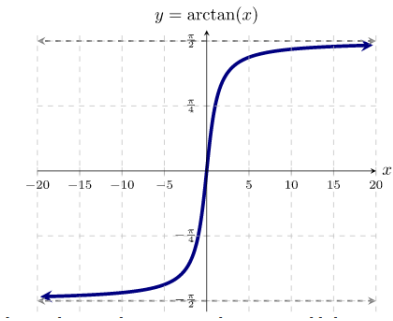


Figure 20.6. The arctan(x) Curve

As usual, the coefficients come from Hart's *Computer Approximations* [**Har68**]: the rational from p.130 and table 5092 on p.275; Fig. 20.7 shows them together.

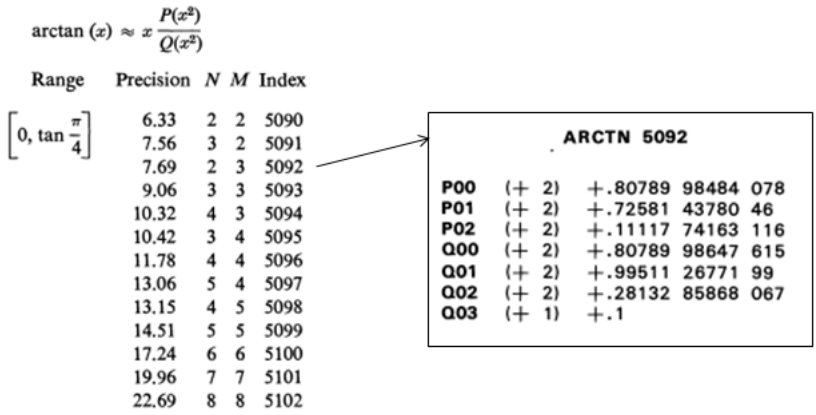
As with the atanh() polynomial, Allison chose a precision close to 8 decimal digits.

20.4.1 Deriving the arctan() **Series.** Let's define $y = \arctan(x)$ so that $x = \tan(y)$, and then differentiate both sides with respect to x:

$$\frac{d}{dx}x = \frac{d}{dx}\tan(y),$$

After applying the chain rule, and some rearranging of terms, we get

$$\frac{dy}{dx} = \frac{1}{\sec^2(y)}.$$



$$\arctan(x) \approx x \frac{P(x^2)}{Q(x^2)}$$

Range	Precision	N	M	Index
$\left[0, \tan\frac{\pi}{4}\right]$	6.33	2	2	5090
	7.56	3	2	5091
	7.69	2	3	5092
	9.06	3	3	5093
	10.32	4	3	5094
	10.42	3	4	5095
	11.78	4	4	5096
	13.06	5	4	5097
	13.15	4	5	5098
	14.51	5	5	5099
	17.24	6	6	5100
	19.96	7	7	5101
	22.69	8	8	5102

ARCTN 5092

P00	(+ 2)	+.80789 98484 078
P01	(+ 2)	+.72581 43780 46
P02	(+ 2)	+.11117 74163 116
Q00	(+ 2)	+.80789 98647 615
Q01	(+ 2)	+.99511 26771 99
Q02	(+ 2)	+.28132 85868 067
Q03	(+ 1)	+.1

Figure 20.7. A Hart Rational Polynomial for arctan

With the help of the trigonometric identity $\sec^2(y) = 1 + \tan^2(y)$, this is

$$\frac{dy}{dx} = \frac{1}{1 + \tan^2(y)},$$

and since we started with $x = \tan(y)$, then

$$\frac{d}{dx}\arctan(x) = \frac{1}{1 + x^2}.$$

The Maclaurin series for arctan() appears by expanding the right-hand side of this derivative. Let's assume that

$$\frac{1}{1+t} = 1 - t + t^2 - t^3 + t^4 - \cdots \qquad |t| < 1.$$

Substituting $t = x^2$ gives

$$\frac{1}{1+x^2} = 1 - x^2 + x^4 - x^6 + x^8 - \cdots \qquad |x| < 1.$$

Plug this into the right-hand side, and integrate term-by-term from 0 to x:

$$\arctan(x) = \int_0^x \frac{1}{1+t^2}\,dt = \int_0^x (1 - t^2 + t^4 - t^6 + t^8 - \cdots)\,dt.$$

Integrating each term yields

$$\arctan(x) = \left[t - \frac{t^3}{3} + \frac{t^5}{5} - \frac{t^7}{7} + \frac{t^9}{9} - \cdots\right]_0^x$$

which evaluates to

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \cdots$$

20.4.2 `dirty.py` in Action. The dirty.py test-rig compares the results when calling `atanD(x)`versus `math.atan(x)` for $x = 0, 0.5, 1, 5, -1$. The output is shown below:

```
Func   | Input      | Dirty        | Python       | Error
-------------------------------------------------------------
atan   |  0         |  0.000000    |  0.000000    | 0.00e+00
atan   |  0.500000  |  0.463648    |  0.463648    | 5.49e-08
atan   |  1         |  0.785398    |  0.785398    | 1.01e-07
atan   |  5         |  1.373401    |  1.373401    | 1.90e-07
atan   | -1         | -0.785398    | -0.785398    | 1.01e-07
```

Errors start to appear at around the seventh decimal place, within the expected range.

20.5 Dirty Sine

Allison's pseudo-code for $sin(x)$ is really aimed at implementing cosine, and starts by switching to that function:

$$\sin(x) = \cos(\frac{\pi}{2} - x)$$

The remaining pseudo-code has five stages:

(1) Input that's too large is detected by looking at the number of decimal digits it uses. If $|x| > 10^{d-3}$, then an error is reported.

(2) The cosine's even property, $\cos(-x) = \cos(x)$, is employed to deal with negative values.

(3) The cosine's periodicity, $\cos(2\pi + x) = \cos x$, is used to constrain the input range to $0 < x < 2\pi$.

(4) The cosine's symmetry across the four quadrants of the unit circle allows its input range to be limited to quadrant 1, along with suitable sign changes:

$$0 \le x < \frac{\pi}{2}: \quad s = +1 \quad x = x$$

$$\frac{\pi}{2} \le x < \pi: \quad s = -1 \quad x = |\pi - x|$$

$$\pi \le x < \frac{3\pi}{2}: \quad s = -1 \quad x = |\pi - x|$$

$$\frac{3\pi}{2} \le x < 2\pi: \quad s = +1 \quad x = |x - 2\pi|$$

s stores the sign required by the result.

(5) A power series polynomial is called:

$$z = x^2$$
$$y = P_0 + P_1 z + P_2 z^2 + P_3 z^3 + P_4 z^4$$
$$\cos = y * s$$

with the coefficients:

P_0	0.999999953464
P_1	−0.499999053455
P_2	0.0416635846769
P_3	−0.0013853704264
P_4	0.00002315393167

The corresponding code in dirty.py:

```
def sinD(x):
  # Shift phase to use cosine approximation
  x = HALF_PI - x
  if abs(x) > 1000:
    raise ValueError("Input too large")
  if x < 0:
    x = -x

  # Range reduction to [0, 2*pi)
  x = x % TWO_PI

  # Determine quadrant (0 to 3; ccw) and sign
  quadrant = int(x / HALF_PI)
  sign = 1
  if quadrant == 1 or quadrant == 2:
    sign = -1
    x = PI - x
  elif quadrant == 3:
```

```
    x = x - TWO_PI

x2 = x * x
# Polynomial approximation for cosine
# Hart p.118, COS 3502
y = 0.9999999 + x2*(-0.4999991 + x2*(0.04166359 + \
    x2*(-0.001385370 + x2*(0.00002315393))))

return y * sign
```

In step (1), the check $|x| > 10^{d-3}$ deals with floating-point precision loss during argument reduction, specifically in step (3) when the range is reduced to $[0, 2\pi)$ with $x = x \,\%\, 2\pi$. A modulo operation may work by repeatedly subtracting 2π. Since π can't be represented exactly in memory, all of these subtractions can cause rounding errors to accumulate.

The x threshold 10^{d-3} uses d, the chosen number of digits of precision, with -3 as a safety margin. I've hardwired $d = 7$ into the code, by checking for $|x| > 1000$.

Steps (2), (3), and (4) use reflectivity, periodicity, and symmetry as they apply to the cosine (see Fig. 20.8) to reduce the input range to $[0, \pi/2)$.

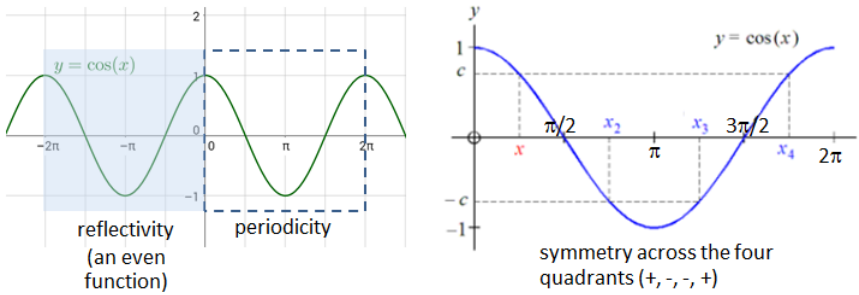


Figure 20.8. Reflectivity, periodicity, and symmetry for cosine

The Maclaurin series expansion for $\cos(x)$ is

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \ldots$$

The substitution $z = x^2$ turns the polynomial into an integer sequence of powers:

$$\cos(x) = 1 - \frac{z}{2} + \frac{z^2}{24} - \frac{z^3}{720} + \frac{z^4}{40320} - \ldots$$

This is the first time our approximation has *not* been a rational polynomial. There's no need since cosine doesn't have asymptotes, unlike atanh() and arctan().

Once again, the coefficients come from Hart's *Computer Approximations* [**Har68**]: the polynomial from p.123 and its table 3502 from p.241; Fig. 20.9 shows them together.

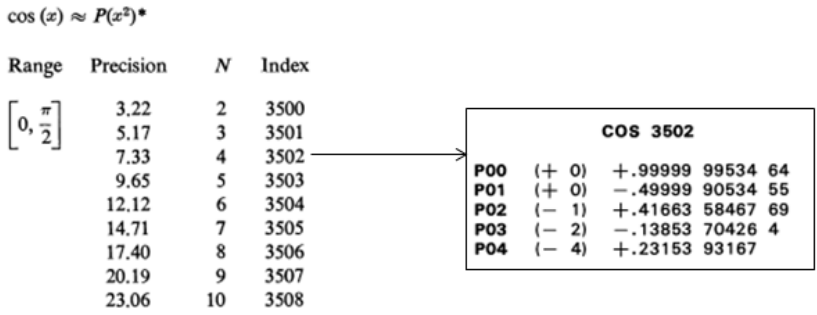


Figure 20.9. A Hart Power Series Polynomial for cos

Allison wanted precision close to 7 decimal digits (i.e. $d = 7$), which backs up his test that $x > 1000$ or 10^{d-3}.

The coefficients in table 3502 are 'related' to the Maclaurin expansion for cosine, as shown in Table 20.2. The differences are due to Hart's use of Remez approximation which tends to modify the series coefficients depending on the input range constraints.

Coefficient	Coef Float	Series Fraction	Series Float
P_0	0.999999953464	$+1$	1.0
P_1	-0.499999053455	$-1/2!$	-0.5
P_2	0.0416635846769	$+1/4!$	0.04166666666 ...
P_3	-0.0013853704264	$-1/6!$	-0.00138888888 ...
P_4	0.00002315393167	$+1/8!$	0.00002480158 ...

Table 20.2. Comparison of Polynomial and Series Coefficients

The dirty.py test-rig compares `sinD(x)` versus `math.sin(x)`. The output:

```
Func   | Input      | Dirty        | Python       | Error
-------------------------------------------------------------
sin    |  0         |  0.000000    |  0.000000    | 2.56e-07
sin    |  0.785398  |  0.707107    |  0.707107    | 2.26e-07
sin    |  1.570796  |  1.000000    |  1.000000    | 1.00e-07
sin    |  3.141593  |  0.000000    |  0.000000    | 9.02e-08
sin    |  6.283185  | -0.000000    | -0.000000    | 4.37e-07
```

Errors start to appear at around the seventh decimal place.

20.5.1 Deriving the Polynomial Coefficients. The majority of the polynomials calculated in Hart's *Computer Approximations* utilize the Remez exchange algorithm. While the power series version of that algorithm is fairly straightforward to implement, the rational Remez requires non-linear optimization. The only Python library I could find that offered both forms was Baryrat (`https://github.com/c-f-h/baryrat`), which actually utilizes a modern replacement for Remez.

For the power series Remez used by `cosD()`, there are several Python implementations to choose from. I went with `py-remezfit` (`https://github.com/ememece67/py-remezfit`), which packages everything into a single file, although it also relies on numpy and scipy. The following call generates 4th-order power series coefficients for cosine in the range $[0, \pi/2)$, with x^2 as the polynomial argument:

```
import numpy as np
from remezfit import remezfit

coefs, prec, _ = remezfit(lambda x: np.cos(x),
                    0, np.pi/2, 4, arg = lambda x: x*x,
                    dtype = np.single)
print("Coefs:", coefs)
print("Correct Digits:", prec)
```

The output:

```
Coefs: [ 9.9999994e-01 -4.9999911e-01  4.1663747e-02
        -1.3854750e-03  2.3172470e-05]
Correct Digits: 1.2458856141206454e-07
```

The generated coefficients aren't quite the same as Hart's 3502 table, probably due to `py-remezfit` using a modern variant of Remez, but closer than the Maclaurin series in Table 20.2.

20.6 The Maths of Function Approximation

The preceding couple of sections focused on functions from Allison's "Quick and Dirty Functions for BASIC" article as concrete examples of the utility of function approximation. However, we didn't address the mathematics behind the generation of the coefficients, instead just utilizing values prepared for us by John F. Hart and his co-authors [**Har68**].

In the following sections, the maths behind the Taylor series and Lagrange polynomials will be considered, followed by the Chebyshev approximation. Chebyshev offers near optimal behavior while being much simpler to understand than the Remez algorithm favored by Hart.

For rational polynomials, we'll consider Padé approximation, which performs well near singularities, and, aside from its reliance on a linear system solver, is much less complicated than the rational Remez method.

20.7 The Taylor Series

Perhaps the simplest way of obtaining an approximation to a function $f(x)$ is via its Taylor series expansion. Suppose you want to approximate $f(x)$ over the interval $[a, b]$, centered at some $x_0 \in [a, b]$, then Taylor's theorem states:

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2}f''(x_0) + \cdots + \frac{(x - x_0)^n}{n!}f^{(n)}(x_0) + \cdots$$

Maclaurin's series is a special case when $x_0 = 0$:

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + \cdots$$

Some examples:

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots \quad x \in \mathbb{R} \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots \quad x \in \mathbb{R} \\ \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots \quad x \in \mathbb{R} \end{aligned}$$

However, many series place limits on the x-range, such as:

Function	Maclaurin Series	Input Range	Radius R
$\frac{1}{1+x}$	$1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + \cdots$	$\lvert x\rvert < 1$	1
$\frac{1}{1+x^2}$	$1 - x^2 + x^4 - x^6 + x^8 - x^{10} + x^{12} - x^{14} + \cdots$	$\lvert x\rvert < 1$	1
$\ln(1 + x)$	$x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \frac{x^6}{6} + \frac{x^7}{7} - \frac{x^8}{8} + \cdots$	$-1 < x \leq 1$	1
$\sqrt{1+x}$	$1 + \frac{x}{2} - \frac{x^2}{8} + \frac{x^3}{16} - \frac{5x^4}{128} + \cdots$	$-1 \leq x \leq 1$	1
$\tan(x)$	$x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315} + \cdots$	$-\frac{\pi}{2} < x < \frac{\pi}{2}$	$\pi/2$
$\sec(x)$	$1 + \frac{x^2}{2} + \frac{5x^4}{24} + \frac{61x^6}{720} + \cdots$	$-\frac{\pi}{2} < x < \frac{\pi}{2}$	$\pi/2$
$\arctan(x)$	$x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \frac{x^{11}}{11} + \frac{x^{13}}{13} - \cdots$	$-1 \leq x \leq 1$	1

For an even longer list, see section 1.3 of Zwillinger's *CRC Standard Mathematical Tables and Formulae* [**Zwi96**] (a copy is available at the Internet Archive

at https://archive.org/details/crcstandardmathe00chem/page/40/mode/2up).

The input range restrictions depend on the distance of the expansion point ($x_0 = 0$) from the nearest singularity, its *radius of convergence*. There's no point passing an x value to a series outside its prescribed range because it won't converge to an answer. Code implementing the series will loop forever unless there's some iteration limit, but the returned value will be meaningless.

By far the easiest way to implement Taylor series in Python is with the Sympy library for symbolic mathematics (https://www.sympy.org/). Expressions can be manipulated algebraically, and Sympy's series module (https://docs.sympy.org/latest/modules/series/series.html#sympy.series.series.series) is ideal for coding the previous examples (see seriesGen.py):

```
import math
import sympy as sp

fNames = ["1/(1+x)", "exp(x)", "cos(x)", "sin(x)",
          "log(1+x)", "atan(x)" ]

spFns = [lambda x: 1/(1+x), sp.exp, sp.cos, sp.sin,
         lambda x: sp.log(1+x), sp.atan ]

x = sp.Symbol("x")
for i, fnm in enumerate(fNames):
  print(f"{fnm}")
  series = sp.series(spFns[i](x), x0=0, n=8, dir='-')  # .removeO()
  print(" ", series)
  print()
```

The x0 argument in the series() call sets the expansion center, while n is the maximum term power. The output:

```
1/(1+x)
  1 - x + x**2 - x**3 + x**4 - x**5 + x**6 - x**7 + O(x**8)

exp(x)
  1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 + x**6/720 +
   x**7/5040 + O(x**8)

cos(x)
  1 - x**2/2 + x**4/24 - x**6/720 + O(x**8)

sin(x)
  x - x**3/6 + x**5/120 - x**7/5040 + O(x**8)

log(1+x)
  x - x**2/2 + x**3/3 - x**4/4 + x**5/5 - x**6/6 +
```

```
  x**7/7 + O(x**8)

atan(x)
  x - x**3/3 + x**5/5 - x**7/7 + O(x**8)
```

The "O()" at the end of each expansion gives the order of the next term in the series.

faUtils.py contains two functions for utilizing Sympy series – `seriesCoefs()` and `evalPowers()`, which are employed below to generate the Maclaurin series for e^x and to calculate e^2.

```
>>> import sympy as sp
>>> from faUtils import *
>>> coefs  = seriesCoefs(sp.exp, verbose=True)
   Poly(1/362880*_u**9 + 1/40320*_u**8 + 1/5040*_u**7 + 1/720*_u**6 +
   1/120*_u**5 + 1/24*_u**4 + 1/6*_u**3 + 1/2*_u**2 + _u + 1, _u,
   domain='EX')
Coefs in ascending power order:
   [1, 1, 1/2, 1/6, 1/24, 1/120, 1/720, 1/5040, 1/40320, 1/362880]

>>> evalPowers(coefs, 2)
7.3887125220458545
>>> import math   # for comparison
>>> math.exp(2)
7.38905609893065
```

The series value for e^2 is only 'fairly' close to the one produced by `math.exp()`, with the discrepancy due to a combination of round-off and truncation errors, which I'll consider next.

20.7.1 Truncation Errors. Truncation errors are inherent in the nature of series. For instance, if we only use the first four terms of e^x to calculate e^2, then we only get

$$e^2 \approx 1 + 2 + \frac{2^2}{2!} + \frac{2^3}{3!} = 6.3333.$$

A function $f(x)$ can be written as an n-term Taylor series plus a truncation error:

$$\begin{aligned} f(x) &= T_n(x) + E_n(x) \\ &= \sum_{k=0}^{n} \frac{f^{(k)}(x_0)(x-x_0)^k}{k!} + E_n(x) \end{aligned}$$

x_0 is the expansion point, and $f^{(k)}(x)$ is the k-th derivative of $f(x)$.

We can obtain a value for $E_n(x)$ by utilizing the property that if $f(x)$ has $n+1$ derivatives in an interval $[a, b]$ containing x_0, then for each x in $[a, b]$ there exists a z between x and x_0 such that

$$E_n(x) = \frac{f^{(n+1)}(z)(x - x_0)^{n+1}}{(n+1)!}.$$

If we know that M is the maximum value of $|f^{(n+1)}(z)|$ in the interval, then

$$|E_n(x)| \leq \frac{M|x - x_0|^{n+1}}{(n+1)!}.$$

Let's employ this bound to estimate the error for e^2 (i.e. when $x = 2$) for a Maclaurin series with $n = 9$ terms:

$$E_n(x) = \frac{f^{(9+1)}(z)(x)^{(9+1)}}{(9+1)!} = \frac{e^z 2^{10}}{10!}.$$

z is between $x_0 = 0$ and 2, and let's say that $e \approx 3$. Therefore,

$$|E_n(x)| \leq \frac{3^2 2^{10}}{10!} \approx 0.00254.$$

Returning to the Python session from earlier, this estimate turns out to be too pessimistic:

```
>>> e2Approx = evalPowers(coefs, 2)
>>> e2Approx
7.3887125220458545
>>> abs(e2Approx - math.exp(2))
0.0003435768847959153
```

The Taylor series definition seems to suggest that it's always better to increase the number of terms in order to reduce the truncation error. A visual depiction of the benefit is shown in Fig. 20.10.

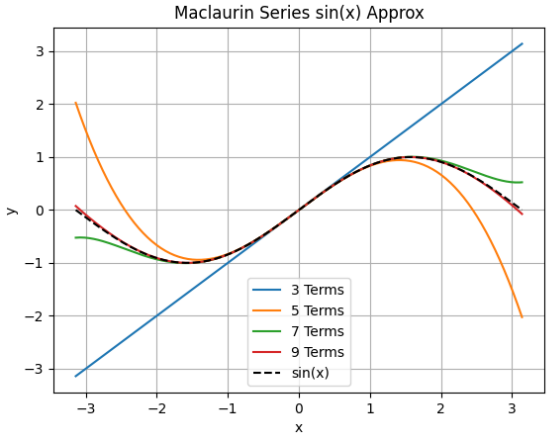


Figure 20.10. Adding More Terms to a Maclaurin Series for $\sin(x)$

sinMac.py generates four versions of the Maclaurin series for $\sin(x)$ with increasing numbers of terms, and plots them against `math.sin(x)` between $-\pi$

and π. Fig. 20.10 highlights how the accuracy of the series around $x_0 = 0$ increases as the number of terms increases, but also shows that it tails off rapidly. sinFar.py examines this property over a wider range (see Fig. 20.11).

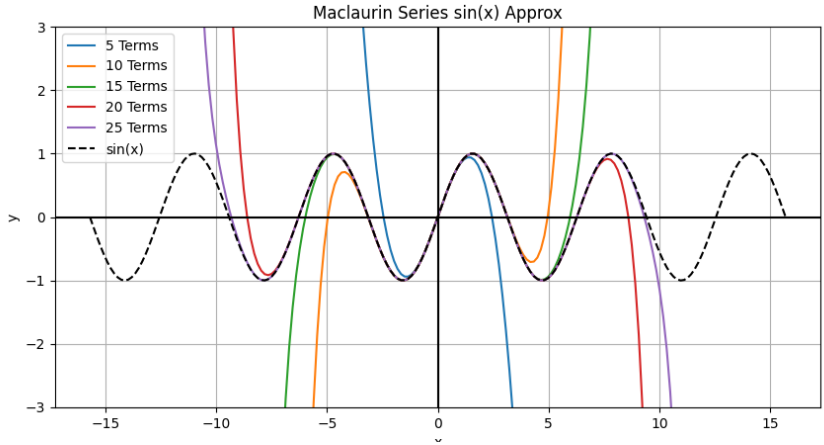


Figure 20.11. A Maclaurin Series for $\sin x$ between -5π and 5π

Adding more terms definitely improves the span of the accuracy, but the number of terms quickly becomes excessive. Also, their coefficients become exceedingly small, raising the issue of rounding errors when they're used in calculations. For example, the coefficients for the 25-term version of $\sin(x)$ are:

```
Using 25 terms:
Coefs in ascending power order:
[0, 1, 0, -1/6, 0, 1/120, 0, -1/5040, 0, 1/362880, 0, -1/39916800, 0,
1/6227020800, 0, -1/1307674368000, 0, 1/355687428096000, 0,
-1/121645100408832000, 0, 1/51090942171709440000, 0,
-1/25852016738884976640000, 0 ]
```

20.7.2 Round-off Errors. Fig. 20.12 shows Maclaurin series expansions for e^{-x} as generated by expFar.py.

As with $\sin(x)$, version of the e^{-x} series with a large number of terms have very small coefficients:

```
Using 25 terms:
Coefs in ascending power order:
[1, -1, 1/2, -1/6, 1/24, -1/120, 1/720, -1/5040, 1/40320, -1/362880,
1/3628800, -1/39916800, 1/479001600, -1/6227020800, 1/87178291200,
-1/1307674368000, 1/20922789888000, -1/355687428096000,
1/6402373705728000, -1/121645100408832000, 1/2432902008176640000,
-1/51090942171709440000, 1/1124000727777607680000,
-1/25852016738884976640000,  1/620448401733239439360000 ]
```

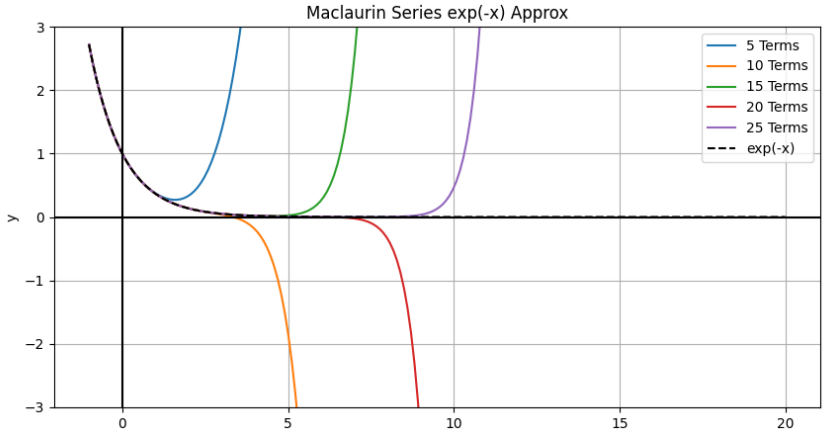


Figure 20.12. A Maclaurin Series for e^{-x} between -1 and 20

Numerical evidence for the resulting rounding errors is produced by expRound.py which calls `expApprox()` to evaluate n terms of e^x:

```
def expApprox(x, n):
  exp = 0
  for i in range(n+1):
    exp += (x**i)/math.factorial(i)
  return exp
```

The program output below shows the result of passing $x = -10$ and $x = -30$ to `expApprox()` to calculate e^{-10} and e^{-30} with different n's. `math.exp()` is also called for the sake of comparison:

```
x == -10
Math e^-10 == 4.53999e-05
Using 10 terms, approx == 1.34259e+03
Using 200 terms, approx == 4.53999e-05

x == -30
Math e^-30 == 9.35762e-14
Using 200 terms, approx == -8.55302e-05
Using 1000 terms, approx == -8.55302e-05
```

A reasonable answer for e^{-10} is produced, but only when using 200 terms. However, rounding errors means that a good result for e^{-30} is out of reach.

The problem is exacerbated by evaluating e^x with negative values which causes the series to add and subtract alternating terms (i.e. $e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$). As soon as a calculation involves the subtraction of similar-sized values then the likelihood of rounding errors increases substantially.

We've already seen the best solution to this problem in dirty.py: the input should be constrained to a narrow range, which in the Taylor series case should be around its x_0 value. `expTaylor()` in expApprox.py implements that strategy:

```
LN2 = math.log(2)   # 0.6931471805599453094

def expTaylor(x, maxIters=16, eps=1e-15):
  if x == 0.0:
    return 1.0

  k = int(round(x/LN2))
  xR = x - k*LN2  # subtract nearest multiple
  term = 1.0
  tot = 1.0
  for n in range(1, maxIters):
    term *= xR/n
    tot += term
    if abs(term) <= eps * max(1.0, abs(tot)):
      # print(n)   # usually about 12
      break
  return tot * (2.0**k)
```

The input is reduced by subtracting the nearest k-th multiple of ln(2) to guarantee that `xR` is in the range $[-\ln(2)/2, \ln(2)/2]$. After the series for e^{xR} has been calculated (using incremental term building rather than expensive calls to `math.factorial()`), then the result for the original x input is constructed with $e^x = 2^k * e^{xR}$.

The test-rig in expApprox.py compares the results of `expTaylor(x)` against `math.exp(x)` for a range of values:

```
     x       Approx       math.exp(x)     Error
-------------------------------------------------
  -100.0   3.72008e-44   3.72008e-44   1.740e-15
   -10.0   4.53999e-05   4.53999e-05   5.970e-16
    -1.0   3.67879e-01   3.67879e-01   1.509e-16
    -0.5   6.06531e-01   6.06531e-01   3.661e-16
     0.5   1.64872e+00   1.64872e+00   1.347e-16
     1.0   2.71828e+00   2.71828e+00   1.634e-16
    10.0   2.20265e+04   2.20265e+04   3.303e-16
   100.0   2.68812e+43   2.68812e+43   1.289e-15
```

20.7.3 The Problem of Singularities. As mentioned at the start of this section, many Taylor series have a restricted input range, and a series will only converge when given a value inside that range. The range is determined by the distance of the expansion point (x_0) from the nearest singularity, known as its radius of convergence.

This constraint is made more complicated by the fact that while a singularity *may* be drawn as an asymptote when the curve is plotted, the radius is actually located in the complex plane, and so may not be visible (e.g. see. Sec. 20.7.4 below).

Even when the asymptote is visible, the radius is more restrictive than the graph of the curve would suggest. For example, consider $1/(1+x)$ and $\ln(1+x)$ in Fig. 20.13:

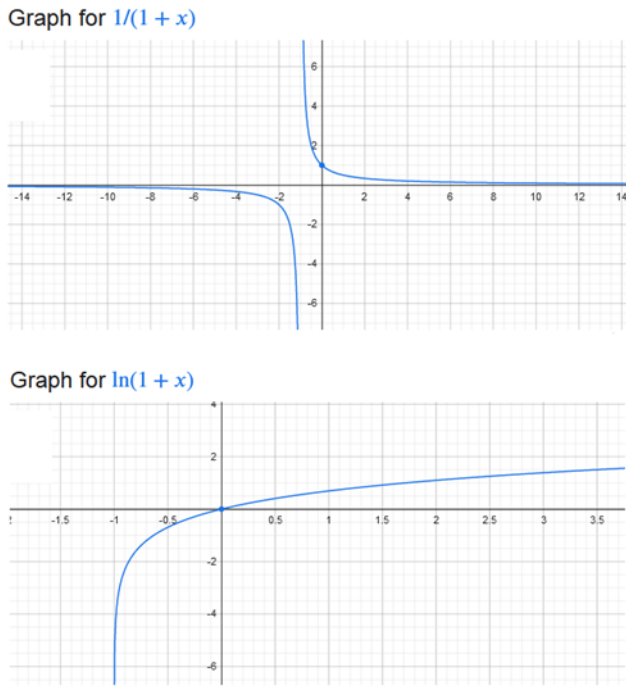


Figure 20.13. Graphs of $1/(1+x)$ and $\ln(1+x)$

$1/(1+x)$ has an asymptote at $x=-1$ which heads towards $\pm\infty$ depending on the direction of approach, while $\ln(1+x)$ has a pole at $x=-1$. The Maclaurin series for $\ln(1+x)$ should clearly not converge for values of $x \leq -1$, but its positive values are also constrained (to the range $-1 < x \leq 1$). The same question arises for $1/(1+x)$ (its range is $|x|<1$), with the additional issue of why we can't pass larger negative numbers to the series (e.g. $x=-5$).

logCat.py generates five versions of the Maclaurin series for $\ln(1+x)$ with increasing numbers of terms, and plots them against `math.log()` in Fig. 20.14.

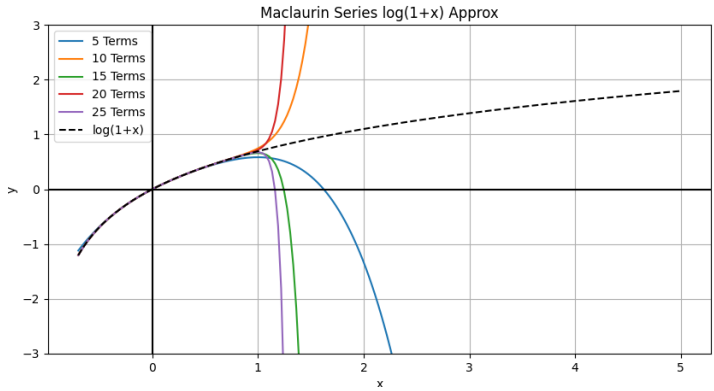


Figure 20.14. Maclaurin Series for $\ln(1+x)$ with Different Numbers of Terms

The range restriction of $-1 < x \leq 1$ makes it impossible to get an accurate answer beyond $x = 1$ no matter how many terms are thrown at the problem.

One way around this limitation is to use identities to rewrite the equation, a strategy we first saw in dirty.py. Recall that `log10D()` utilizes the hyperbolic function $\text{atanh}(x)$. Its Maclaurin series is

$$\text{atanh}(x) = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \cdots$$

Its radius of convergence is $R = 1$, restricting it to the range $-1 < x < 1$, which seems to offer no improvements over sticking with $\ln(1+x)$. However, the $\text{atanh}(x)$ series only employs addition, so avoids the cancellation errors related to subtraction that affect $\ln(1+x)$.

`logApprox()` in logApprox.py implements this approach:

```
LN2 = 0.6931471805599453094
SQRT2 = 1.4142135623730950488
SQRT_HALF = SQRT2/2         # sqrt(2/4) == sqrt(2)/2

def logApprox(x, maxIters=16, eps=1e-15):
  if x <= -1.0:
    raise ValueError("logApprox() requires x > -1")
  if x == 0.0:
    return 0.0

  m = 1.0 + x; k = 0
  # m is scaled into the interval
  # sqrt(1/2) <= m <= sqrt(2)
  while m > SQRT2:
    m *= 0.5; k += 1
```

```
  while m < SQRT_HALF:
    m *= 2; k -= 1
  r = m - 1
  z = r/(2 + r)
  z2 = z*z

  term = z; tot = z    # for atanh series
  denom = 3
  for _ in range(maxIters):
    term *= z2
    delta = term/denom
    tot += delta
    if abs(delta) <= eps * max(1.0, abs(tot)):
      break
    denom += 2
  return k*LN2 + 2.0*tot
```

`logApprox()` scales its input before evaluating the atanh() series. The reduction is based on the equality $1 + x = m \cdot 2^k$ which maps x to m in the interval $\sqrt{1/2} \leq m \leq \sqrt{2}$. Then $r = m - 1$ is used to shift the range to either side of 0, and r is utilized in the ln() = atanh() identity:

$$\ln(1 + r) = 2 \cdot \operatorname{atanh}(r/(2 + r))$$

Another simplification is to let $z = r/(2 + r)$, reducing the series to

$$\operatorname{atanh} z = z + z^3/3 + z^5/5 + z^7/7 + \cdots$$

so that

$$\ln(1 + r) = 2 \cdot (z + z^3/3 + z^5/5 + \cdots)$$

All of these renamings and scalings end up placing $|z| \leq 0.171573\ldots$, which is well within the convergence range of the atanh() series.

At the end of `logApprox()`, its result for the original x input is constructed using

$$\ln(1 + x) = (k \cdot \log 2) + \log m.$$

The test-rig in logApprox.py compares the results of `logApprox(x)` against `math.log(1+x)` (actually `math.log1p(x)`):

```
       x         Approx    math.log1p(x)   Rel Error
-------------------------------------------------------
 -0.999999  -1.38155e+01  -1.38155e+01   0.000e+00
    -0.999  -6.90776e+00  -6.90776e+00   0.000e+00
     -0.99  -4.60517e+00  -4.60517e+00   1.929e-16
      -0.1  -1.05361e-01  -1.05361e-01   3.952e-16
     -0.01  -1.00503e-02  -1.00503e-02   8.630e-16
    -0.001  -1.00050e-03  -1.00050e-03   8.669e-16
     0.001   9.99500e-04   9.99500e-04   1.100e-13
```

```
  0.01   9.95033e-03   9.95033e-03   1.046e-15
   0.1   9.53102e-02   9.53102e-02   7.280e-16
     1   6.93147e-01   6.93147e-01   0.000e+00
    10   2.39790e+00   2.39790e+00   1.852e-16
   100   4.61512e+00   4.61512e+00   1.924e-16
1e+06   1.38155e+01   1.38155e+01   0.000e+00
```

20.7.4 Invisible Asymptotes. Since asymptotes can occur in the complex plane for a function, they're not necessarily visible when that function is plotted in the real plane. This is illustrated by $1/(1+x^2)$, shown in Fig. 20.15.

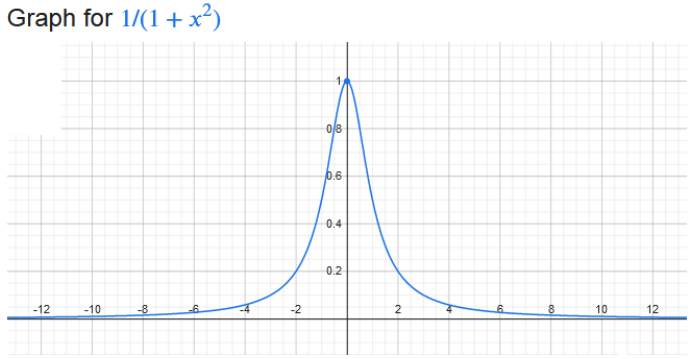


Figure 20.15. $1/(1+x^2)$ Graphed

Its Maclaurin series is:

$$\frac{1}{1+x^2} = 1 - x^2 + x^4 - x^6 + x^8 - x^{10} + x^{12} - x^{14} + \cdots,$$

and has a radius of convergence $R = 1$, limiting is input range to $|x| < 1$. The reason becomes slightly clearer if we factor the function's denominator: $1 + x^2 = (x - i)(x + i)$. Singularities occur when $x = \pm i$, and the function's radius is the distance from the origin to the closest of those points.

limitRadius.py repeatedly plots the Maclaurin series for $1/(1+x^2)$ between $x = -5$ and 5, each time with more terms. Fig. 20.16 shows that the convergence limits become more sharply defined, but never reliably reach beyond the $|x| < 1$ boundaries.

There's nothing we can do to change the singularities but a Taylor series can utilize any expansion point. In movedCenter.py, $1/(1+x^2)$ is again plotted between $x = -5$ and 5, but expanded around $x = 2$. The code changes are minimal:

```
for n in [4, 5, 8, 9]:
  print(f"\nUsing {n} terms:")
  fCoeffs = faUtils.seriesCoefs(fn, x0=2, nTerms=n,
```

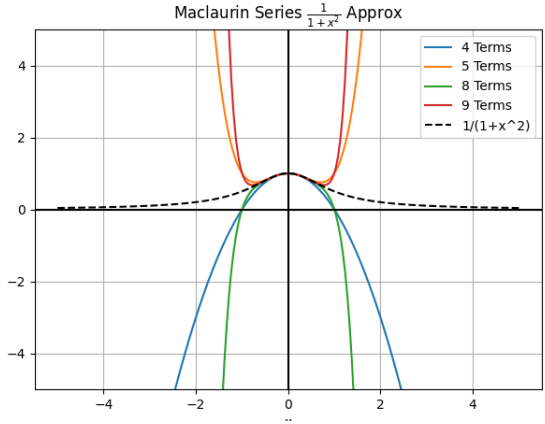


Figure 20.16. Plots of Maclaurin Series for $1/(1+x^2)$

```
                                    verbose=True)
ys = [ faUtils.evalPowers(fCoeffs, x, x0=2) for x in xs]
plt.plot(xs, ys, label= f"{n} Terms at x0=2")
```

The four plots are shown in Fig. 20.17.

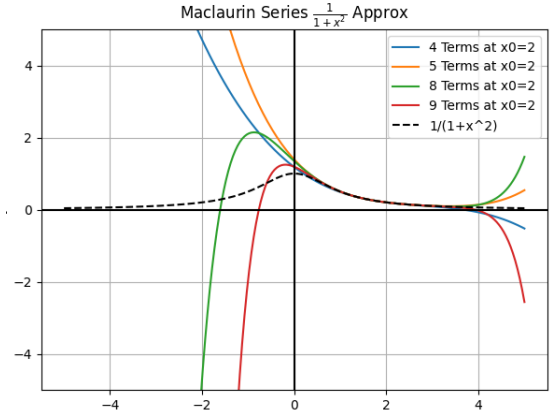


Figure 20.17. Plots of Taylor Series at $x_0 = 2$ for $1/(1+x^2)$

To expand $f(x) = 1/(1+x^2)$ about $x = 2$, let $h = x - 2$, and substitute it into the equation:

$$f(x) = \frac{1}{1+(2+h)^2} = \frac{1}{5+4h+h^2}.$$

Factoring out 5,

$$f(x) = \frac{1}{5}\left(\frac{1}{1 + {}^4\!/_5 h + {}^1\!/_5 h^2}\right).$$

The right-hand side can be expanded by using

$$\frac{1}{1+u} = 1 - u + u^2 - u^3 + \cdots, \qquad |u| < 1,$$

if we set

$$u = \frac{4}{5}h + \frac{1}{5}h^2.$$

After collecting powers of h, we get

$$f(x) = \frac{1}{5} - \frac{4}{25}(x-2) + \frac{11}{125}(x-2)^2 - \frac{24}{625}(x-2)^3 + \frac{41}{3125}(x-2)^4 + \cdots$$

The polynomial coefficients generated by `Sympy.series()` in the code above are reassuringly the same:

```
Using 5 terms:
   Poly(41/3125*_u**4 - 24/625*_u**3 + 11/125*_u**2 -
    4/25*_u + 1/5, _u, domain='EX')
Coefs in ascending power order:
   [1/5, -4/25, 11/125, -24/625, 41/3125]
```

The singularities for $1/(1+x^2)$ are still at $x = \pm i$, but the radius of convergence is now the shortest distance from the expansion point at $x_0 = 2$. Interpreting the numbers as points in the complex plane gives us:

$$2 = (2,0), \qquad i = (0,1), \qquad -i = (0,-1).$$

To which we apply the distance formula,

$$|2 - i| = \sqrt{(2-0)^2 + (0-1)^2} = \sqrt{4+1} = \sqrt{5}$$

and

$$|2 + i| = \sqrt{(2-0)^2 + (0+1)^2} = \sqrt{4+1} = \sqrt{5}.$$

Since both distances are the same, $R = \sqrt{5}$, so the series converges for $|x-2| < \sqrt{5}$. In other words, $x \in [0.236..., 4.236...]$, which is (roughly) confirmed by the plots in Fig. 20.17.

20.7.5 Removable Singularities. A *removable* singularity means that the Taylor series for the function can be rearranged to remove the element that tends to infinity. Consider $\sin(x)/x$ whose denominator appears to send the function towards a singularity as x approaches 0. However, the Taylor series for $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \ldots$, so the denominator can be divided into each term to give $\sin(x)/x = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} + \frac{x^8}{9!} - \ldots$.

An interactive Python session confirms these coefficients:

```
>>> import sympy as sp
>>> from faUtils import *
>>> f = lambda x: sp.sin(x)/x
>>> coefs = seriesCoefs(f, verbose=True)
   Poly(1/362880*_u**8 - 1/5040*_u**6 + 1/120*_u**4 -
   1/6*_u**2 + 1, _u, domain='EX')
Coefs in ascending power order:
   [1, 0, -1/6, 0, 1/120, 0, -1/5040, 0, 1/362880, 0]
```

20.8 The Lagrange Polynomial

The main disadvantage of using a Taylor series approximation is that it only approximates the function around a single expansion point. This gives it excellent local accuracy, but this decreases rapidly at points further away from x_0. The other issue is that the presence of singularities limit the series' range even for values which seem to be nowhere near the asymptote.

One solution is to move to function approximation that utilizes interpolation over a series of (x, y) coordinates. Perhaps the simplest approach of this type is the *Lagrange polynomial*.

Let's assume that we have $n+1$ numbers, $x_0, x_1, \ldots, x_n$, and a function $f(x)$. The Lagrange polynomial is

$$P(x) = f(x_0)L_{n,0}(x) + \cdots + f(x_n)L_{n,n}(x) = \sum_{k=0}^{n} f(x_k)L_{n,k}(x),$$

where, for each $k = 0, 1, \ldots, n$,

$$\begin{aligned} L_{n,k}(x) &= \frac{(x-x_0)(x-x_1)\ldots(x-x_{k-1})(x-x_{k+1})\ldots(x-x_n)}{(x_k-x_0)(x_k-x_1)\ldots(x_k-x_{k-1})(x_k-x_{k+1})\ldots(x_k-x_n)} \\ &= \prod_{\substack{i=0 \\ i\neq k}}^{n} \frac{(x-x_i)}{(x_k-x_i)}. \end{aligned}$$

Note that for $k = 0$, $L_{n,0}(x_0) = 1$ while for all the other x values $L_{n,0}(x_i) = 0$. Similarly, when $k = 1$, $L_{n,1}(x_1) = 1$ while $L_{n,1}(x_i) = 0$ for the other x's, and this property applies to all the $L_{n,k}()$ terms. If the coordinates are $(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)$, then this implies that

$$\begin{array}{rcccl} P(x_0) &=& 1\cdot f(x_0) + 0\cdot f(x_1) + \cdots + 0\cdot f(x_n) &=& y_0 \\ P(x_1) &=& 0\cdot f(x_0) + 1\cdot f(x_1) + \cdots + 0\cdot f(x_n) &=& y_1 \\ \vdots && \vdots && \vdots \\ P(x_n) &=& 0\cdot f(x_0) + 0\cdot f(x_1) + \cdots + 1\cdot f(x_n) &=& y_n \end{array}$$

So $P(x)$ passes through all of the supplied coordinates.

20.8.1 Approximating $1/x$. We have $x_0 = 2$, $x_1 = 2.75$, and $x_2 = 4$, and want to find the Lagrange polynomial for $f(x) = 1/x$. First, determine $L_{2,0}(x)$, $L_{2,1}(x)$, and $L_{2,2}(x)$:

$$\begin{aligned} L_{2,0}(x) &= \frac{(x-2.75)(x-4)}{(2-2.5)(2-4)} &&= \frac{2}{3}(x-2.75)(x-4), \\ L_{2,1}(x) &= \frac{(x-2)(x-4)}{(2.75-2)(2.75-4)} &&= -\frac{16}{15}(x-2)(x-4), \\ L_{2,2}(x) &= \frac{(x-2)(x-2.75)}{(4-2)(4-2.5)} &&= \frac{2}{5}(x-2)(x-2.75). \end{aligned}$$

Also, $f(x_0) = f(2) = 1/2$, $f(x_1) = f(2.75) = 4/11$, and $f(x_2) = f(4) = 1/4$, so that

$$\begin{aligned} P(x) &= \sum_{k=0}^{2} f(x_k) L_{2,k}(x) \\ &= \frac{1}{3}(x-2.75)(x-4) - \frac{64}{165}(x-2)(x-4) + \frac{1}{10}(x-2)(x-2.75) \\ &= \frac{1}{22}x^2 - \frac{35}{88}x + \frac{49}{44}. \end{aligned}$$

It's interesting to compare this polynomial with a Taylor series for $1/x$. Let's create one centered on $x = 1$:

$$\frac{1}{x} = \frac{1}{1+(x-1)}.$$

Expand the right-hand side using the geometric series for $1/(1+u)$ with $u = x-1$:

$$\frac{1}{x} = \sum_{n=0}^{\infty} (-1)^n (x-1)^n.$$

The first few terms are

$$\frac{1}{x} = 1 - (x-1) + (x-1)^2 - (x-1)^3 + (x-1)^4 - \cdots.$$

The radius of convergence is $R = 1$ since the expansion point is $x = 1$ and a singularity occurs at $x = 0$. Therefore, the series converges for $|x-1| < 1$, or equivalently when $0 < x < 2$. In other words, this Taylor series can not be used to approximate $f(x) = 1/x$ at $x = 3$ unlike the Lagrange.

Implementing a Lagrange Polynomial is easy if we utilize Sympy's `sympy.polys` module for polynomial algebras, and specifically its `interpolating_poly()` function (`https://docs.sympy.org/latest/modules/polys/reference.html`). It generates a Lagrange when supplied with n (x, y) data points, as shown in lagSymHyp.py, which replicates our $1/x$ example:

```
x = sp.symbols('x')
xs = [2, 2.75, 4]   # 0.3, 0.5
ys = [1/v for v in xs]
ipoly = interpolating_poly(len(xs), x, X=xs, Y=ys)
```

```
print("Poly:"); print(ipoly); print()

p = sp.expand(ipoly)   # multiply out all the terms
print("Expanded:", p)
print("Factored:, sp.factor(p))
print("Using Fractions:", sp.nsimplify(p))

# Convert to a Python function
pFunc = sp.lambdify(x, p, "math")
print("P(3) approx:", pFunc(3))
```

The output:

```
Poly:
0.333333333333333*(x - 4)*(x - 2.75) -
0.387878787878788*(x - 4)*(x - 2) +
0.1*(x - 2.75)*(x - 2)
Expanded:
0.0454545454545455*x**2 - 0.397727272727273*x + 1.11363636363636
Factored:
1.11363636363636*(0.0408163265306122*x**2 - 0.357142857142857*x + 1.0)
Using Fractions:
x**2/22 - 35*x/88 + 49/44

P(3) approx: 0.3295454545454505
```

lagSymHyp.py goes on to use the polynomial to generate a series of coordinates between $x = 1/10$ and 5, and plots them against $1/x$, as shown in Fig. 20.18.

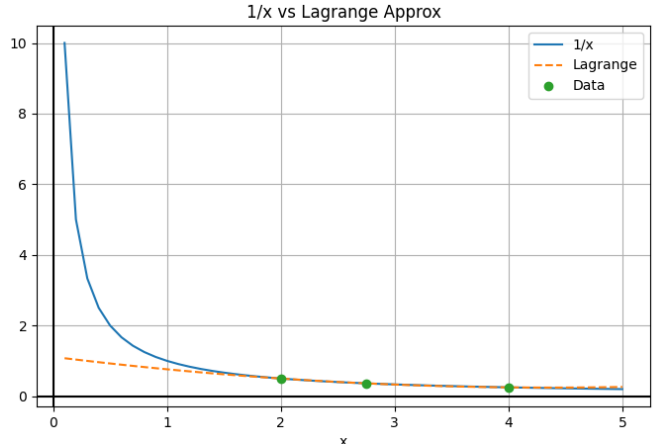


Figure 20.18. Lagrange Polynomial Plot for $1/x$ With 3 Data Points

The polynomial is a good fit within the interpolation range ($x_0 = 2$ to $x_2 = 4$), and passes through those points as expected. However, care must be taken when using the approximation outside the $[2, 4]$ interval, and this becomes clearer if we add $x = 0.3$ and 0.5, which produces the plot in Fig. 20.19.

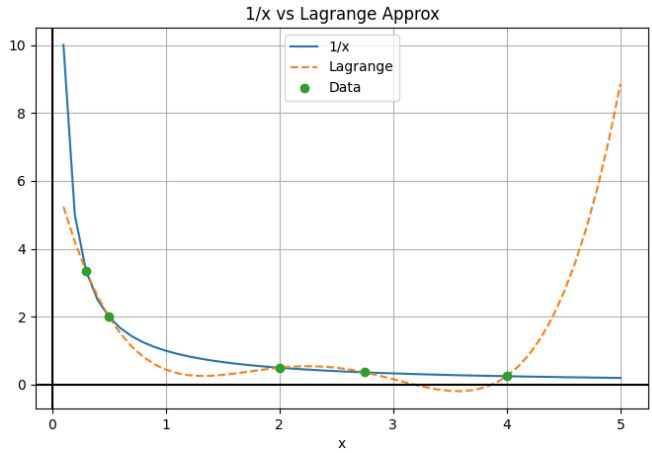


Figure 20.19. Lagrange Polynomial Plot for $1/x$ with 5 Data Points

The polynomial is now of degree 4:

```
10*x**4/33 - 191*x**3/66 + 95909090909091*x**2/10000000000000 -
130037878787879*x/10000000000000 + 64469696969697/10000000000000
```

Aside from the potential for large errors outside the interpolation range, the quality of the results *within* that range have also decreased. This can be remedied by adding more data points, but deciding how many to add and where they should be placed is tricky. Indeed, it may be better to simply return to the first polynomial, and accept a smaller range. This complexity is one reason for preferring Chebyshev polynomial approximation (see the next section), which avoids this problem.

20.8.2 The Lagrange Remainder Term. It's useful to be able to calculate an error term for a Lagrange polynomial.

Let $x_0, x_1, \ldots, x_n$ be numbers in the interval $[a, b]$, then for each x, a z exists where

$$f(x) = P(x) + \frac{f^{(n+1)}(z)}{(n+1)!}(x - x_0)(x - x_1)\ldots(x - x_n),$$

where $P(x)$ is the Lagrange polynomial, and the remainder is the error (called $R(x)$). It's similar to the error form we saw for the Taylor series:

$$\frac{f^{(n+1)}(z)}{(n+1)!}(x-x_0)^{n+1}.$$

Let's use the Lagrange remainder to find the maximum error in the $[2,4]$ interval for the $1/x$ approximation. We have

$$f'(x) = -x^{-2}, \quad f''(x) = 2x^{-3}, \text{ and } f'''(x) = -6x^{-4}.$$

Therefore,

$$\begin{aligned} R(x) &= \frac{f'''(z)}{3!}(x-x_0)(x-x_1)(x-x_2) \\ &= -z^{-4}(x-2)(x-2.75)(x-4), \text{ for } z \text{ in } [2,4]. \end{aligned}$$

The maximum value for z^{-4} in the interval is $2^{-4} = 1/16$, but what about the expression:

$$(x-2)(x-2.75)(x-4) = x^3 - \frac{35}{4}x^2 + \frac{49}{2}x - 22.$$

The cubic's critical points occur at $x = 7/3$ and $7/2$ as 'confirmed' by Fig. 20.20.

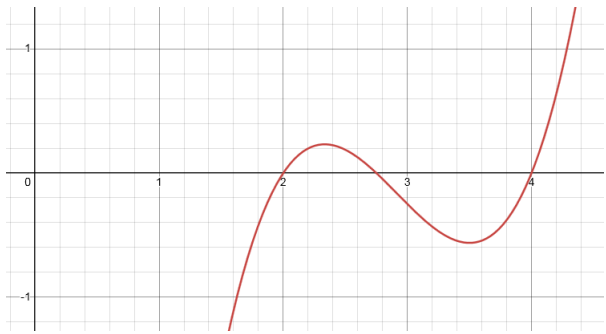


Figure 20.20. Critical Points in $(x-2)(x-2.75)(x-4)$

The corresponding y-values are $25/108$ and $-9/16$, with the latter being bigger in magnitude. Combining these results,

$$\begin{aligned} R(x) &= \left|\frac{f'''(\xi(x))}{3!}(x-x_0)(x-x_1)(x-x_2)\right| \leq \frac{1}{16\cdot 6}\left|-\frac{9}{16}\right| \\ &= \frac{3}{512} \approx 0.00586. \end{aligned}$$

20.8.3 Approximating $\sin(x)$. Let $f(x) = \sin(x)$, $n = 3$, and $x_k = k\pi/3$ for $k = 0, 1, 2, 3$. The $L_{n,k}$ values are:

$$\begin{aligned} L_{3,0}(x) &= \frac{x-(\pi/3)}{0-(\pi/3)} \cdot \frac{x-(2\pi/3)}{0-(2\pi/3)} \cdot \frac{x-\pi}{0-\pi} \\ &= \frac{1}{2\pi^3}(\pi - 3x)(2\pi - 3x)(\pi - x) \end{aligned}$$

$$L_{3,1}(x) = \frac{9}{2\pi^3}x(2\pi - 3x)(\pi - x)$$

$$L_{3,2}(x) = -\frac{9}{2\pi^3}x(\pi - 3x)(\pi - x)$$

$$L_{3,3}(x) = \frac{1}{2\pi^3}x(\pi - 3x)(2\pi - 3x)$$

Utilizing these in the Lagrange polynomial:

$$\begin{aligned} P(x) &= \sin(0) \cdot L_{3,0}(x) + \sin(\pi/3) \cdot L_{3,1}(x) + \sin(2\pi/3) \cdot L_{3,2}(x) + \sin(\pi) \cdot L_{3,3}(x) \\ &= \frac{\sqrt{3}}{2} \cdot \frac{9}{2\pi^3}x(2\pi - 3x)(\pi - x) - \frac{\sqrt{3}}{2} \cdot \frac{9}{2\pi^3}x(\pi - 3x)(\pi - x) \\ &= \frac{9\sqrt{3}}{4\pi^2}x(\pi - x) \end{aligned}$$

The error bound is:

$$\begin{aligned} |R(x)| &= \left|\frac{\sin z}{4!}(x-0)\left(x-\frac{\pi}{3}\right)\left(x-\frac{2\pi}{3}\right)(x-\pi)\right| \\ &\le \frac{1}{24}\max_{0\le x\le\pi}\left|x\left(x-\frac{\pi}{3}\right)\left(x-\frac{2\pi}{3}\right)(x-\pi)\right| \\ &= \frac{1.20258}{24} = 0.050108 \end{aligned}$$

lagSymSine.py implements this polynomial in code:

```
xs = [k*sp.pi/3 for k in range(4)]
ys = [sp.sin(v) for v in xs]
ipoly = interpolating_poly(len(xs), x, X=xs, Y=ys)
```

and its factored form is reported as:

```
-9*sqrt(3)*x*(x - pi)/(4*pi**2)
```

which agrees with our calculations.

The code also plots the polynomial against $\sin(x)$ in the interval $[0, \pi]$ and draws a second graph showing how the absolute error varies between the two curves (see Fig. 20.21).

The error plot shows that the calculated maximum error is a little too large since the polynomial's error tops out at around 0.045.

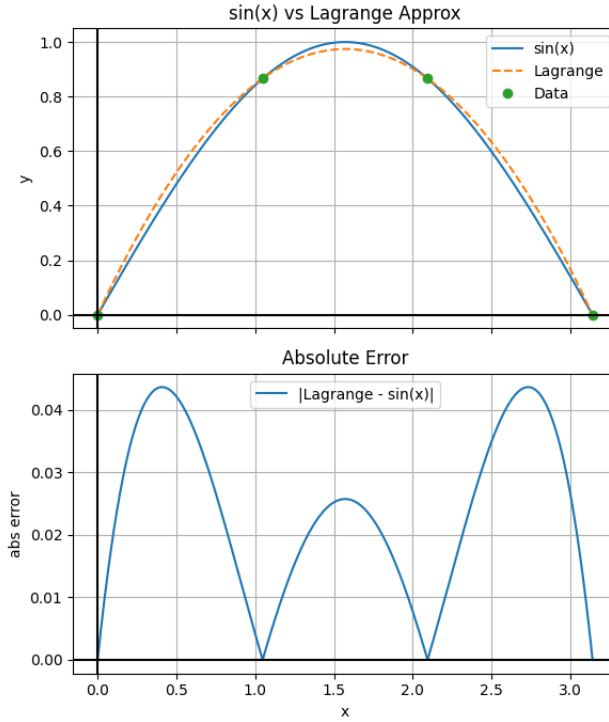


Figure 20.21. Lagrange Polynomial Plot for $sin(x)$

20.8.4 Lagrange Approximation Without Sympy. If there's no need for a polynomial in algebraic form, then it's possible to calculate the polynomial's result using summations over $L_{n,k}()$s expressed as products. This approach is implemented by `lagrangeInterp()` in faUtils.py:

```
def lagrangeInterp(xs, ys, x):
  # compute the Lagrange polynomial at x
  n = len(xs)
  total = 0.0
  for k in range(n):
    term = ys[k]
    for j in range(n):  # 0 to n except for k
      if k != j:
        term *= (x - xs[j]) / (xs[k] - xs[j])
    total += term
  return total
```

This function is used by lagrangeSine.py to replicate the sin(x) approximation in lagSymSine.py. The crucial lines are:

```
xs = [k*math.pi/3 for k in range(4)]
ys = [math.sin(x) for x in xs]
xPlot = faUtils.linspace(0, math.pi, 200)
yPlot = [faUtils.lagrangeInterp(xs, ys, x) for x in xPlot]
```

20.9 Orthogonality

Least squares (see section 6.8) can fit a function approximation to a collection of data points, but the matrix that needs to be solved can be prone to rounding errors. However, if we assume that the polynomial approximation is *orthogonal* then the calculations become much simpler and less likely to produce errors. Orthogonality is the core element that makes the Chebyshev approximation of the next section so successful, so we'll spend a little time here explaining the concept.

Suppose $f(x)$ is to be approximated by a polynomial $P_n(x)$ of at most n degree across the x range $[a, b]$. We want to minimize the least squares error

$$\int_a^b [f(x) - P_n(x)]^2 \, dx.$$

Let

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = \sum_{k=0}^{n} a_k x^k,$$

where we want to determine $a_0, a_1, \cdots, a_n$ subject to error minimization

$$E \equiv E_2(a_0, a_1, \ldots, a_n) = \int_a^b \left(f(x) - \sum_{k=0}^{n} a_k x^k \right)^2 dx.$$

A necessary condition is that

$$\frac{\partial E}{\partial a_j} = 0, \quad \text{for each } j = 0, 1, \ldots, n.$$

After expanding the right-hand side of E:

$$E = \int_a^b [f(x)]^2 \, dx - 2 \sum_{k=0}^{n} a_k \int_a^b x^k f(x) \, dx + \int_a^b \left(\sum_{k=0}^{n} a_k x^k \right)^2 dx,$$

we partially differentiate it with respect to each a_j:

$$\frac{\partial E}{\partial a_j} = 0 = -2 \int_a^b x^j f(x) \, dx + 2 \sum_{k=0}^{n} a_k \int_a^b x^{j+k} \, dx.$$

Hence, to find $P_n(x)$, we solve $(n+1)$ a_k equations involving $(n+1)$ j powers:

$$\sum_{k=0}^{n} a_k \int_a^b x^{j+k} \, dx = \int_a^b x^j f(x) \, dx, \quad \text{for each } j = 0, 1, \ldots, n.$$

Let's define the right-hand sides as:

$$b_j = \int_a^b x^j f(x)\,dx, \quad \text{for each } j = 0, 1, \ldots, n.$$

and evaluate the integrals on the left-hand sides. The resulting system can be written as a linear system of the form

$$\left[\frac{b^{k+j+1} - a^{k+j+1}}{k+j+1}\right]_{k,j=0}^{n} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}$$
$$C \qquad \mathbf{a} \quad = \mathbf{b}$$

The $n+1$ square matrix C is a *Hilbert matrix*, and a classic example of how round-off errors affect results.

20.9.1 Linearly Independent Functions. Fortunately there's a different technique for obtaining least squares approximations that turns out to be much more computationally efficient. To facilitate the discussion, we need some new concepts.

A set of functions $\{\phi_0, \ldots, \phi_n\}$ is said to be *linearly independent* over some range $[a, b]$ if, whenever

$$c_0\phi_0(x) + c_1\phi_1(x) + \cdots + c_n\phi_n(x) = 0, \quad \text{for all } x \in [a, b],$$

then we have $c_0 = c_1 = \cdots = c_n = 0$.

It can be proved that if $\phi_j(x)$ is a polynomial of degree j where $j = 0, 1, \ldots, n$, then $\{\phi_0, \ldots, \phi_n\}$ **is** linearly independent. Also, any other polynomial can be written as a linear combination of $\phi_0(x)$, $\phi_1(x)$, $\ldots$, $\phi_n(x)$. As a rough analogy, linearly independent functions are like the x, y, and z unit vectors in 3D space, and combinations of those vectors can define any other vector.

As an example, consider the set $\beta = \{1, x+1, x^2+1\}$, which is linearly independent because the only solution to $c_1(1) + c_2(x+1) + c_3(x^2+1) = 0$ is $c_1 = c_2 = c_3 = 0$. Now write some arbitrary polynomial, $p(x) = 2x^2 + 3x + 1$, as a linear combination of the elements of β:

$$\begin{aligned} 2x^2 + 3x + 1 &= a_0(1) + a_1(x+1) + a_2(x^2+1) \\ &= (a_0 + a_1 + a_2) + a_1 x + a_2 x^2, \end{aligned}$$

and so $a_2 = 2$, $a_1 = 3$, and $a_0 = -4$.

20.9.2 Orthogonal Functions. Now we introduce weight functions, and define orthogonality.

The purpose of a weight function w is to assign varying degrees of importance to approximations over portions of the x interval. For example,

$$w(x) = \frac{1}{\sqrt{1-x^2}}$$

places less emphasis near the center of the interval $(-1, 1)$ and more emphasis when $|x|$ is near 1.

Suppose $\{\phi_0, \phi_1, \ldots, \phi_n\}$ is a set of linearly independent functions on $[a, b]$ and w is a weight function for $[a, b]$. We want to linearly combine these functions to define a polynomial $P(x)$

$$P(x) = \sum_{k=0}^{n} a_k \phi_k(x)$$

while also minimizing the error

$$E = E(a_0, \ldots, a_n) = \int_a^b w(x) \left[f(x) - \sum_{k=0}^{n} a_k \phi_k(x) \right]^2 dx.$$

This problem reduces to the situation considered at the beginning of this section when $w(x) = 1$ and $\phi_k(x) = x^k$, for each $k = 0, 1, \ldots, n$.

This reduces to the series of partial derivatives like the ones from earlier, except that $w(x)$ is now included:

$$\frac{\partial E}{\partial a_j} = 2 \int_a^b w(x) \left[f(x) - \sum_{k=0}^{n} a_k \phi_k(x) \right] \phi_j(x)\, dx.$$

Rearranging the equations gives

$$\int_a^b w(x) f(x) \phi_j(x)\, dx = \sum_{k=0}^{n} a_k \int_a^b w(x) \phi_k(x) \phi_j(x)\, dx, \quad \text{for } j = 0, 1, \ldots, n.$$

Now is the time to introduce orthogonality, which will allow us to avoid solving a $n+1$-square matrix in the next step. Orthogonality is the principle that the functions $\phi_0, \phi_1, \ldots, \phi_n$ can be chosen so that

$$\int_a^b w(x) \phi_k(x) \phi_j(x)\, dx = \begin{cases} 0, & \text{when } j \neq k, \\ \alpha_j > 0, & \text{when } j = k, \end{cases}$$

This permits the equations to be greatly simplified, becoming

$$\int_a^b w(x) f(x) \phi_j(x)\, dx = a_j \int_a^b w(x) [\phi_j(x)]^2\, dx = a_j \alpha_j,$$

for each $j = 0, 1, \ldots, n$, and are easily solved

$$a_j = \frac{1}{\alpha_j} \int_a^b w(x) f(x) \phi_j(x)\, dx.$$

A rough analogy of orthogonality is the dot product of vectors (see section 10.3). If two vectors are orthogonal (i.e. separated by 90°), then their dot product will be 0. This kind of property is desirable when choosing a set of linearly independent vectors to act as axes in 3D space.

Orthogonality is more restrictive than linear independence. As an example consider the following set of polynomials, which are linearly independent but not orthogonal:

$$p_0(x) = 1, \qquad p_1(x) = x, \qquad p_2(x) = x^2.$$

First, let's show that they're linearly independent. Suppose there are constants a, b, and c such that

$$a\,p_0(x) + b\,p_1(x) + c\,p_2(x) = 0$$

for all x. Substituting the polynomials into this equation gives $a + bx + cx^2 = 0$. The polynomial is zero only if all of its coefficients are zero, so the set $\{1, x, x^2\}$, is linearly independent. However, for a set to be orthogonal, every distinct pair must have an *inner product* $\langle p_i, p_j \rangle = 0$, where

$$\langle p_i, p_j \rangle = \int_{-1}^{1} p_i(x) p_j(x)\,dx.$$

Let's check this for 1 and x:

$$\langle 1, x \rangle = \int_{-1}^{1} x\,dx = 0.$$

What about x and x^2?

$$\langle x, x^2 \rangle = \int_{-1}^{1} x^3\,dx = 0.$$

But for 1 and x^2,

$$\langle 1, x^2 \rangle = \int_{-1}^{1} x^2\,dx = \left[\frac{x^3}{3}\right]_{-1}^{1} = \frac{2}{3},$$

so the set isn't orthogonal because of this pair.

It's not difficult to modify the set's elements, specifically x^2, so that the set becomes orthogonal. Let's define $p_2(x) = x^2 + a$, and once again check the inner products, but with the aim of choosing a value for a that makes the set orthogonal.

Test $\langle 1, x^2 + a \rangle$,

$$0 = \int_{-1}^{1} (x^2 + a)\,dx = \int_{-1}^{1} x^2\,dx + a \int_{-1}^{1} 1\,dx.$$

Evaluating the integrals,

$$0 = \frac{2}{3} + 2a.$$

Hence, $a = -1/3$, making $p_2(x) = x^2 - 1/3$, and the set becomes $\{1,\ x,\ x^2 - 1/3\}$. Its orthogonality should be checked:

$$\begin{aligned}\langle 1, x\rangle &= \int_{-1}^{1} x\,dx = 0.\\ \langle 1,\ x^2 - 1/3\rangle &= \int_{-1}^{1} \left(x^2 - 1/3\right)\,dx = 2/3 - 2/3 = 0.\\ \langle x,\ x^2 - 1/3\rangle &= \int_{-1}^{1} \left(x^3 - \frac{x}{3}\right)\,dx = 0.\end{aligned}$$

Since all of the pairwise inner products vanish, the set is orthogonal.

20.9.3 The Orthogonality of $\cos 2\theta$ and $\cos 5\theta$. Consider the inner product of these two functions in the interval $[0, 2\pi]$, which we'll show is equal to 0:

$$\int_0^{2\pi} \cos 2\theta\, cos5\theta\, d\theta = 0.$$

Rewrite both functions as polynomials in $\cos\theta$ by the use of multiple-angle identities,

$$\cos 2\theta = 2\cos^2\theta - 1$$

and

$$\cos 5\theta = 16\cos^5\theta - 20\cos^3\theta + 5\cos\theta.$$

Therefore,

$$\int_0^{2\pi} \cos 2\theta \cos 5\theta\, d\theta = \int_0^{2\pi} (2\cos^2\theta - 1)(16\cos^5\theta - 20\cos^3\theta + 5\cos\theta)\, d\theta.$$

Expanding,

$$\begin{aligned}&= \int_0^{2\pi} \Big(32\cos^7\theta - 40\cos^5\theta + 10\cos^3\theta\\ &\qquad - 16\cos^5\theta + 20\cos^3\theta - 5\cos\theta\Big)\,d\theta\\ &= \int_0^{2\pi} \left(32\cos^7\theta - 56\cos^5\theta + 30\cos^3\theta - 5\cos\theta\right)\,d\theta.\end{aligned}$$

Each term contains an odd power of $\cos\theta$:

$$\cos\theta, \quad \cos^3\theta, \quad \cos^5\theta, \quad \cos^7\theta$$

and for any odd power

$$\int_0^{2\pi} \cos^{2k+1}\theta\, d\theta = 0,$$

because the positive and negative halves cancel over a full period. Hence,

$$32\int_0^{2\pi} \cos^7\theta\, d\theta - 56\int_0^{2\pi} \cos^5\theta\, d\theta + 30\int_0^{2\pi} \cos^3\theta\, d\theta - 5\int_0^{2\pi} \cos\theta\, d\theta = 0.$$

Therefore, the integral equals 0 and so $\cos 2\theta$ and $\cos 5\theta$ are orthogonal. This is confirmed by ortho.pdf which calculates the dot product of multiple $\cos(2\theta)$ and $\cos(5\theta)$ values in the interval $[0, 2\pi]$:

```
n = 2; m = 5
nPts = 1000
thetas = [ (2*math.pi*i)/nPts for i in range(nPts)]
cosN = [math.cos(n*t) for t in thetas]
cosM = [math.cos(m*t) for t in thetas]
result = sum([a*b for a, b in zip(cosN, cosM)])
print(f"Dot product = {result:7f}")
```

Our reason for including this example is the importance of this cosine form for proving the orthogonality of Chebyshev polynomials in the next section (specifically subsection 20.10.2).

20.10 Chebyshev Approximation

The Chebyshev polynomial of degree n is defined as $T_n(x) = \cos(n \arccos(x))$, which, perhaps surprisingly, can be reexpressed with the help of trigonometric identities as powers of x,

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ &\cdots \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \quad n \geq 1. \end{aligned}$$

Crucially for their role in function approximation, these polynomials are orthogonal in the interval $[-1, 1]$ and, when combined with the weight function $1/\sqrt{1-x^2}$, simplify to:

$$\int_{-1}^{1} \frac{T_i(x)T_j(x)}{\sqrt{1-x^2}}\, dx = \begin{cases} 0 & i \neq j \\ \pi/2 & i = j \neq 0 \\ \pi & i = j = 0 \end{cases}$$

20.10.1 Deriving the $T_n(x)$ Polynomials.

Note that

$$T_0(x) = \cos(0) = 1 \quad \text{and} \quad T_1(x) = \cos(\arccos(x)) = x.$$

For $n \geq 1$, we introduce the substitution $\theta = \arccos(x)$ so that

$$T_n(x) = \cos(n \arccos(x)) = \cos(n\theta), \quad \text{where } \theta \in [0, \pi].$$

The recurrence relation is derived by noting that

$$T_{n+1}(x) = \cos((n+1)\theta) = \cos(\theta)\cos(n\theta) - \sin(\theta)\sin(n\theta)$$

and

$$T_{n-1}(x) = \cos((n-1)\theta) = \cos(\theta)\cos(n\theta) + \sin(\theta)\sin(n\theta).$$

Adding these gives us

$$T_{n+1}(x) + T_{n-1}(x) = 2\cos(\theta)\cos(n\theta).$$

or

$$T_{n+1}(x) = 2\cos(\theta)\cos(n\theta) - T_{n-1}(x).$$

Replacing $\cos(\theta)$ by x and θ by $\arccos(x)$, we have for $n \geq 1$,

$$T_{n+1}(x) = 2x\cos(n\arccos(x)) - T_{n-1}(x),$$

that is,

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).$$

Note that $\deg(T_n) = n$, and the recursion relation shows that the leading coefficient of the right-hand side of $T_n(x)$ is always 2^{n-1}.

20.10.2 Proving Orthogonality. To show the orthogonality of the Chebyshev polynomials, consider

$$\int_{-1}^{1} \frac{T_n(x)T_m(x)}{\sqrt{1-x^2}}\,dx = \int_{-1}^{1} \frac{\cos(n\arccos(x))\cos(m\arccos(x))}{\sqrt{1-x^2}}\,dx.$$

Let's differentiate $\theta = \arccos(x)$:

$$d\theta = -\frac{1}{\sqrt{1-x^2}}\,dx$$

and so

$$\int_{-1}^{1} \frac{T_n(x)T_m(x)}{\sqrt{1-x^2}}\,dx = -\int_{\pi}^{0} \cos(n\theta)\cos(m\theta)\,d\theta = \int_{0}^{\pi} \cos(n\theta)\cos(m\theta)\,d\theta.$$

This integral is a more general form of our section 20.9.3 example.

Now suppose that $n \neq m$, and since

$$\cos(n\theta)\cos(m\theta) = \frac{1}{2}\left[\cos(n+m)\theta + \cos(n-m)\theta\right],$$

we have

$$\begin{aligned}\int_{-1}^{1} \frac{T_n(x)T_m(x)}{\sqrt{1-x^2}}\,dx &= \frac{1}{2}\int_{0}^{\pi} \cos(n+m)\theta\,d\theta + \frac{1}{2}\int_{0}^{\pi} \cos(n-m)\theta\,d\theta \\ &= \left[\frac{1}{2(n+m)}\sin(n+m)\theta + \frac{1}{2(n-m)}\sin(n-m)\theta\right]_0^{\pi} = 0.\end{aligned}$$

By a similar technique, we can also show

$$\int_{-1}^{1} \frac{[T_n(x)]^2}{\sqrt{1-x^2}}\,dx = \frac{\pi}{2}, \quad \text{for each } n \geq 1.$$

You may want to check this, and also the $n = 0$ case.

The Chebyshev polynomials also satisfy a discrete form of orthogonality: if x_k ($k = 1, \ldots, n$) are the n zeros of $T_n(x)$, and if $i, j < n$, then

$$\sum_{k=1}^{n} T_i(x_k)T_j(x_k) = \begin{cases} 0 & i \neq j \\ n/2 & i = j \neq 0 \\ n & i = j = 0 \end{cases}$$

20.10.3 Zeros and Extrema. Two important properties of the Chebyshev polynomials are the distribution of their zeros, and their maximums and minimums.

$T_n(x)$ will always have n zeros in the interval $[-1, 1]$, located at the points

$$x_k = \cos(\frac{\pi(k - 1/2)}{n}) \qquad k = 1, 2, \ldots, n$$

This follows from the fact that all the zeros of $T_n(x)$ are solutions for $0 = \cos(n \arccos(x))$, and $\cos(y) = 0$ if and only if $y =$ odd integer $\cdot (\pi/2)$.

There will also always be $n + 1$ maxima and minima at

$$x_j = \cos(\frac{\pi j}{n}) \qquad j = 0, 1, \ldots, n$$

$T_n(x) = 1$ at every maxima , and -1 at the minima, which follows from how $T_n(x) = \cos(y)$ for some y.

These characteristics are visualized in Fig. 20.22 which plots $T_0(x) - T_5(x)$. The graph also shows that the roots are more numerous at the two ends of the $[-1, 1]$ range, which makes Chebyshev polynomials better suited for dealing with the oscillations that appear in some function approximations.

Fig. 20.22 was generated by chebPolys.py as a byproduct of it generating the equations for the first six Chebyshev polynomials:

```
T0(x) = 1
T1(x) = x
T2(x) = 2*x**2 - 1
T3(x) = 4*x**3 - 3*x
T4(x) = 8*x**4 - 8*x**2 + 1
T5(x) = 16*x**5 - 20*x**3 + 5*x
```

The code contains two ways to generate these polynomials, both of which use Sympy for algebraic manipulation. `chebyshevIters()` employs the Chebyshev recurrence relation:

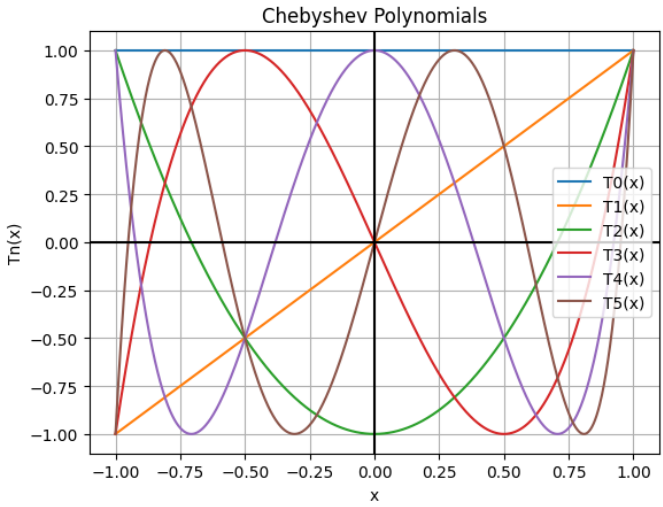


Figure 20.22. Plots of the Chebyshev Polynomials $T_0(x) - T_5(x)$

```
def chebyshevIters(n):
  # the nth Chebyshev polynomial in x, T_n(x)
  if n == 0:
    return 1
  if n == 1:
    return x

  # implement T_n(x) = 2*x*T_{n-1}(x) - T_{n-2}(x)
  t0 = 1
  t1 = x
  for _ in range(2, n+1):
    t2 = sp.expand(2*x*t1 - t0)
      # expand the polynomial expression
    t0 = t1
    t1 = t2
  return t1
```

The second approach is to simply call Sympy's `chebyshevt()`:

```
for n in range(6):
  # the nth Chebyshev polynomial in x, T_n(x)
  # cPoly = chebyshevIters(n)
  cPoly = sp.chebyshevt(n, x)
  print(f"T{n}(x) = {cPoly}")
```

20.10.4 Using the Chebyshev Polynomials. Our aim is to approximate a function $f(x)$ in the interval $[-1, 1]$ in terms of a summation of n $T_j(x)$ Chebyshev polynomials ($j = 0, 1, ..., n-1$) multiplied by coefficients c_j that we have to determine. The resulting $(n-1)$-degree approximation for $f(x)$ has the form:

$$f(x) \approx \frac{1}{2}c_0 + \sum_{j=1}^{n-1} c_j T_j(x)$$

The coefficients c_j are defined by

$$\begin{aligned} c_j &= \tfrac{2}{n}\textstyle\sum_{k=1}^{n} f(x_k)T_j(x_k) \\ &= \tfrac{2}{n}\textstyle\sum_{k=1}^{n} f\Big[\cos\Big(\frac{\pi(k-1/2)}{n}\Big)\Big]\cos\Big(\frac{\pi j(k-1/2)}{n}\Big) \end{aligned}$$

Note that we're using the n zeros, x_k, of the polynomial, which means that we're only able to solve for n c_j coefficients.

The c_j equation is derived using $T_n(x) = \cos(n \arccos(x))$ and the n zeros located at

$$x_k = \cos\left(\frac{\pi\,(k-1/2)}{n}\right), \qquad k = 1, 2, \ldots, n$$

Consider the kth zero in the $T_j(x)$ polynomial:

$$T_j(x_k) = \cos\left(j \arccos\left[\cos\left(\frac{\pi\,(k-1/2)}{n}\right)\right]\right)$$

so that

$$T_j(x_k) = \cos\left(\frac{\pi\, j\,(k-1/2)}{n}\right).$$

Let's assume that $f(x)$ can be approximated by some combination of Chebyshev polynomials such that:

$$f(x) \approx \sum_{j=0}^{n-1} a_j T_j(x).$$

Focus on the kth zero:

$$f(x_k) = \sum_{j=0}^{n-1} a_j T_j(x_k).$$

Multiply both sides by $T_i(x_k)$, where $i = 0, 1, ..., n-1$:

$$f(x_k)T_i(x_k) = \sum_{j=0}^{n-1} a_j T_j(x_k)T_i(x_k).$$

Now sum over the zero's range $k = 1, ..., n$:

$$\sum_{k=1}^{n} f(x_k)T_i(x_k) = \sum_{j=0}^{n-1} a_j \sum_{k=1}^{n} T_j(x_k)T_i(x_k).$$

The payoff of having the Chebyshev polynomials be orthogonal has arrived, since all the $T_j(x) \cdot T_i(x)$ multiples vanish except when $i = j$. There are two cases to consider:

Case 1: $i = j \neq 0$.

$$\sum_{k=1}^{n} f(x_k)T_j(x_k) = a_j \frac{n}{2}.$$

Hence

$$a_j = \frac{2}{n} \sum_{k=1}^{n} f(x_k)T_j(x_k) \qquad (j \neq 0).$$

Case 2: $i = j = 0$. Since $T_0(x) = 1$ we have

$$\sum_{k=1}^{n} f(x_k) = a_0 n.$$

Therefore

$$a_0 = \frac{1}{n} \sum_{k=1}^{n} f(x_k).$$

We use these a_j's to calculate the Chebyshev c_j coefficients. For $j = 0$,

$$a_0 = \frac{1}{n} \sum_{k=1}^{n} f(x_k) \qquad \text{and} \qquad c_0 = \frac{2}{n} \sum_{k=1}^{n} f(x_k).$$

Therefore, $a_0 = \frac{1}{2}c_0$, while for $j \geq 1$, $a_j = c_j$.

Starting from

$$f(x) \approx \sum_{j=0}^{n-1} a_j T_j(x),$$

replace a_0 by $\frac{1}{2}c_0$ and a_j by c_j for $j \geq 1$ to produce:

$$f(x) \approx \frac{1}{2}c_0 + \sum_{j=1}^{n-1} c_j T_j(x).$$

20.10.5 Reducing the Degree of the Approximation. Consider what happens if we truncate the approximation to the mth term:

$$f(x) \approx \frac{1}{2}c_0 + \sum_{j=1}^{m-1} c_j T_j(x).$$

Since the $T_j(x)$'s are bounded between ± 1, the loss can be no larger than the sum of the discarded c_j's. In fact, if the c_j's are rapidly decreasing (which is the typical case), then the error is dominated by $c_m T_m(x)$, an oscillatory function with $m+1$ extrema distributed smoothly over the interval $[-1, 1]$.

20.10.6 Implementing Chebyshev Fitting and Evaluation. We need code to obtain the c_j coefficients, and also to evaluate the approximation using those coefficients and $T_j(x)$.

For the first task, we've modified a function presented in Section 5.8 of Press et. al's *Numerical Recipes in C* [**PTVF92**]. `chebFit()` in faUtils.py allows the approximation range to be between arbitrary limits a and b, instead of just -1 to 1. This is effected by a change of variable

$$y = \frac{x - 1/2(b+a)}{1/2(b-a)}$$

The rest of the code implements the summation expressions for calculating c_j:

```
def chebFit(a, b, nTerms, f):
  coefs = [0]*nTerms
  scale = (b-a)/2
  offset = (a+b)/2

  # sample f() at Chebyshev zeros
  fzs = [ f(offset + math.cos(math.pi*(k+0.5)/nTerms) * scale)
                          for k in range(nTerms) ]
          # 0 to nTerms-1 == index 1 to n in the math
  fac = 2/nTerms

  # calculate nTerms Chebyshev coefs c_0 to c_n-1
  for j in range(nTerms):
    tot = 0
    for k in range(nTerms):
      # 0 to nTerms-1 == index 1 to n in the math
      tot += fzs[k] * math.cos(math.pi*j*(k+0.5)/nTerms)
    coefs[j] = fac * tot
  coefs[0] *= 0.5
  return coefs
```

`chebEval()` in faUtils.py calculates the $f(x)$ approximation. It generates each Chebyshev polynomial using the recurrence relation $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$ inside a loop:

```
def chebEval(a, b, coefs, nTerms, x):
  if (x-a)*(x-b) > 0:
    print("x not in range in chebEval()")
    sys.exit(1)
  xv = (2*x - (b+a))/(b-a)     # map x onto [-1,1]

  t0 = 1  # T_0(x) = 1
  approx = coefs[0] * t0
  if nTerms == 0:
    return approx
```

```
  t1 = xv   # T_1(x) = x
  approx += coefs[1] * t1
  for k in range(2, nTerms):
    # T_n(x) = 2*x*T_{n-1}(x) - T_{n-2}(x)
    t2 = 2*xv*t1 - t0
    approx += coefs[k] * t2
    t0 = t1      # shift vars
    t1 = t2
  return approx
```

This is not quite the same approach as used by Press *et. al* [**PTVF92**] who employ Clenshaw's recurrence.

The two functions are tested in chebTest.py by generating an 8-term approximation for $f(x) = x(x-2)\sin(x)$ in the range $[-\pi/2, \pi/2]$:

```
nTerms = 8   # try 6, 8, 10
a = -math.pi/2; b = math.pi/2
print(f"Evaluating f(x) using {nTerms} terms in [{a},{b}]:")
coefs = faUtils.chebFit(a, b, nTerms, f)
faUtils.printCoefs(coefs)

print(f"    x            actual       chebyshev     abs error")
for i in range(-4, 5):
  x = i * math.pi/2 / 10
  chebx = faUtils.chebEval(a, b, coefs, nTerms, x)
  print(f"{x:12.6f} {f(x):12.6f} {chebx:12.6f}  {abs(f(x) - chebx):12.6f}")
```

The approximation's results are correct to two or more decimal places, and improve with a larger nTerms setting.

chebRunge.py generates three Chebyshev polynomials that fits $f(x) = 1/(1+25x^2)$ in the range $[-1, 1]$ using 6, 11, and 16 terms, as seen in Fig. 20.23. Recall that a variant of this function $(1/(1+x^2))$ posed problems for the Taylor series approximation back in section 20.7.4, and this version is a well-known difficult case for approximations such as Lagrange that have a tendency to oscillate.

Oscillation also occur with the Chebyshev polynomial, but the fact that the approximation's zeros are concentrated at the two ends of the interval helps to reduce the size of the problem.

20.11 Padé Approximation

Polynomial approximations can model most functions, are easily evaluated, and give good results. However, a polynomial can't easily represent poles or steep local behavior, and also has a tendency to oscillate.

Rational approximations, such as the Padé, perform well near singularities, can often approximate functions accurately with fewer terms than polynomial

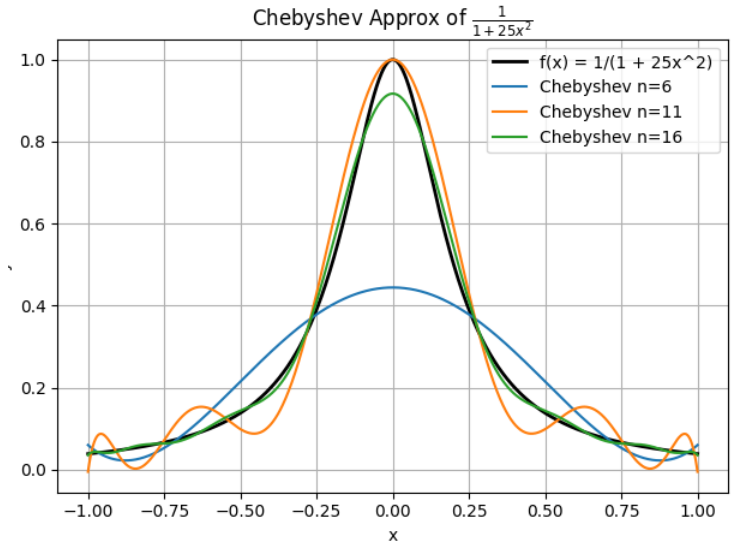


Figure 20.23. Three Approximations to $1/(1+25x^2)$

versions, and avoid oscillations by spreading errors more evenly over the expansion interval. On the downside, their generation requires the solving of a linear system, and they may be less stable outside the interval.

Suppose we are given a Taylor power series $\sum_{k=0}^{\infty} a_k x^k$ representing a function $f(x)$. A Padé approximation is a rational

$$r(x) = \frac{p(x)}{q(x)} = \frac{p_0 + p_1 x + \cdots + p_n x^n}{q_0 + q_1 x + \cdots + q_m x^m}$$

which has a expansion that agrees with the Taylor series based on the chosen (n, m) order. Although there are $n+1$ numerator and $m+1$ denominator coefficients in the rational, we can divide through by q_0, leaving just $n+m+1$ unknowns. This suggests that the rational ought to fit the power series through the orders $1, x, x^2, \ldots, x^{n+m}$, or in other words:

$$\sum_{k=0}^{\infty} a_k x^k = \frac{p_0 + p_1 x + \cdots + p_n x^n}{q_0 + q_1 x + \cdots + q_m x^m} + O(x^{n+m+1})$$

or

$$(q_0 + q_1 x + \cdots + q_m x^m)(a_0 + a_1 x + \ldots) = p_0 + p_1 x + \cdots + p_n x^n + O(x^{n+m+1}). \quad (1)$$

We'll assume that the term coefficients for x^{n+1} and higher on the right-hand side of equ. (1) equal 0. This lets us write the expressions on the left-hand side which

utilize the same powers of x as:

$$\begin{aligned} q_m a_{n-m+1} + q_{m-1} a_{n-m+2} + \cdots + q_0 a_{n+1} &= 0 \\ q_m a_{n-m+2} + q_{m-1} a_{n-m+3} + \cdots + q_0 a_{n+2} &= 0 \\ &\vdots \\ q_m a_n + q_{m-1} a_{n+1} + \cdots + q_0 a_{n+m} &= 0 \end{aligned}$$

The first equation is for terms involving x^{n+1}, the second for x^{n+2}, and so on up to x^{n+m}. Since $q_0 = 1$, the last term of these equations simplifies to an a coefficient, leaving m linear equations involving m q's. These can be expressed in matrix form as:

$$\begin{bmatrix} a_{n-m+1} & a_{n-m+2} & \cdots & a_n \\ a_{n-m+2} & a_{n-m+3} & \cdots & a_{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ a_n & a_{n+1} & \cdots & a_{n+m-1} \end{bmatrix} \begin{bmatrix} q_m \\ q_{m-1} \\ \vdots \\ q_1 \end{bmatrix} = - \begin{bmatrix} a_{n+1} \\ a_{n+2} \\ \vdots \\ a_{n+m} \end{bmatrix}$$

This linear system can be solved to find the q coefficients.

The rational's numerator values, $p_0, p_1, \ldots, p_n$ are obtained by equating the coefficients of $1, x, x^2, \ldots, x^n$ on the left- and right-sides of equ. (1):

$$\begin{aligned} a_0 &= p_0 \\ a_1 + q_1 a_0 &= p_1 \\ a_2 + q_1 a_1 + q_2 a_0 &= p_2 \\ &\vdots \\ a_n + \textstyle\sum_{k=1}^{\min(n,m)} q_k a_{n-k} &= p_n \end{aligned}$$

This is a linear system of $n+1$ unknowns in $n+1$ equations, so is solvable.

Often these two linear systems are combined into a single matrix:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & -a_0 & 0 & \cdots & 0 \\ & & \vdots & & & & \vdots & \\ 0 & 0 & \cdots & 1 & -a_n & -a_{n-1} & \cdots & -a_{n-m+1} \\ 0 & 0 & \cdots & 0 & -a_{n+1} & -a_n & \cdots & -a_{n-m+2} \\ 0 & 0 & \cdots & 0 & -a_{n+2} & -a_{n+1} & \cdots & -a_{n-m+3} \\ & & \vdots & & & & \vdots & \\ 0 & 0 & \cdots & 0 & -a_{n+m-1} & -a_{n+m-2} & \cdots & -a_n \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_n \\ q_1 \\ q_2 \\ \vdots \\ q_m \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \\ a_{n+1} \\ a_{n+2} \\ \vdots \\ a_{n+m} \end{bmatrix}$$

20.11.1 A Padé Approximation for e^{-x}. Consider the Maclaurin series:

$$e^{-x} = \sum_{i=0}^{\infty} \frac{(-1)^i}{i!} x^i.$$

We'll use this series to formulate a Padé approximation for e^{-x} of degree 5 with $n = 3$ and $m = 2$. We need values for p_0, p_1, p_2, p_3, q_1, and q_2 so that the coefficients of x^k for $k = 0, 1, \ldots, 5$ match in the expression:

$$\left(1 - x + \frac{x^2}{2} - \frac{x^3}{6} + \cdots\right)(1 + q_1 x + q_2 x^2) = (p_0 + p_1 x + p_2 x^2 + p_3 x^3).$$

Expanding and collecting terms:

$$\begin{aligned}
x^5: &\quad -\frac{1}{120} + \frac{1}{24}q_1 - \frac{1}{6}q_2 = 0 & \qquad x^2: &\quad \frac{1}{2} - q_1 + q_2 = p_2 \\
x^4: &\quad \frac{1}{24} - \frac{1}{6}q_1 + \frac{1}{2}q_2 = 0 & \qquad x^1: &\quad -1 + q_1 = p_1 \\
x^3: &\quad -\frac{1}{6} + \frac{1}{2}q_1 - q_2 = p_3 & \qquad x^0: &\quad 1 = p_0.
\end{aligned}$$

$p_0 = 1$ and there are five more variables to solve using the equations for x^1 to x^5. The solution is:

$$\left\{p_1 = -\tfrac{3}{5},\ p_2 = \tfrac{3}{20},\ p_3 = -\tfrac{1}{60},\ q_1 = \tfrac{2}{5},\ q_2 = \tfrac{1}{20}\right\}$$

so the approximation is

$$r(x) = \frac{1 - \frac{3}{5}x + \frac{3}{20}x^2 - \frac{1}{60}x^3}{1 + \frac{2}{5}x + \frac{1}{20}x^2}.$$

20.11.2 Implementing Padé. faUtils.py contains two Padé support functions – `padeApprox()` and `padeSolve()`. `padeApprox()` generates a power series for the given function, and passes it to `padeSolve()` to create the approximation:

```
def padeApprox(synfn, n, m, x0=0):
  print(f"\nUsing the series:")
  coefs = seriesCoefs(synfn, x0=x0, nTerms=n+m+1, verbose=True)
  p, q = padeSolve(n, m, coefs)
  # print the coefficients in p and q;
  # generate a Latex expression for the p/q rational
  #   :
  return p, q, latexFnm
```

The power series is obtained using `seriesCoefs()` which was described back in section 20.7.

`padeSolve()` is an implementation of the two-stage linear solving outlined above. Gaussian elimination (see section 6.8.4) solves the linear equations for the q coefficients, and the p coefficients are obtained by simple substitutions:

```
def padeSolve(n, m, coefs):
  # Build linear system for q1..qm (m equations)
  # since we know q0 == 1
  # Sum(q[j] * coefs[n+k-j]) = -coefs[n+k] for k = 1..m
  arr = []
  for k in range(1, m+1):
    # k == 1 corresponds to x^{n+1}
    row = []
    for j in range(1, m+1):
      row.append(coefs[n+k-j])
    row.append(-coefs[n+k])
    arr.append(row)

  xs = gaussian(arr)  # Solve for q1..qm
  # extract q coefficients
  q = [0] * (m+1)
  q[0] = 1
  for j in range(1, m+1):
    q[j] = xs[j-1]

  # Compute p coefficients (p0 to pn) using the q's
  # p[k] = Sum(q[j] * coefs[k-j])
  p = [0] * (n+1)
  for k in range(n+1):
    tot = 0
    for j in range(min(k, m) + 1):
      tot += q[j] * coefs[k-j]
    p[k] = tot
  return p, q
```

20.11.3 Using Padé. padeTests.py presents a list of functions to the user, and the one selected is given a Padé approximation based on a predefined (n, m) order. The approximation is evaluated against multiple values, and its results are compared with those produced by the actual function. The approximation is also plotted against the function.

Let's compare the `padeApprox()` e^{-x} with the version calculated by hand earlier. The call to `padeTests.py` is:

```
> python padeTests.py
Functions: ['sin(x)', 'exp(x)', 'exp(-x)',
            '1-0.5*ln(1+x^2)', 'tan(x)']
Enter index of function name: 2
Approximating exp(-x) for order (3, 2)
```

Fig. 20.24 shows the plot of the approximation versus the actual function.

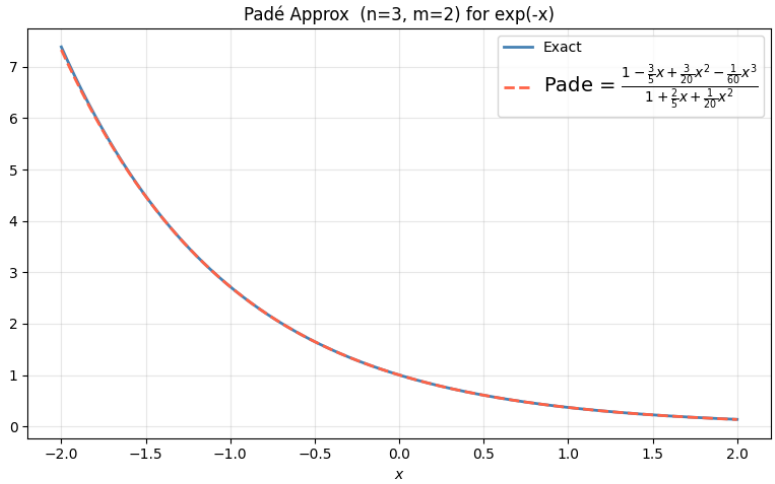


Figure 20.24. A Padé Approximation for e^{-x}

Note that the equation for the Padé approximation printed in the plot's legend matches the one we calculated above. The program's table of comparisons:

```
exp(-x) Comparisons:
      x         Actual          Pade          Abs Err       Rel Err
------------------------------------------------------------------------
  -2.000      7.38905610     7.33333333     0.05572277    0.00754126
  -1.000      2.71828183     2.71794872     0.00033311    0.00012254
  -0.500      1.64872127     1.64871795     0.00000332    0.00000201
   0.000      1.00000000     1.00000000     0.00000000    0.00000000
   0.500      0.60653066     0.60652921     0.00000145    0.00000239
   1.000      0.36787944     0.36781609     0.00006335    0.00017220
   2.000      0.13533528     0.13333333     0.00200195    0.01479252
Average Relative Error = 3.23327537e-03
```

The approximation is excellent around the Taylor series' expansion point ($x_0 = 0$), but becomes less accurate further away. One way to improve matters is to increase the number of n and m coefficients, but the better approach, as typified by Allison's code at the start of this chapter, is to scale and translate the x input down to a narrower range.

One use of a Padé approximation is to determine the location of function singularities when only the first few terms in its expansion are known. Let's demonstrate this in a simple way by looking at $\tan(x)$ which is known to have singularities at $x = \pm\pi/2, \pm 3\pi/2$, etc.

A Padé approximation for $\tan(x)$ generated by `padeTests.py`:

$$r(x) = \frac{x - \frac{2}{21}x^3}{1 - \frac{3}{7}x^2 + \frac{1}{105}x^4}$$

and its plot between -2 and 2 is shown in Fig. 20.25.

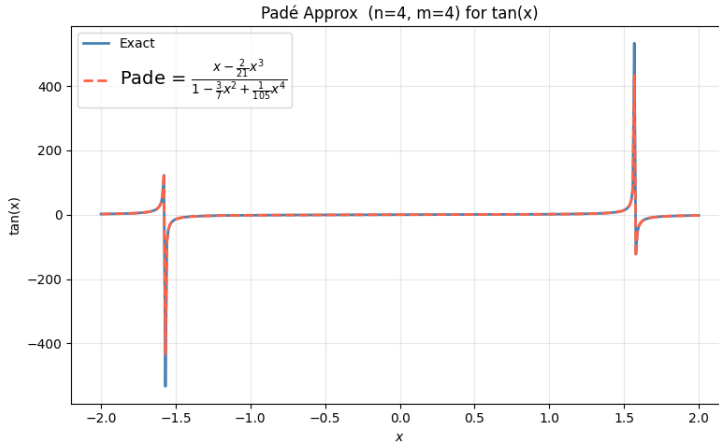


Figure 20.25. A Padé Approximation for $\tan(x)$

It's remarkable that the curve is able to deal fairly well with the singularities at $\pm\pi/2$.

Aside from an examination of the plot, the curve's singularities can also be identified by setting the rational's denominator to 0 and solving it:

$$1 - \frac{3}{7}x^2 + \frac{1}{105}x^4 = 0$$

Let $y = x^2$ so that $y^2 - 45y + 105 = 0$, and then use the quadratic formula:

$$y = \frac{45 \pm \sqrt{45^2 - 4(105)}}{2}$$

to get

$$y_1 \approx 42.531 \quad \text{and} \quad y_2 \approx 2.469.$$

Taking square roots:

$$x \approx \pm 1.571 \quad \text{and} \quad x \approx \pm 6.522$$

and note that $\frac{\pi}{2} \approx 1.570796$.

The $r(x)$ rational for $\tan(x)$ appears to be very different from its standard Maclaurin series expansion:

$$\tan(x) = x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315} + \frac{62x^9}{2835} + \frac{1382x^{11}}{155925} + \cdots$$

This may seem strange since the Padé approximation is derived from the Maclaurin. But if the $r(x)$ numerator is algebraically divided by its denominator, the Maclaurin can be recovered (subject to truncation errors caused by the choice of (n, m)). This can be confirmed by using Sympy:

```
>>> import sympy as sp
>>> x = sp.Symbol('x')
>>> numer = x - (2/21)*x**3
>>> denom = 1 - (3/7)*x**2  + (1/105)*x**4
>>> expr = numer/denom
>>> series = expr.series(x, 0, 8)
>>> seriesFrac = sp.nsimplify(series)
>>> seriesFrac
x + x**3/3 + 2*x**5/15 + 17*x**7/315 + O(x**8)
```

20.11.4 A Return to 'Quick and Dirty'. Most of Allison's functions from section 20.1 use rational approximations obtained from Hart's *Computer Approximations* [**Har68**]. Hart utilized Remez rationals, but it's interesting to see how well Padé can perform under similar constraints.

dpade.py is similar to `padeTests.py`, but utilizes Taylor expansions around points other than $x_0 = 0$. It presents a list of functions to the user, and the selected one is given a Padé approximation based on an (n, m) order. The rational is evaluated against a series of values, and the results are compared to those produced by the actual function.

For example, $\arctan(x)$ uses a Taylor series expanded around $x_0 = 0.5$, and an order $(2, 3)$ which is the same as `atanD(x)` in `dirty.py` (see section 20.4).

`dpade.py`'s Padé approximation is:

$$\frac{\frac{4126}{8899} + \frac{8957}{8844}x + \frac{275}{1591}x^2}{1 + \frac{2413}{5258}x + \frac{635}{2342}x^2 - \frac{141}{2387}x^3}$$

or

$$\begin{aligned} p &= [0.463648, 1.012777, 0.172847] \\ q &= [1, 0.458920, 0.271136, -0.059070] \end{aligned}$$

while the coefficients employed in `dirty.py` are:

$$\begin{aligned} p &= [80.78998, 72.58144, 11.11774] \\ q &= [80.78999, 99.51128, 28.13286, 1]. \end{aligned}$$

Even after scaling, the two sets are disappointingly quite different. In addition, while `atanD(x)` boasts 6 or 7 decimal places of accuracy, the `dpade.py` approximation only manages 3-4:

```
atan(x) Comparisons:
      x         Actual          Pade          Abs Err       Rel Err
-------------------------------------------------------------------------
   0.100     0.09966865     0.09980653     0.00013788    0.00138338
   0.500     0.46364761     0.46364761     0.00000000    0.00000000
   0.900     0.73281510     0.73286184     0.00004673    0.00006377
```

20.11.5 Chebyshev Rational Function Approximation. More accurate rational approximations are possible if we use a Chebyshev polynomial as the starting point rather than a Taylor series. The rest of the procedure is pretty much unchanged except that each x^k term in the Padé approximation is replaced by a kth-degree Chebyshev polynomial, $T_k(x)$. $r(x)$ has the form

$$r(x) = \frac{\sum_{k=0}^{n} p_k T_k(x)}{\sum_{k=0}^{m} q_k T_k(x)},$$

and the original $f(x)$ is expressed as a sum of Chebyshev polynomials

$$f(x) = \sum_{k=0}^{\infty} a_k T_k(x).$$

These are made equivalent as before

$$\sum_{k=0}^{\infty} a_k T_k(x) = \frac{\sum_{k=0}^{n} p_k T_k(x)}{\sum_{k=0}^{m} q_k T_k(x)}$$

or

$$(T_0(x) + q_1 T_1(x) + \cdots + q_m T_m(x))(a_0 T_0(x) + a_1 T_1(x) + \dots) = (p_0 T_0(x) + p_1 T_1(x) + \cdots + p_n T_n(x)).$$

Two problems make this Chebyshev approach more difficult to implement. One is that the multiplication of the polynomial $q(x)$ and the series for $f(x)$ involves products of Chebyshev polynomials. This is addressed by making use of the relationship

$$T_i(x)T_j(x) = \frac{1}{2}\left[T_{i+j}(x) + T_{|i-j|}(x)\right].$$

The other issue involves the computation of the Chebyshev series for $f(x)$; usually numerical integration is required.

Exercises

(1) dirty.py contains Python versions of Dennis Allison's five short but efficient subroutines for $\sqrt{x}$, $\log_{10} x$, 10^x, arctan x, and cos x [**All78**]. The functions utilize various range reducing formulae, and most employ polynomial or rational function approximations borrowed from Hart's *Computer Approximations* [**Har68**].

Fig. 20.2 shows that Allison approximates $\log_{10}(x)$ using a rational with coefficients that offer a precision of 8.66 (8 or 9 correct decimal digits). He approximates arctan(x) with a similar level of precision, as shown in Fig. 20.7. However, it would be more useful nowadays to offer 16 correct decimal digits when working with Python floats. This level of precision for $\log_{10}(x)$ and arctan(x) leads us to Hart's 2328 and 5100 tables, shown in Fig. 20.26.

LOG10 2328

P00	(+ 3)	−.42616 43208 95968 90386 37477
P01	(+ 3)	+.91144 54651 77967 80082 9589
P02	(+ 3)	−.64969 14305 03776 14358 6509
P03	(+ 3)	+.17197 78945 75261 36426 622
P04	(+ 2)	−.12409 45842 40688 46300 36
Q00	(+ 3)	−.49063 98062 30494 57788 77669
Q01	(+ 4)	+.12128 86972 67476 95006 65389
Q02	(+ 4)	−.10541 52597 77636 39322 90112
Q03	(+ 3)	+.37689 50738 71445 48066 757
Q04	(+ 2)	−.47842 12345 10828 24153 49
Q05	(+ 1)	+.1

ARCTN 5100

P00	(+ 4)	+.12097 47001 75809 07217 24071 5
P01	(+ 4)	+.30310 74595 61150 83044 21280 7
P02	(+ 4)	+.27617 19824 61388 34959 05378 4
P03	(+ 4)	+.11141 29072 84551 83546 17294 2
P04	(+ 3)	+.19257 92014 48155 96134 74286
P05	(+ 2)	+.11322 15941 16764 65523 6245
P06	(− 1)	+.97627 21591 71763 30369 83
Q00	(+ 4)	+.12097 47001 75809 07287 51419 7
Q01	(+ 4)	+.34343 23596 19753 51716 54706 9
Q02	(+ 4)	+.36645 44956 32837 49893 50479 6
Q03	(+ 4)	+.18216 00339 29184 64941 50922 5
Q04	(+ 3)	+.42307 16464 80904 78045 24206
Q05	(+ 2)	+.39917 88424 86537 98150 1999
Q06	(+ 1)	+.1

Figure 20.26. Hart's Tables for $\log_{10}(x)$ and arctan(x) with 16 d.p. of Accuracy

Modify `log10D(x)` and `atanD(x)` in dirty.py to use the coefficients in Fig. 20.26. Incidentally, there's no need to for you to type in these lengthy values, as they're available in HartTables.txt.

Compare the quality of the results with the old versions of `log10D(x)` and `atanD(x)`.

(2) Investigate the power series methods for computing e^{-x} when x is close to -3. The Maclaurin series is:

$$e^{-x} = \sum_{n=0}^{\infty} \frac{(-1)^n x^n}{n!}$$

and the Taylor series centered at $x = -3$ is

$$e^{-x} = e^3 \sum_{n=0}^{\infty} \frac{(-(x+3))^n}{n!}$$

Note that the Taylor series is equivalent to the Maclaurin with variable renaming. Let $\delta = x + 3$, and then

$$e^{-x} = e^3 e^{-\delta} = e^3 \sum_{n=0}^{\infty} \frac{(-\delta)^n}{n!}.$$

Explain why the second series might be expected to converge more rapidly near $x = -3$, and why one form is more susceptible to round-off errors.

See how the errors vary with polynomial degree and across a range of x values. You can measure the absolute error using $|f(x) - P_N(x)|$, and the relative error with $\frac{|f(x)-P_N(x)|}{|f(x)|}$, where $P_N(x)$ is the power series expanded to N terms.

Several techniques for rewriting a Maclaurin series are listed below. Investigate how they affect the quality of the e^{-x} series.

- **Reciprocals**. Since $e^{-x} = 1/e^x$, you can compute $e^x = e^{-3}e^{\delta}$ and take the reciprocal of the result.
- **Repeated Halving**. Use

  $$e^{-x} = \left(e^{-x/2^m}\right)^{2^m}$$

  where m is chosen so that $|x|/2^m$ is small. This approach (and the next two) relies on the fact that multiplication by powers of two is both fast and accurate in floating-point arithmetic, and often implemented by bit-shifting.
- **Centering**. Since $e^{-\ln 2} = 1/2$, you can write $x = k \ln 2 + r$, giving $e^{-x} = 2^{-k}e^{-r}$.
- **Rational Approximation**. Define

  $$e^{-\delta} = \frac{e^{-\delta/2}}{e^{\delta/2}},$$

  and approximate its numerator and denominator separately.
- **Hyperbolics**. Utilize the identity $e^{-x} = \cosh(x) - \sinh(x)$, along with the Maclaurin series for those functions.

Another type of series optimization involves the way that calculations are performed. For example:

- **Horner's Method**. Rather than evaluating

$$1 - \delta + \frac{\delta^2}{2!} - \frac{\delta^3}{3!} + \cdots,$$

rewrite the expression as

$$1 + \delta\left(-1 + \delta\left(\frac{1}{2} + \delta\left(-\frac{1}{6} + \cdots\right)\right)\right).$$

- **Summing Terms Recursively**. Instead of computing $\frac{\delta^n}{n!}$ from scratch each time (e.g. by calling `math.factorial()`), use

$$t_0 = 1, \qquad t_{n+1} = t_n \frac{-\delta}{n+1},$$

- **Sum Small Terms First**. Sum the series terms in reverse order to reduce the chance of rounding errors.

3. Table 20.3 lists the population of the United States from 1950 to 2020 along with estimations for 2021-2024. Use Lagrange interpolation to approximate the population in the years 1940, 2000, 2024, and 2050. The population in 1940 was approximately 132,165,000. How accurate do you think your 2050 figure is?

Year	Population
1950	151,325,798
1960	179,323,175
1970	203,211,926
1980	226,545,805
1990	248,709,873
2000	281,421,906
2010	308,745,538
2020	331,449,281
2021	332,031,554 (est)
2022	333,287,557 (est)
2023	336,806,231 (est)
2024	340,110,988 (est)

Table 20.3. Population of the United States, 1950–2024.

There's no need to type in this population data, as it's available in `usPop.txt`.

4. Using a Maclaurin polynomial for xe^x, obtain a small degree Chebyshev approximation while keeping the error less than 0.01 across the $[-1, 1]$ interval.

5. Using a Maclaurin polynomial for $\sin(x)$, obtain a small degree Chebyshev approximation while keeping the error less than 0.01 across the $[-1, 1]$ interval.

6. Show that for any positive integers i and j with $i > j$, we have

$$T_i(x)T_j(x) = \frac{1}{2}[T_{i+j}(x) + T_{i-j}(x)].$$

7. Show that the Chebyshev polynomial $T_n(x)$ has n distinct zeros in $[-1, 1]$.

8. The table below lists results for the Padé approximation of $f(x) = e^{-x}$ using degree 5 with $n = 3$ and $m = 2$, the sixth Maclaurin polynomial $(n + m + 1)$, and function values when $x = 0.2, 0.4, 0.6, 0.8$, and 1. The Padé approximation is clearly better than the Maclaurin.

x	Actual	Pade	Mac	Pade Err	Mac Err
0.200	0.818731	0.818731	0.818733	0.000000	0.000003
0.400	0.670320	0.670320	0.670400	0.000000	0.000080
0.600	0.548812	0.548808	0.549400	0.000004	0.000588
0.800	0.449329	0.449310	0.451733	0.000019	0.002404
1.000	0.367879	0.367816	0.375000	0.000063	0.007121

But does the Padé approximation still win when it utilizes different orders of degree 5:

a. $n = 0, m = 5$

b. $n = 1, m = 4$

c. $n = 2, m = 3$

d. $n = 4, m = 1$

e. $n = 5, m = 0$

Answers

3. See usPop.py.

4. See chebExp.py.

6. If $i > j$, then

$$\begin{aligned} \tfrac{1}{2}(T_{i+j}(x) + T_{i-j}(x)) &= \tfrac{1}{2}(\cos(i+j)\theta + \cos(i-j)\theta) \\ &= \cos(i\theta)\,\cos(j\theta) \\ &= T_i(x)\,T_j(x). \end{aligned}$$

(n,m)	Error $\times 10^{-6}$
(0,5)	133
(1,4)	80
(2,3)	13
(3,2)	17
(4,1)	47
(5,0)	321

Table 20.4. Padé Errors for Different (n, m).

7. The zeros of $T_n(x)$ are $x_k = \cos\left(\frac{2k-1}{2n}\pi\right)$ for $k = 1, 2, \ldots, n$.

The cosine function is strictly decreasing from 0 to π, with values decreasing from $\cos(0) = 1$ to $\cos(\pi) = -1$. So, in reverse order:

$$\begin{aligned}(-1 = \cos(\pi)) &< (\cos\left(\tfrac{2n-1}{2n}\pi\right) = x_n) \\ &< x_{n-1} < \cdots < (x_1 = \cos\left(\tfrac{\pi}{2n}\right)) \\ &< (\cos(0) = 1)\end{aligned}$$

Hence the zeros are distinct and lie in $[-1, 1]$.

8. See padeExp.py.

The average error for different versions of (n, m) are given in Table 20.4

Appendix A

A Crash Course in Probability

Fig. A.1 shows a spinner with a perimeter length of 1:

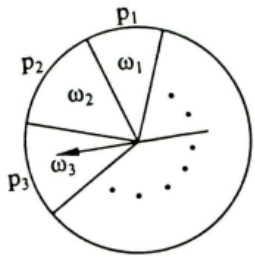


Figure A.1. Selecting a probability

Each segment ω_i has an arc length of p_i, such that

$$p_i \geq 0, \sum_{i \geq 1} p_i = 1. \tag{A.1}$$

Spinning the spinner once is a *one stage random experiment*, where ω_i denotes the *outcome* that the spinner comes to rest in sector i. The set

$$\Omega = \{\omega_1, \omega_2, \omega_3, ...\}$$

of all the possible outcomes is the *sample space*.

Spinning the spinner n times is an *n-stage experiment*, the outcomes being n-words from the alphabet Ω. Suppose that in n spins, ω_i is obtained F_i times. Then F_i is the *absolute frequency* and $f_i = F_i/n$ is the *relative frequency* of ω_i.

The symmetry of the circle implies that if the relative frequency of ω_i always approaches the same value, that value has to be the proportion of the perimeter assigned to ω_i, i.e. p_i. Indeed, Las Vegas records of various spinning devices confirm that for large n, $f_i \approx p_i$, although, as n increases, f_i's approach towards p_i is both slow and erratic. We call p_i the *probability* of the outcome ω_i

We should mention that it is a profound problem to come up with a logical argument that explains why an average over many repetitions of a random experiment should approach a limit; it could conceivably fluctuate aimlessly forever.

We call every subset A of Ω an *event*. Suppose a spin results in $\omega \in A$ then we say that *the event A has occurred*. For every event the frequency of occurrences of A is the sum of the frequencies of the outcomes in A. Hence the probability of the event A is

$$P(A) = \text{ sum of the probabilities of all outcomes in A.} \tag{A.2}$$

If $|\Omega| = n$ and $p_i = 1/n$ for all i (i.e. the outcomes are equally likely) then we have

$$P(A) = \frac{|A|}{|\Omega|} = \frac{\#\text{ of elements in A}}{\#\text{ of elements in }\Omega} = \frac{\#\text{ of favorable cases}}{\#\text{ of possible cases}}. \tag{A.3}$$

You might think that a more natural description of the spinner would be to use the angles as the set of outcomes. However, that model actually requires a more complicated theory.

We want to study *discrete probability*, which is the theory of random experiments in which every possible outcome has a positive probability.

Usually we are given a random process in which we perform a series of random experiments. This process can be pictured as a *directed graph* (see Fig. A.2), with vertices called *states*. Conceptually, there is a spinner located inside each state which determines the *transition probabilities* for moving to the neighboring states.

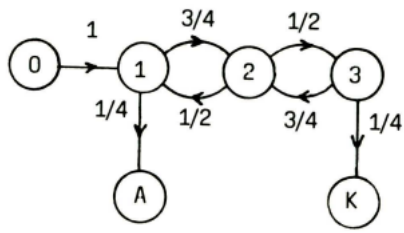


Figure A.2. A graph of transition probabilities

A "particle" starts in some state, say state 0, and moves from state to state, according to the transition probabilities. For example, the spinner in state 3 (see Fig. A.3), decides whether to move back to state 2 or to state K. The series of

state transitions is called a *random walk*. For instance, one of the infinitely many possible paths is 0121232123K.

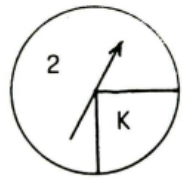


Figure A.3. Deciding between '2' and 'K'

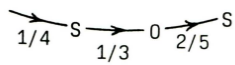
With this terminology we can say:

Discrete probability studies random walks on graphs.

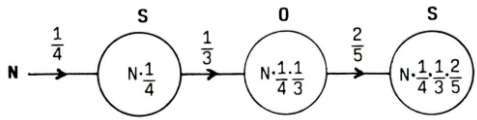
To see how path probabilities are obtained from transition probabilities, consider the following example. From each box in Fig. A.4a we select a letter at random from left to right, and write down the letters in the order they were drawn. The probability of the outcome being "SOS" in depicted in Fig. A.4b.



(a) Selecting SOS



(b) Probabilities for selecting SOS



(c) Transition probabilities for SOS

Figure A.4. SOS probabilities

Suppose we repeat the experiment a large number, N, times. Fig. A.4c shows the computation of the approximate numbers of paths through the two intermediate states, to the final state. It tells us that the probability of "SOS" is

$$P(SOS) = \frac{1}{4}\frac{1}{3}\frac{2}{5} = \frac{1}{30}.$$

A general formulation of what we have just observed is:

P1. The probability of a path is equal to the product of the probabilities along the path.

This is the *first path rule*.

A second example illustrates another rule.

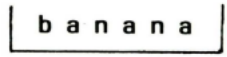


Figure A.5. A banana box

From the box in Fig. A.5 we draw two letters at random without replacement. What is the probability that the second letter is 'a'?

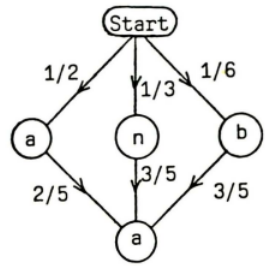


Figure A.6. Probability that the second letter is 'a'

Tracing the three paths in Fig. A.6 which produce 'a' in the second draw results in the equation

$$P(a \text{ at second draw}) = \frac{1}{2}\frac{2}{5} + \frac{1}{3}\frac{3}{5} + \frac{1}{6}\frac{3}{5} = \frac{1}{2}.$$

We can now formulate a *second path rule*:

P2. The probability of getting from S to F is the sum of the probabilities of all the paths leading from S to F.

This is a reformulation of Equ. (A.2) into the language of paths. We are thinking of all the possible outcomes as all the ordered pairs of letters that can be drawn, and A = {aa, ba, na}.

One objective of the theory of probability is to compute the probabilities of complicated events associated with random experiments from the known probabilities of the outcomes of a single experiment. For this purpose a simplified concept of probability suffices. The only thing we need to add is that the probabilities

p_i of the outcomes ω_i of a single experiment are non-negative real numbers satisfying Equ. (A.1), and that the probabilities of compound events are given by the boxed rules P1 and P2.

The fact that individual outcomes are unpredictable, while long-term averages approach limits, is a subtle and mysterious fact. Fortunately, in order to compute probabilities we only need the simple consequences which we stated above! Mathematicians were very pleased when it became clear that the theory could be derived from axioms as simple as these. However, we ought to add that this relative-frequency interpretation of probability lets us easily grasp many aspects of probability that are a lot less easy to formulate or prove axiomatically.

Next we look at a few examples that bring up additional concepts.

A.1 A Bold Gamble

You have 1 dollar but you desperately need 5 dollars. So you go into a casino where you can win as much as you bet with probability p and lose with probability $q = 1 - p$. You decide to use the *bold strategy*: at each stage you stake as much money as possible in order to minimize the distance to your goal. In this scenario, what is your probability of going broke?

Fig. A.7 shows the graph of the game. As a convenience, the state labels correspond to the amount of money you currently possess. Therefore, you start at state 1 and stake all your money. With probability p you move to state 2, and with probability q you are ruined. In state 2 you stake all your money again, but in state 4 you stake just one dollar (since you're only one dollar from your goal), and in state 3 two dollars.

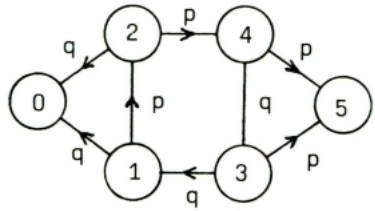


Figure A.7. Probability graph for the *bold gamble*

The probability B of going broke is the sum of the probabilities along all the paths leading to state 0 (i.e. the 'no money' state). You can lose directly, or after one, two, ... cycles around the graph. Thus $B = q + pq + p^2q^2(q + pq) + p^4q^4(q + pq) + \ldots$. This is a geometric series which sums to

$$B = \frac{q + pq}{1 - p^2q^2}.$$

A.2 Random Variables and Expectation

Often a random experiment produces a *random variable* X. This means that the possible outcome can come from a range of real numbers $x_1, x_2, ...$ whose probabilities of being chosen are $p_1, p_2, ...$ You might like to interpret the x_i as gains in a roulette game (Fig. A.8).

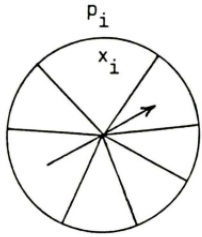


Figure A.8. Probability as roulette

Suppose you spin the spinner in Fig. A.8 N times, where N is large. We can compute the average gain as follows. We will win x_i approximately Np_i times, so the total gain will be approximately

$$Np_1x_1 + Np_2x_2 + Np_3x_3 + ...$$

If we play long enough, the average gain per game is going to be close to $1/N$-th of this:

$$E(X) = x_1p_1 + x_2p_2 + x_3p_3 + ... = \sum_{i\geq 1} x_ip_i. \tag{A.4}$$

The quantity E(X) defined by Equ. (A.4) is called the *expectation* or *expected value* of X. Note that long term averages, and the complicated concept of playing "long enough", only appear when we have to explain the meaning of $E(X)$.

Suppose that besides X, another random variable X' with values $x'_1, x'_2, ...$ and corresponding probabilities $p'_1, p'_2, ...$ is defined. If a is any numerical constant then the definition of expectation as a long-term average tells us that the expectations of the random variables aX and $X + X'$ are:

$$E(aX) = aE(X) \tag{A.5}$$

and

$$E(X + X') = E(X) + E(X'). \tag{A.6}$$

Equ. (A.5) is an immediate consequence of Equ. (A.4). To derive Equ. (A.6) we need to set up a notation for an experiment in which one value is obtained for each of the two random variables X, X'. (The same approach still applies if the experiment involves two values from the same random variable.)

An outcome of the two-variable experiment is that we get a value x_i for X and and a value x'_j for X'. Let p_{ij} denote the probability of this outcome. In general these quantities can not be computed from $p_1, p_2, ...$ and $p'_1, p'_2, ...$

For example, suppose we have an box with 4 balls, two with '0' written on them and two with '1', and X and X' are obtained by drawing 2 balls without replacement. If we assume that each ball is equally likely to be the first and each of the remaining ones is equally likely to be the second, then we get

$$p_1 = p_2 = p'_1 = p'_2 = \frac{1}{2} \quad \text{and} \quad p_{11} = p_{22} = \frac{1}{6}, \quad p_{12} = p_{21} = \frac{1}{3}.$$

The relationship between the individual probabilities and the joint probabilities is the following. The event $X = x_i$ occurs if we have any one of the outcomes $X = x_i$ and $X' = x'_1$ or x'_2 or x'_3, ... Hence by Equ. (A.2):

$$p_{i1} + p_{i2} + ... = p_i \text{ and similarly} \tag{A.7}$$

$$p'_{1j} + p'_{2j} + ... = p'_j. \tag{A.8}$$

Applying Equ. (A.4) to $X + X'$ we get

$$E(X + X') = \sum_{i,j} (x_i + x'_j) p_{ij}.$$

If we separate out the terms involving the x_i and apply Equ. (A.7) we get $E(X)$, and the remaining terms combine to give $E(X')$.

A.3 More on Random Walks

Usually a graph includes a set A of *absorbing states* – states that once you enter, you cannot leave. The other states are in the set I of *interior* states. We are interested in two problems.

(1) If we start in some state i, what is the probability p_i of reaching some subset W of A?

(2) Starting in a state i, what is the expected time (# of transitions), E_i, to reach A?

For $i \in W$ we have $p_i = 1$, and for $i \in L = A \backslash W$ (the complement of W in A) we have $p_i = 0$. If we are in state $i \in I$, let p_{ik} denote the probability that the next move will be to state k. so, we have

$$p_i = \sum p_{ik} p_k, \tag{A.9}$$

where the sum is over all the neighbors k of i. Indeed, we move from i to k with probability p_{ik}, and from there we reach W with probability p_k. For all $i \in A$ we have $E_i = 0$, and for $i \in I$ we get

$$E_i = 1 + \sum p_{ik} E_k \quad \text{(The sum is over all neighbors k of i.)} \tag{A.10}$$

We have made one step to k with probability p_{ik}, and from there the expected time to absorption is E_k.

Expectations can be infinite. For instance, in a symmetric random walk on a line, starting at O, the expected number of steps to return to O is infinite. The system of equations $E_i = 1 + \frac{1}{2}E_{i-1} + \frac{1}{2}E_{i+1}$ has no solution, unless we consider $E_i = +\infty$ a solution. This result models how, as time goes on, the particle wanders over ever longer stretches of the line. We can show that it's almost certain that the particle will eventually return to the origin or, equivalently, to any vertex it has previously visited. However, the average time it takes to return to a particular place will increase without limit as more and more new vertices are visited.

A.4 Five Practice Problems

In this section, we gain some practice in representing a problem using a graph and in applying Equs. (A.9) and (A.10).

A.4.1 Problem 1. What is the expected time m to get from state 0 to 1 in Fig. A.9?

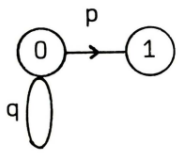


Figure A.9. Getting from '0' to '1'

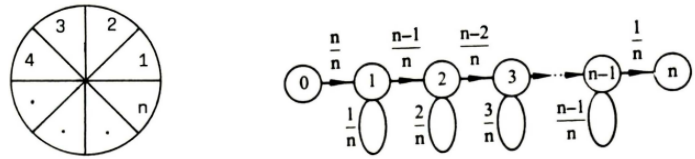
The expected time to get from state 1 to 1 is 0. So we get

$$m = 1 + qm, \quad q = 1 - p, \text{ and } m = \frac{1}{p}. \tag{A.11}$$

A.4.2 Problem 2. Spin the spinner in Fig. A.10a until all n outcomes have occurred. What is the expected number of spins?

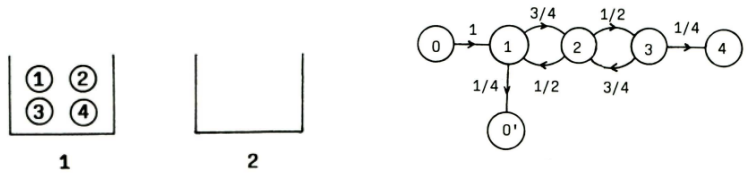
Consider Fig. A.10b: we are in state i if we have collected i outcomes. By Equ. (A.11) the expected time to get from i to $i+1$ is $n/(n-i)$. Due to Equs. (A.5) and (A.6), the expected time to get from state 0 to n is

$$E(n) = \sum_{i=1}^{n} \frac{n}{(n-i)} = n(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}).$$



(a) n outcomes (b) Transition probabilities

Figure A.10. Getting all n outcomes



(a) Two boxes (b) Transition probabilities for moving a ball

Figure A.11. Moving balls between boxes

A.4.3 Problem 3. Fig. A.11a shows two boxes labeled 1 and 2, with box 1 initially containing four balls. We repeatedly randomly select a ball from any box and move it to the other box.

We'll say that the system is in state i if the second box contains i balls, and will stop if either we reenter state 0 or state 4 is reached.

The random process can be represented by Fig. A.11b. Let p_0 be the probability of being absorbed in state 4 if we start in state 0. The instantiated versions of Equ. (A.9) are

$$p_0 = p_1, \quad p_1 = \frac{3}{4}p_2, \quad p_2 = \frac{1}{2}p_1 + \frac{1}{2}p_3, \quad p_3 = \frac{3}{4}p_2 + \frac{1}{4}$$

Solving these we get $p_0 = \frac{3}{8}$.

Let E_i be the expected time until we stop after starting in state i. Equ. (A.10) is instantiated as

$$E_0 = 1 + E_1; \quad E_1 = 1 + \frac{3}{4}E_2; \quad E_2 = 1 + \frac{1}{2}E_1 + \frac{1}{2}E_3; \quad E_3 = 1 + \frac{3}{4}E_2.$$

This system of equations sets $E_0 = 8$.

A.4.4 Problem 4. Start with $x = 3$ coins. While $x > 0$, toss the coins and eliminate those that come up heads. What is the expected number of tosses until $x = 0$?

Let a, b, c be the expected number of tosses for $x = 3, 2, 1$. From Fig. A.12, we get $a = 1 + a/8 + 3b/8 + 3c/8$, $b = 1 + b/4 + c/2$, $c = 1 + c/2$, with the results $c = 2, b = 8/3, a = 22/7$.

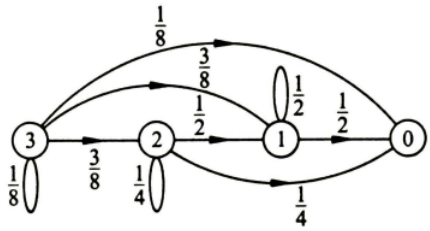


Figure A.12. Eliminating three coins

A.4.5 Problem 5. Let's reconsider the Three Pile Game from Ex. 2 of "Additional Exercises for Chapters 1-4". We have three piles of a, b, and c chips. A clock is ticking, and each second two different piles X, Y are chosen at random and a chip moved from X to Y. Let T be the time until one pile is empty. Devise an equation for the expected time $E(T) = f(a, b, c)$.

By simulation we can arrive at a guess

$$f(a, b, c) = \frac{3abc}{a + b + c}.$$

Theory lets us verify this equation. The six neighbors of the state (a, b, c) are $(a, b + 1, c - 1)$, $(a, b - 1, c + 1)$, $(a + 1, b, c - 1)$, $(a - 1, b, c + 1)$, $(a + 1, b - 1, c)$, and $(a - 1, b + 1, c)$. Equ. (A.10) becomes

$$f(a, b, c) = 1 + \frac{1}{6}\sum f(x, y, z) \text{ (sum over all neighbors of } (a, b, c)) \qquad \text{(A.12)}$$

$$f(a, b, 0) = f(a, 0, c) = f(0, b, c) = 0 \quad \text{(boundary conditions).} \qquad \text{(A.13)}$$

We can verify that the f function is a solution for these equations by substitution, but is it the only solution? Suppose that there's another solution $g(a, b, c)$. We consider $h(a, b, c) = f(a, b, c) - g(a, b, c)$ so that at an interior point (i.e. a point where a, b, c are all positive) this difference satisfies

$$h(a, b, c) = \frac{1}{6}\sum h(x, y, z).$$

In words: the value of h at an interior point is the average of the values at the six neighboring points.

Let $h_{\max}$ be the maximum of h. If $h = h_{\max}$ at an interior point then we must also have $h = h_{\max}$ at all the neighbors of that point. This means that we can move from point to point obtaining $h = h_{\max}$ at each one. However, since $h = 0$ on the boundary, then h must $= 0$ everywhere.

Alternatively, if $h = h_{\max}$ only occurs at the boundary then we still have $h_{\max} = 0$ but now we can only say that $h \leq 0$ everywhere else. However, we can apply the same argument to the minimum of h and get $h \geq 0$ everywhere, which implies that $h = 0$ everywhere. Hence, every solution g is equal to f.

A.5 More Information

An excellent modern introduction to probability and statistics is the text by Joseph K. Blitzstein and Jessica Hwang [**BH19**], developed from the Harvard course, 'Statistics 110' (`https://projects.iq.harvard.edu/stat110/`). Applications areas include genetics, medicine, computer science, and information theory. There's a free online (but not printable) version of the second edition at `http://probabilitybook.net`, and a series of 35 video lectures on Youtube at `https://www.youtube.com/watch?v=KbB0FjPg0mw&list=PL2SOU6wwxB0uwwH80KTQ6ht66KWxbzTIo`.

A superb classic presentation of discrete probability theory, for those who wish to immerse themselves in the subject, is William Feller's *Introduction to the Theory of Probability*, Vol. 1 [**Fel68**].

Appendix B

Solutions to Some Exercises

B.1 Sections 1.1 to 1.7

1. a) $f(20)/f(19) = 4.7326261527$ b) Same answer as in a). Use Listing 1.25 after changing 1E-09 into 1E-08.

2. bisiter-q2.py. This iterative program finds a root of the continuous function f in the interval $[a, b]$, if $f(a) * f(b) < 0$. The boolean variable `faneg` is true if $f(a)$ is negative. The name should remind you of this.

3. a) 0.73908513226 b) 1.7632228322 c) 0.56714328937 d) 4.493409458

4. x_1 = -1.8793852418, $x_2 = 0.34729635529$, $x_3 = 1.5320888860$.

5. b) We conjecture $g^2(n) - 5 * f^2(n) = 4 * (-1)^n$. Proof by induction.

6. a) True. b) True. Proof by induction.

7. a) $f(1/3) = p \cdot f(2/3) = p(p + q \cdot f(1/3))$, $f(1/3) = p^2/(1 - pq)$.

b) $f(2/5) = p \cdot f(4/5) = p^2 + pq$?f(3/5) $= p^2 + p^2q + pq^2$f(1/5) $= p^2 + p^2q + p^2q^2/(2/5)$, f(2/5) $= p^2(1 + q)/(1 - p^2q^2)$.

c) f(1/1984) $= p^{11}/(1 - p^4q)$. For $p = q = 1/2$ we get $f(x) = x$.

d) The longer the period of the binary expansion of x the harder it is to find $f(x)$.

8. joseph2-q8.py

11. We get an unbranching recursion if we compute u values in blocks of two. This reduces the number of operations to $O(\log n)$. (The time complexity is greater because just the number of digits in $u(n)$ is $O(n)$.)

12. By a slight modification of joseph1.py we get Listing 5.25.

13. a)

```
s = 0; n = 0
while n < 10000:
  n += 1
  s += 1/n
```

b)

```
s = 0; n = 10000
while n < 0:
  s += 1/n
  n -= 1
```

c) When two positive floating-point numbers with different exponents are added, the digits of the smaller number which are in places below the least significant digit of the larger number are mostly neglected. At best, they are taken into account in the rounding of the result. Thus, if we form the sum starting with the smallest terms, the error will be smaller than if we start with the largest terms. It would be even better to add groups of small terms and then add the resulting totals instead of adding the terms individually to the total.

14. The computations suggest

$$\text{round}(\frac{n}{2}) + \text{round}(\frac{n}{4}) + \text{round}(\frac{n}{8}) + ... = n. \tag{B.1}$$

This can be proved as follows. The formula is true for $n = 0$. If we increase n by 1, the term $\text{round}(n/2^k)$ in the sum (B.1) is unchanged except when n reaches a value which is an odd multiple of 2^{k-1}, and then it increases by 1. For each value of n there is one such k and therefore the sum of the series is n.

15. a)

```
for x in range(100):
  for y in range(100):
    if abs(x*x - x*y - y*y) == 1:
      print(x, y);
```

We get pairs of successive Fibonacci numbers.

b) We observe that the solutions are

$$(\text{fib}(n+1), \text{fib}(n)) \quad n = 1, 2, ... \tag{B.2}$$

It is easy to verify that

$$\text{if } w = u + v, \text{ then } w^2 - wv - v^2 = -(v^2 - vu - u^2). \tag{B.3}$$

This equation enables us to start from the solution $(1, 1) = (\text{fib}(2), \text{fib}(1))$ and obtain the family of solutions (B.2).

Next we show that if x and y are > 0 and

$$|x^2 - xy - y^2| = 1 \tag{B.4}$$

then (x, y) is in the family (B.2).

If $x = 1$ then Equ. (B.4) implies $y = 1$ so we have the first member of the family (B.2). If $x > 1$, (B.4) implies $y < x$. By (B.3), $(y, x - y)$ is also a solution with positive components. We can continue to descend as long as the x value is > 1, i.e. until we reach $(1, 1)$. We can now reverse the process and get back to the (x, y) we started with, and this amounts to forming the Fibonacci sequence.

16. We give five different solutions, which are successively more sophisticated and efficient. A sixth program will be given later.

equisum1-q16.py
equisum2-q16.py
equisum3-q16.py
equisum4-q16.py
equisum5-q16.py

Listings `equisum1-q16.py` to `equisum4-q16.py` use 1000000, 28000, 2800, 1400 comparisons, respectively. In Listing `equisum2-q16.py` we denote by $p(s)$ the number of triples (a, b, c) with fixed sum s. For each such sum $s = a + b + c$ there are $p(s)$ triples (d, e, f) with the same sum $s = d + e + f$. So $p(s) * p(s)$ is the total number of cases with sums. This is added up for $s = 0$ to 27. In Listing `equisum3-q16.py` we save the inner c-loop by means of the test if $(a + b \le s)$ and $(a + b \ge s - 9)$, which is satisfied for $0 \le c \le 9$. For given s, a, b the variable c is also given by $c = s - a - b$.

In Listing `equisum4-q16.py` we use the bijection $abcdef \leftrightarrow a'b'c'd'e'f'$ between blocks with equisum property, where $a' = 9 - a$, etc. If $s = a + b + c$ then $s' = a' + b' + c' = 27 - s$. So triples (a, b, c) with sums and $27 - s$ are equi-numerous. Here s runs from 0 to 13 and the total count is doubled at the end.

The recursive program Listing `equisum5-q16.py` is based on the recurrences

$$p(s) = p(s-1) + s + 1 \quad \text{for } s \le 9,$$
$$p(s) = 100 - p(s-10) - p(17-s) \quad \text{for } s > 9.$$

A triple (a, b, c) with sum s is determined by its two digits a, b. Also, any pair of digits a, b with

$$s - 9 \le a + b \le s \tag{B.5}$$

will form such a triple with $c = 9 - a - b$. Thus $p(s)$ is the number of solutions of (B.5). To count these, consider the 100 lattice points (a, b) in or on the boundary of the square $\mathrm{S} : 0 \le a, b \le 9$. For $s \le 9$, $p(s)$ is the number of lattice points in or on the boundary of the triangle cut off from S by the line $a + b = s$. For $s > 9$ we can think of the lattice points in the region (B.5) as the lattice points in S from which we remove the lattice point in or on the boundary of the triangle $a + b \le s - 10$ and also the lattice points in or on the boundary of the triangle $a + b \ge s + 1$. The last triangle is congruent to the triangle $a + b \le 17 - s$.

B.2 Section 2.1

4. a) We want to find $x = 7^{9999} \bmod 1000$. We first find the smallest power of 7 which is 1 (mod 1000).

```
p = 1; i = 0
while True:
  i += 1;
  p = (7*p) % 1000
  if p == 1:
    break
```

We get $i = 400$. Since $9999 = 400 * 24 + 399$ we have $7^{9999} \equiv 7^{399} \pmod{1000}$.

```
p = 1
for i in range(399):
  p = (7*p) % 1000;
```

Now $p = 143$, which is the solution x.

A completely different solution is based on Listing 2.4. We want to find $x = 7^{9999} \bmod 1000$, or $7x = 7^{10000} \bmod 1000$. Now $7^{10000} = 7^{400*25} = 1 \pmod{1000}$. Thus $7x \equiv 1 \pmod{1000}$ or $7x + 1000y = 1$. Listing 2.4 gives $x = 143$, $y = -1$, and so $x = 143$. b) and c) may require Listing 2.5.

6. a)

```
sum = 0
for i in range(11):
  sum += i
```

b)

```
def sum(n):
  if n == 1:
    return 1
  else:
    return sum(n-1)+n
```

7. a)

```
def bin(n):
  if n > 1:
    bin(n // 2)
  print(n % 2, end='')
```

b)

```
while n > 0:
  print(n % 2, end='')
  n = n // 2
```

In b) the digits are printed in reverse order. In a) the digits are printed in correct order. But if we switch the bin() and print() procedures we get the bits in reverse order.

binary-q8.py

8.

```
d = 0
while n > 0:
  d += n % 10
  n = n // 10
```

9. digdig-q10.py

$f(n)$ is 9 if n is a multiple of 9; otherwise it is n mod 9.

10. rev6-q11.py

11. subs6-q12.py

12. subs6-q13.py

13. q(n) = fib(n + 1)/fib(n) approaches $q = 1.6180339887... = (\sqrt{5}+1)/2$. Show by induction that $q(n) - q = (-1)^n/\text{fib}(n)/q^n$.

fibRatio-q15.py

14. divBy-q16.py

15. a) The sequence completes a cycle if two successive terms repeat. This must happen because of the pigeonhole principle.

b) Addition mod m is invertible. Hence the sequence can be extended uniquely into the past. But in a loop with a tail the end of the tail has two predecessors.

c) FibPeriodMODm-q17.py

16. a) $L(p^n) = L(p) * p^{n-1}$ for every prime p.

b) $L(p_1^{a_1} p_2^{a_2} \cdots p_r^{a_r}) = \text{lcm}(L(p_1^{a_1}), L(p_2^{a_2}), \cdots, L(p_r^{a_r}))$. In particular,

$$L(10^n) = \text{lcm}(3 * 2^{n-1}, 2^2 * 5^n) = \begin{cases} 12 * 5^n & \text{for } n \le 3 \\ 15 * 10^{n-1} & \text{for } n > 3 \end{cases}$$

17. a) $L|p-1$. b) $L|2(p+1)$. c) $L = 20$.

19. Zeta(3) $= 1 + 1/2^3 + \cdots + 1/n^3 + 1/2(n+1/2)^2 = 1.2020569032...$ from $n = 200$ on.

20. Let A_d be the event gcd(x, y, z) = d. A_d is equivalent to the occurrence of four independent events

$$d|x, \quad d|y, \quad d|z, \quad \gcd(x/d, y/d, z/d) = 1.$$

Thus

$$P(A_e) = \frac{1}{d} * \frac{1}{d} * \frac{1}{d} * p_3 = p_3/d^3, \text{ where } p_3 = P(\gcd(x, y, z) = 1).$$

Since $A_1 \cup A_2 \cup A_3 \cup ...$ will occur with probability 1, we have

$$p_3 \sum_{d \ge 1} 1/d^k = 1, \text{ or } p_3 = 1/\sum_{d \ge 1} 1/d^3.$$

21. Let A_d be the event $\gcd(x_1, \cdots, x_k) = d$. That is

$$d|x_1, \cdots, d|x_n, \quad \gcd(x_1/d, ..., x_n/d) = 1.$$

Then $P(A_d) = p_k/d^k$, where $P(A_1) = 1$. Thus

$$p_k \sum_{d\geq 1} 1/d^k = 1 \rightarrow p_k = 1/\sum_{d\geq 1} 1/d^k.$$

22. a) will be proved after c).

First we show that the process we gave for generating the Morse-Thue sequence is equivalent to the following:

Denote by S the operation, applicable to a sequence of 0's and 1's, which replaces each 0 by 01 and each 1 by 10. Observe that 10 is the complement of 01 so that if we apply S to a pair of complementary sequences we get a pair of complementary sequences. We claim the initial segment of 2^n digits of the Morse-Thue sequence is $S^n(0)$. Indeed this is clear for $n = 0$ and $n = 1$. Now $S^{n+1}(0) = S^n(01)$, which is $S^n(0)$ followed by $S^n(1)$. Since the S-operator takes complements into complements, what we have now is the rule of formation of the Morse-Thue sequence originally given.

Observe that the operation S can be applied to the entire infinite Morse-Thue sequence and it leaves the sequence unchanged. Suppose now that $x(n) = 0$. In the Morse-Thue sequence $x(n)$ is preceded by n digits. If we apply S to the sequence we replace $x(n) = 0$ by 01, and then digits preceding it by $2n$ digits. So in this case $x(2n) = x(n)$ and $x(2n + 1) = 1 - x(2n)$. The case $x(n) = 1$ follows similarly. This proves c); the assertion b) is just another way of phrasing the first part of c).

To prove a), we note that the two equations c) together imply that $x(n)$ and $x(2n + 1)$ are always different. If the sequence were ultimately periodic and $x(n)$ were in the periodic part of the sequence, we could conclude that $n + 1$ is not a multiple of the period. The same would be true of $n + 2$, $n + 3$, ... but this is impossible.

c) MorseThueC-q24.py

d) is just a way of expressing the definition of the sequence.

MorseThueD-q24.py

e) is true because the sequence of $n+1$ digit binary numbers is obtained from the sequence of all numbers with up to n digits by putting a 1 and possibly some 0's in front of them.

MorseThueE-q24.py

23. b) One can argue the same way as for the Morse-Thue sequence above.

Keane-q25.py

24. You save about 12% in running time.

25. pascalTriangle-q27.py

Use $n = 15$ for input.

c) Replace the 5th line from below by

```
x[i] = (a+b) % 2
print(x[i])
a = b
b = x[i+1]
```

26. a)

```
for i in range(1,801):
  if (i-1)%3 == 0:
    if (i-1)%4 == 0:
      if (i-1)%5 == 0:
        if i%7 == 0:
          print(i)
```

A better approach is this: $i - 1$ is a multiple of 3, 4, 5, i.e. of 60. So we must check the terms $60i + 1$ for divisibility by 7.

```
for i in range(1,50):
  if (60*i + 1)%7 == 0:
    print(60*i+1)
```

b) One gets $n = 301 + 420t$. For $t = 0$ we get $n = 301$ eggs of weight ≈ 17 kg. For $t = 1$ we get $n = 721$ eggs of weight ≈ 40 kg, which can be carried by a strong man on his back, but not by an old woman in her basket.

c) One generalization: by taking them a, b, c at a time she got remainders r, s, t, but by taking them d at a time, it came out even:

```
for i in range(1,max+1):
  if (i-r)%a == 0:
    if (i-s)%b == 0:
      if (i-t)%c == 0:
        if i%d == 0:
          print(i)
```

If the smallest non-negative solution is s then all solutions are $x = s + t * \mathrm{lcm}(a, b, c)$ (Chinese remainder theorem.)

28. a) Let $m = a + b$. Then $b \equiv -a \pmod{m}$. In every step a mod m is replaced by $2a$ mod m. Hence the process will terminate if for some i, $2^i a$ is a multiple of m. This will be the case if and only if a is divisible by the largest odd divisor of m.

b) If a and b have a common divisor, divide it out. So we can assume with out loss of generality that $\gcd(a, m) = 1$. We get pure periodicity if for some n, $2^n a = a \pmod{m}$. This means $(2^n - 1)a$ is divisible by m. This implies $2^n \equiv 1 \pmod{m}$. So m divides $2^n - 1$ and it must be odd. Conversely, if m is odd, $2^n \equiv 1 \pmod{m}$ for some n and the least such n is the length of the cycle.

29. $f(0) = 0$, $f(n) = n \bmod 2 + f(n \mathbin{//} 2)$ for $n > 0$

30. $n = 5186$

31. involution-q33.py

B.3 Sections 2.2 to 2.4

3. pyttrip3.py

7. $c_4 = \pi/2$

8. $c_{2k} = \pi^k/k!, c_{2k+1} = 2(2\pi)^k/(1 \times 3 \times \cdots \times (2k+1))$. The volume of the unit ball approaches 0 as k goes to infinity. The proportion goes to 0 even faster.

10. $\pi = 3.1415927054$ and $\pi = 3.1415926535$. All digits of the second number are correct.

B.4 Section 2.5

1. prison-q1.py

2. The 'obvious' program for the U-sequence is

unique-q2.py

Comment: `can` is the candidate currently being tested. If it can be represented in one way ($l = 1$) then it is stored in an array a[1..n] of successful candidates.

3. An 'obvious' program is:

unique1-q3.py

Use 50 or 100 for n.

4. a) firstFib-q4.py

For b) and c) one gets the same frequencies.

5. $q^2 \nmid n$ with probability $1 - 1/q^2$ for any prime q. The probability that a 'random' integer is square free is

$$p = \prod_{\text{all primes}} (1 - 1/q^2) = 6/\pi^2.$$

6. Look up the solution of Ex. 22 of Sections 2.6 to 2.12 below. There you will find an efficient program for the Frobenius Problem.

7. c) collatz-q7.py

This is one of many versions using a procedure to get from n to its successor $f(n)$.

11. cycles-q11.py

B.5 Section 2.5.4

2. We show first that the sequence of numbers whose ternary representations contain no 2's contains no three numbers a, b, c such that $a + b = 2c$. The digits of $2c$ are all 0's and 2's. Since the digits in a and in c are all 0's and 1's, the equation could hold only if the 1's in a were in the same place as the 1's in b, which would mean $a = b$.

Next we show that every number which is not in the sequence forms an arithmetic progression with two smaller numbers which are in the sequence. Let c be a number which has one or more 2's in its ternary representation. Let b be obtained by replacing these 2's by 1's, and let a be obtained by replacing them by 0's. Then a, b, c form an arithmetic progression. Since the only numbers excluded from our sequence are the ones which form an arithmetic progression with two smaller numbers in the sequence, our sequence is the one given by the greedy algorithm.

B.6 Sections 2.6 to 2.12

1. rotation-q1.py

2. The recursive program is clear. Here is the iterative program:
bIter-q2.py

6. a) 2498 b) 1229587

7. 63992

8. 4562

9. $a(200, 8) = 104561$ with D = {1, 2, 4, 10, 20, 40, 100, 200}. Scale by 2, so that coins become integers.

10. weights-q10.py

Here $m = a_1 + \cdots + a_s$ and c[n] is the number of solutions of $a_1x_1 + + a_sx_s = n$. The numbers c[0],..., c[m] are stored in the array c[0..m].

11. pans-q11.py

This straightforward program is valid for $s = 4$ and $m = a[1] + a[2] + a[3] + a[4]$.

It is easy to see how to generalize and complete the program. Unfortunately its time complexity is $O(3^s)$. So it is only feasible for small s, say $s = 12$. The following program has complexity which is proportional to $n * s$, i.e. $O(n * s)$.

rep3-q11.py

12. c) TwoThreeFive-q12.py

17. revit-q17.py

18. a) fn-q18.py

b) In the interval 1..1988 the relation $i = f(i)$ is satisfied 92 times.

g) fn-18g.py

21. $CBA = (A^R B^R C^R)^R$. Write the corresponding program. It uses the procedure reverse four times.

22. The following algorithm can be used in the exploration of the Frobenius Problem:

frobenius-q22.py

Let $b(k,i)$ be the number of ways of representing k as a sum of numbers from among $a(1),...,a(i)$. The same summand may occur several times. Representations which differ only in the order of the terms are considered the same. Let $b(0,i) = 1$ and $b(k,0) = 0$ for $k > 0$. (This expresses the useful convention that the empty sum has value 0). A representation of k as a sum of numbers from among $a(1),...,a(i)$ is one of two kinds: either it has no summand $a(i)$ or it is obtained from a representation of $k - a(i)$ by adding $a(i)$ to it. This gives the recursion formula

$$b(k,i) = \begin{cases} b(k,i-1) & \text{if } k < a(i) \\ b(k,i-1) + b(k-a(i),i) & \text{if } k \geq a(i). \end{cases} \tag{B.6}$$

It is easy to write a program which computes the lists $b[k][1], b[k][2], ...$ by means of Equ. (B.6). Observe that once $b[k][i]$ has been computed, $b[k][i-1]$ is no longer needed so instead of a two-parameter list b[k][i] we can use a 1-parameter list b[0..n] and recompute it for successive values of i. In this way we get the algorithm above.

23. Let $p(k,i)$ be the number of ordered i-tuples of decimal digits with sum k. We have $p(0,i) = 1$ for $i = 0,1,...$ and $p(k,0) = 0$ for $k = 1,2,....$ It will simplify our formulas to define $p(k,i) = 0$ for $k = -1,-2,...,-9$ and all i. A recursive formula for $p(k,i)$ $(k,i > 0)$ can be obtained as follows: the i-tuples with sum k and last digit 1, ..., 9 are obtained from the i-tuples with sum $k-1$ and last digit 0, ..., 8 by adding one to the last digit. The number of the latter i-tuples is $p(k-1,i) - p((k-1)-9,i-1)$. The number of i-tuples with sum k and last digit 0 is $p(k,i-1)$. Combining these facts we get

$$p(k,i) = p(k-1,i) + p(k,i-1) - p(k-10,i-1), \quad k,i > 0. \tag{B.7}$$

We know the number of ordered triples with sum k is the same as the number of ordered triples with sum $27-k$. Hence the number of 6-tuples we want is the same as the number of 6-tuples such that the sum of all 6 digits is 27; we get one kind from the other by replacing each of the last 3 digits d_i by $9-d_i$. Thus $p(27,6)$ is an answer to our question.

equisum6-q23.py

B.7 Sections 2.13 to 2.14

8. chain-q8.py

By means of `chain-q8.py` we get the pairs of primes in Table B.1.

We can speed up `chain-q8.py`. Change the loop as follows:

```
while (not a) or (not b):
  p += 2
  a = prime(p, 3)
  if a:
    b = prime(p+p+1, 3)
```

max	p	2p+1
1000	1013	2027
10000	10061	20123
100000	100043	200087
1000000	1000151	2000303
10000000	10000079	20000159
100000000	100000223	200000447

Table B.1. Outputs for various max values

In the program each time both p and $2p+1$ are tested for primality. In the modified program, $2p+1$ is only tested if p is prime, and we must declare the boolean variables a and b.

9. The Euler polynomial $f(x) = x^2+x+41$ is especially rich in primes. From 0 to 2398 there are 1199 primes, i.e. 50%. From 0 to 4000 there are 1860 primes, i.e. 46.5%.

EulerPolynomial-q9.py

16. $n = 4^k(8q+7)$.

17. a) $8q+7$ is not the sum of 3 squares.

b) If $4^k(8q+7)$ is a sum of 3 squares then so is $4^{k-1}(8q+7)$

19. taxi-q19.py

20. $6578 = 1^4 + 2^4 + 9^4 = 3^4 + 7^4 + 8^4$

21. sieveout-q21.py

```
> python  sieveout-q21.py
n=? 100
3 5 7 8 9 11 13 14 15 17 19 20 21 23 24 25 26 27 29 31 32
33 34 35 37 38 39 41 43 44 45 47 48 49 50 51 53 54 55 56
57 59 61 62 63 64 65 67 68 69 71 73 74 75 76 77 79 80 8
1 83 84 85 86 87 89 90 91 92 93 94 95 97 98 99
```

Transform the term $z+y+xy$ so that you can see which integers are contained in this list. The output is for $n = 100$.

B.8 Section 2.16

4. coinStack-q4.py

B.9 Chapters 1 to 2

1. skip-q1.py

This program lists the numbers skipped by our function. For input 1000 it gives the squares up to 961. Show that exactly the squares are skipped!

2. The function skips the triangular numbers, (numbers of the form $n(n+1)/2$).

3. The function skips the numbers $(kn-k+1)n+\lfloor(k-1)/4\rfloor$.

4. a) The function skips the numbers $\lfloor(n+k-1)n)/k\rfloor$.

b) Combine the results in 3. and 4 a).

6. stirling-q6.py

7. sum3Cubes-q7.py

Here x, y, z are non-negative integers. To see that numbers of the form $9n\pm4$ are not representable, consider $x^3+y^3+z^3$ modulo 9.

10. Gauss has shown that all positive integers can be represented as a sum of at most three triangular numbers. The following program checks this up to $n = 32767$.

triangle-q10.py

11. In the program mode we find the mode of the increasing sequence a[i] = $\lfloor\sqrt{i}\rfloor$:

mode-q11.py

12. commons-q12.py

13. c) A closed formula for g is $g(n) = \lfloor(n+1)t\rfloor$ with $t = (\sqrt{5}-1)/2$.

gfn-q13.py

14. c) The points $(n, h(n))$ are well approximated by a straight line $y = tx$ through the origin. By substituting $h(n) \approx tn$ into the functional equation for h we get $tn \approx n - t^3(n-1)$ or $t^3+t \approx 1+t^3/n$. For $n \to \infty$ we see that t must satisfy $t^3+t-1=0$. bis.py gives t = 6.8232780381E-1. So $h(n)$ = round(nt) and $h(n)$ = trunc$((n+1)t)$ are possible candidates. Up to $n = 16000$, $h(n)$ = round$(n*t)$ is correct in about 85% of all cases and deviates from the correct value of $h(n)$ by +1 or -1 in the remaining cases. trunc$((n+1)*t)$ deviates from $h(n)$ in over 20% of all cases by +1 or -1. Can you find an exact formula for $h(n)$?

16. sextuple-q16.py

17. eulerReal-q17.py

x	1	1	1	3	1	5	7	3	17	11	23	45	91	89	93	271	85
y	3	1	5	1	11	9	13	31	5	57	67	47	87	275	449	101	999
n	4	3	5	6	7	8	9	10	11	12	13	14	16	17	18	19	20

Table B.2. Output from eulerReal-q17

By running `eulerReal.py` we get the data in Table B.2 that any x is one half of the sum or difference of the two preceding x, y. From the current x, y you get the next y by computing $(7x+y)/2$ or $|7x-y|/2$. If the next x is a sum then the next y is a difference, and vice versa. That is we have the two transformations S,

T:

$$S : (x, y) \to ((x + y)/2, |7x - y|/2), \quad T : (x, y) \to (|x - y|/2, (7x + y)/2)$$

Prove this by induction! You will have to prove in addition that one of the two new pairs consists of odd elements.

21. MinComElem-q21.py

24. floyd-q24.py

29. a) 381654729 b) 3816547290

30. bellTriangle-q30.py

For $k = 0, 1, ..n$, let $B_{n,k}$ be the number of partitions of {1, ..., n} such that the other elements of the set to which n belongs are all $\leq$ k. We have $B_{n,0} = B_{n-1}$ and $B_{n,n-1} = B_n$.

The partitions counted by $B_{n,k+1}$ are of two kinds. First, those in which the element $k + 1$ is not in the same set as n. There are $B_{n,k}$ such partitions. Second, those in which $k + 1$ and n are in the same set. Removing the element $k + 1$ from all of these, we see that the number of these partitions is $B_{n-1,k}$. Thus $B_{n,k+1} = B_{n,k} + B_{n-1,k}$.

32. c) Both are correct.

33. olympiad-q33.py

35. Each number occurs twice except 1, which occurs four times, and the other powers of 2, which occur three times.

B.10 Section 3.10

1. The following generates a random permutation: randPerm-q1.py

B.11 Section 3.11

1. Since the period is $p - 1$, all of $1, 2, ..., p - 1$ occur as remainders if we divide a positive integer which is $< p$ by p. The digits we obtain from these remainders are the integer parts of $10/p, 20/p, ..., 10(p-1)/p$. The number of times we get the digit d is equal to the number of multiples of 10 in the interval $dp \leq x < (d+1)p$. That number is at least $\lfloor p/10 \rfloor$ and at most $\lceil p/10 \rceil$. The proof for pairs, triples etc. is similar.

B.12 Section 3.12

2. The pure periodicity follows from the fact that the stream of digits can be uniquely extended into the past.

3. a) Consider the sequence mod 2: 11011 11011 11011 .... The numbers 1,2,3,4 taken mod 2 are 1,0,1,0. This pattern does not occur.

b) This is 1,0,0,1 mod 2, and so it cannot occur.

c) Extend the sequence one step into the past and you get 519831138....

d) This block will occur again since the sequence is purely periodic.

4. $x(n) = (x(n-3)+x(n-5)) \bmod 2$ and $y(n) = (4y(n-4)+y(n-5)) \bmod 5$ have periods 2^5-1 and 5^5-1. The period of the resulting sequence is the least common multiple of these two numbers. Since they are prime to each other it is their product 31 * 3124 = 96844.

5. The combined sequence has period $(2^{17}-1)(5^7-1) = 10239790804$.

B.13 Section 3.15

1. Suppose that after the first transformation the largest value occurring among the x_i is M and that a smaller positive value also occurs. If only one such value occurs then after the next step there will be two such values next to each other. Let $x_i, x_{i+i}, ..., x_j$ be an interval of maximal length such that it begins and ends with a positive entry < M and none of the entries in between is = M. Then if we apply the transformation once more, the interval $x'_{i-1}, x'_i, ..., x'_j$ will have positive entries < M at both ends and no entry M inside. Thus the length of the longest such interval increases by at least 1 at each step and after at most $n-1$ steps no entry M is left. We see that after at most $n(M-1)$ steps we shall reach a set such that all the positive x_i have the same value.

3. If the polygonal set x consists of 0's and 1's, then Tx also consists of 0's and 1's, and the number of 1's is even. Also, if n is odd then T has an inverse on the set X_0 of x with an even number of 1's. It follows that for odd n all $x \in X_0$ are on cycles.

4. Applying the transformation T twice to a cycle with an even number of entries is the same as applying it once to the cycle consisting of the even numbered entries and once to the cycle consisting of the odd numbered entries. Hence $2c(n)$ is a T-period for any $2n$-component cycle. To see that in some cases the T-period is not shorter than $2c(n)$, use that if the even numbered entries in the original cycle are all 0, the odd numbered entries change only in every second step.

8. a) The following program plays Bulgarian solitaire, starting with any number of stacks of any sizes. It stops when, arranged in increasing order, the i-th stack contains i or $i+1$ cards, and it prints out the stacks in increasing order.

bulsolit-q8.py

B.14 Chapters 1 to 3

3. We get the infinite binary word as follows: $w_1 = 0, w_2 = 001, w_3 = w_2w_2w_1$. By induction we get $w_{k+1} = w_kw_kw_{k-1}$. Let a_k and b_k be the number of zeros and ones in w_k, Then $a_{k+1} = 2a_k + a_{k-1}$, $b_{k+1} = a_k$, $t_k = a_k/a_{k-1}$, $t_{k+1} = a_{k+i}/a_k =$

$2 + 1/t_k$, For $k \to \infty$ we get $t = 2 + 1/t$, or $t = \sqrt{t} + 1$. The ratio a_k/b_k tends to an irrational limit. Thus the sequence is not periodic. If it were periodic, t_k would tend to the rational ratio of zeros to ones in one period. For the infinite binary word we have zeros/ones $= (\sqrt{2} + 1)/1$, zeros/(zeros+ones) $= (\sqrt{2}+1)/(2+\sqrt{2}) = 1/\sqrt{2}$, ones/(zeros+ones) $= 1/(2 + \sqrt{2})$. So every $(2 + \sqrt{2})$th digit is a 1. The n-th 1 should have place number $\approx (2 + \sqrt{2})n$. We find empirically that the greatest integer part of the last expression is the place number of the n-th 1. This is tested by the following program. It first constructs n bits of the word. For input u it then finds the place number i of the u-th 1 and compares with f(u) = trunc($(2 + \sqrt{2})$ * u).

BinWord-q3.py

We also give a proof that always $i =$ trunc($(2 + \sqrt{2})n$). Let S be the operator which performs the following substitutions on binary sequences: it replaces each 0 by 001 and each 1 by 0.

Lemma 1. For any sequence D $= d_1...d_n$ of binary digits the proportions of 0's among the digits of D and S(D) are on opposite sides of $\frac{1}{\sqrt{2}}$.

Proof. Let the number of 0's in D be z. Let n' be the number of digits in S(D) and let z' be the number of 0's among them. Then

$$n' = n + 2z, \quad z' = n + z. \tag{B.8}$$

It is easy to check that the ratio $\frac{s}{n}$ would remain unchanged by S if it had the value $\frac{1}{\sqrt{2}}$. Since this is irrational the ratio can not have this value for a finite sequence, but it is instructive to examine how the difference $z - \frac{n}{\sqrt{2}}$ is changed by the substitution S. We find

$$z' - \frac{n'}{\sqrt{2}} = -(\sqrt{2} - 1)(z - \frac{n}{\sqrt{2}}). \tag{B.9}$$

So S not only changes the sign of the difference $z - \frac{n}{\sqrt{2}}$ but also decreases its magnitude by the factor 0.414.... Interestingly, this is the reciprocal of the limit of the factors by which S increases the lengths of sequences as the proportion of 0's approaches the stable value. In the case of the initial segments 0, S(0), S(S(0)), ... of our sequence, the ratios $\frac{n}{z}$ are the continued fraction approximants of $\sqrt{2}$, as one can deduce from the recursion formulas.

Lemma 2. Let $I_n = a_1...a_n$ be an initial segment of our sequence. Let z_n be the number of 0's in the segment. Then $z_n > \frac{n}{\sqrt{2}}$ if $a_n = 0$ and $z_n < \frac{n}{\sqrt{2}}$ if $a_n = 1$.

Roughly speaking, this says that the ratio of digits hovers so close to the invariant ratio that the last digit can tip it in its own direction.

Proof. One can verify the claim for small values of n. Assume $n > 12$ and that Lemma 2 has been proved for $I_1, ..., I_{n-1}$.

If $a_n = 1$ then $I_n = S(I_m)$ for some $m < n$, and the last digit of I_m is 0. Then Lemma 2 follows from the induction hypothesis and Lemma 1. If I_n terminates with a 0, there are two possibilities:

a) $I_n = \mathrm{S}(I_m)$ where I_m ends with 1. Here the induction hypothesis and Lemma 1 again give the required result.

b) The final 0 of I_n is the first or the second 0 of a 001 which comes from substituting for a 0 in a precursor initial segment. For definiteness let it be the first 0. Then $I_n = (\mathrm{S}(I_m))0 = \mathrm{S}(I_n 1)$. Think of the sequence $I_m 1$ as the initial segment up to and including the first 1 beyond a_m, with the 0's beyond a_m deleted. Then we see that the induction hypothesis implies that the proportion of 0's in $I_m 1$ is $< \frac{1}{\sqrt{2}}$ and hence by Lemma 1 the proportion in I_n is $> \frac{1}{\sqrt{2}}$. The case when the final 0 of I_n is the second 0 of a 001 can be dealt with similarly.

We are ready to prove our formula for the number of digits n up to and including the k-th 1. Lemma 2 tells us $k > (1 - \frac{1}{\sqrt{2}})n$. Also, $a_{n+1} = 0$ since every 1 in the sequence is preceded by at least two 0's. Applying Lemma 2 to I_{n+1} we get $k < (1 - \frac{1}{\sqrt{2}})(n+1)$. We can write these two inequalities as $n < k(2+\sqrt{2}) < n+1$, q.e.d.

For a lattice point interpretation of this problem, see the section on continued fractions in Felix Klein's *Elementary Mathematics from an Advanced Standpoint* [**Kle12**].

7. equisum-q7.py

9. $n = 2053$.

10. die100-q10.py

B.15 Sections 4.5 to 4.6

1. Represent a nonincreasing sequence of at most m numbers, each of which is $\leq n$, by columns of unit squares with bases on the x-axis. If we look at such a figure sideways, it becomes a representation of a nonincreasing sequence of at most n numbers, each of which is $\leq m$.

2. TwoSamp-q2.py

5. TwoSamit-q6.py

B.16 Section 4.13

1. TwoCompleteSets-q1.py

4. CoverMultCase-q4.py

5. `CoverHypgCase.py` is obtained from CoverMultCase-q4.py by setting up a list of occupied squares and rejecting a random choice if the square is already occupied.

B.17 Section 4.14

tripleBirthday-q2.py

Ten repetitions of this program gave 88.77 for the expected waiting time.

B.18 Chapters 3 to 4

1. craps-q1.py

2. CoinGame-q2.py

3. $E(D_n^2) = n$.

4. $E(D_n^2) = n$.

5. $E(D_n^2) = n$.

randomWalk-q5.py

6. About 0.495.

lotto-q6.py

7. $E(D_n^2) = n$.

8. randDirWalk-q8.py performs a random walk of n steps m times and finds the mean of the squares of the distances from the origin at the end of each walk. Choose $m = 1000$ and $n = 10, 20, 50, 100$ and you will realize that the expectation of the square of the distance is n. Our interpretation of random choice of a point on a unit sphere S is this: it falls into a subset M of S with probability area(M)/4π. We use the theorem of Archimedes that a spherical zone of height h on a sphere of radius 1 has area $2\pi h$. We choose h at random from (-1, 1). The geographical longitude a of the point is chosen by means of a = 2π*random(). If we set $r = \sqrt{1 - h * h}$, then $x = x + r * \cos(a), y = y + r * \sin(a), z = z + h$.

We find that $E(D_n^2) = n$.

10. PokerTest-q10.py

For $n = 10000$ we get 3069 5063 1046 679 95 47 1.

15. a) palindrome-q15.py generates m n-words from the alphabet {0, 1, 2} and counts those which start with a palindrome. The proportion of such words is an estimate for P_n. Table B.3 can be used for checking the program:

16. totalTourCube-q16.py

The exact value is 1996/95 = 21.0105.

B.19 Chapters 1 to 4

2. The formula is $f(a, b, c) = 3abc/(a + b + c)$. Lennart Råade of Sweden put this problem to a number of people in the last 20 years, but nobody could solve it. In 1992, I (Engel) ran a version of ThreeTowerGame-q2.py, and its output suggested the formula which I then proved. (See the end of Appendix A.)

n	Pn
2	1/3
3	5/9
4	$17/27 = 0.63$
5	$55/81 = 0.679$
6	$169/243 = 0.6955$
7	$517/729 = 0.70919$
8	$1561/2187 = 0.71376$
9	$4709/6561 = 0.7177$
10	$14153/19683 = 0.71905$
11	0.7203
12	0.72072
14	0.72125
16	0.72142
18	0.721481581

Table B.3. Output from palindrome-q15.py.

B.20 Section 5.4

2. We assume the random sets are chosen by flipping a fair coin to decide for each element whether it should be in the set. After the first guess, G is in a set containing r other numbers which were picked at random from the set of $n-1$ numbers other than G, etc. The guessing ends when $r = 0$.

RandGuess-q2.py

B.21 Section 5.5.5

1. knights-q1.py

4. The following program tries to find a knight's path visiting every square of an $(m-2)$ x $(n-2)$ chessboard, using Warnsdorf's method. Warnsdroff-q4.py

The list b[i][j] will contain the ordinal numbers of the squares in the path. The list is bordered with two dummy rows on all 4 sides. This enables us to list the squares accessible by a knight jump from a given square (of the actual chessboard) in the same way, whereas otherwise we would have to do it differently for squares near or on the edge. We initialize b[i][j] to be 0 on the chessboard proper and 1 on the border rows. This will cause the algorithm not to put squares of the border rows into the knight's path.

At each stage the algorithm starts from the current position (i, j). It checks all squares reachable by changing the coordinates by at most 2. If s, t are the amounts added to the coordinates, the condition for reachability by knight jump has the simple form $|st| = 2$. For each square reachable from (i, j), we check how

many squares reachable from it are still free. We select as the next square the one with coordinates `imin`, `jmin`, which is a free square reachable from (i, j) and has fewer free neighbors than any of the squares previously tried, and not more than any tried after it. In the program, c becomes the number of free neighbors of a square, and `min` becomes the least value of c among the neighbors of (i, j).

The construction terminates once we have selected a free square with no free neighbors. If this happens before $m \cdot n$ squares have been covered, the algorithm has failed.

Note that if the current square has several free neighbors with `min` free neighbors of their own, choosing the first one our algorithm finds is arbitrary; one could make these choices at random. Then a second run may succeed if the first one fails.

A related approach tries to find a closed knight's tour by using Warnsdorf's selection criterion, and backtracking when it gets stuck.

B.22 Section 5.6

3. savinqBoxes-q3.py

By simulation we find $p(n, k) = k/n$. Here is one of the proofs from the report on the competition in the Hungarian high school periodical *Matematikai Lapok*.

We ignore k for the time being. We consider the following procedure for opening all boxes. Break box 1. Open as many boxes with keys as you can. The last of those has key 1 in it. Now break the lowest numbered box which is still locked, open all the boxes you can, etc.

Let us call the order in which the keys appear in this procedure the *retrieval permutation* P' corresponding to the *hiding permutation* P. Given a retrieval permutation P', we can reconstruct the hiding permutation P which produces it. Hence every permutation of keys is a possible retrieval permutation. If all permutations are equally likely to occur as hiding permutations then they are also equally likely to occur as retrieval permutations.

We note that the key in the last place in a retrieval permutation is the key of the last box which was broken. Let the number of that key be l. When we broke box l, all boxes with smaller numbers were already open, so to open all boxes we needed to break one whose number was $\geq l$.

Let us return now to the opening stage which consists of breaking the first k boxes at the start and not breaking any more. We can open all boxes if and only if the last entry l of the retrieval permutation is one of $1, 2, ..., k$. The probability of this is k/n.

6. The following does the computations for a), b), c) and f).

GoldPerm2-q6.py

B.23 Section 5.8

2. c) Let

$$x(n) = \lfloor nt \rfloor, \quad y(n) = n + \lfloor nt \rfloor = \lfloor n(1+t) \rfloor = \left\lfloor nt^2 \right\rfloor$$

We need to show only that the sequences $x(n)$ and $y(n)$ together contain every positive integer exactly once. This is an instance of S. Beatty's theorem. One way of proving it is as follows. For a given positive integer n, the number of terms $< n$ in the sequence $t, 2t, 3t, ...$ is $\lfloor n/t \rfloor$. The number of terms $< n$ in $t^2, 2t^2, 3t^2, ...$ is $\lfloor n/t^2 \rfloor$. Since t is irrational, n/t and n/t^2 are both irrational, but $n/t + n/t^2 = n(1+t)/t^2 = n$. Thus

$$\lfloor n/t \rfloor + \left\lfloor n/t^2 \right\rfloor = n - 1.$$

This is the number of terms in the two sequences taken together. By taking $n = 1, 2, 3, ...$ we see that there is exactly one multiple of either t or t^2 in each of the intervals $(1, 2), (2, 3), ...$ Hence every positive integer is in exactly one of the sequences $\lfloor nt \rfloor, \lfloor nt^2 \rfloor$.

4. L consists of all multiples of $k + 1$ including 0.

5. L consists of all multiples of 3 including 0.

6. L consists of all multiples of 4 including 0.

12. L consists of all multiples of 6 including 0.

B.24 Section 6.2

13, 14. These computations are much easier in languages which support integer arithmetic with arbitrarily large integers.

B.25 Section 12

1. (a) The transition probabilities for this Markov chain with three states are as follows:

```
               POOR SATISFACTORY PREFERRED
POOR            0.6    0.4          0
SATIISFACTORY   0.1    0.6         0.3
PREFERRED        0     0.2         0.8
```

so that the transition probability matrix is

```
    0.6  0.4   0
P = 0.1  0.6  0.3
     0   0.2  0.8
```

(b) We will find the limiting fraction of drivers in each of these categories from the components of the stationary distribution vector π, which satisfies the

following equation:

$$\pi = \pi P.$$

The former is equivalent to the following system of linear equations:

$$\begin{aligned} \pi_1 &= 0.6\pi_1 + 0.1\pi_2 \\ \pi_2 &= 0.4\pi_1 + 0.6\pi_2 + 0.2\pi_3 \\ \pi_3 &= 0.3\pi_2 + 0.8\pi_3 \\ 1 &= \pi(1) + \pi_2 + \pi_3. \end{aligned}$$

This has the following solution: $\pi = (1/11, 4/11, 6/11)$.

Thus, the limiting fraction of drivers in the POOR category is 1/11, in the SATISFACTORY category 4/11, and in the PREFERRED category 6/11. By the way, the proportions of the drivers in each category in 15 years approximate these numbers with two significant digits (you can check it, calculating P^{20} and looking at its rows).

Use `steadyTests.py` and `powersTest.py` on `cars.txt`.

3. The transition matrix is

```
       GO   A   B   C
   GO  1/6  1/3 1/3 1/6
P=  A  1/6  1/6 1/3 1/3
    B  1/3  1/6 1/6 1/3
    C  1/3  1/3 1/6 1/6
```

$\pi = (1/4, 1/4, 1/4, 1/4)$ using `steadyTests.py` on `monopoly.txt`.

From this we see that $\pi \cdot f = 0$ where

```
    15
f= -30
    -5
    20
```

4. We use a Markov chain model with 3 states, H, M, and E, where the state reflects the difficulty of the most recent exam. We are given the transition probabilities:

```
    H     M     E
H   0    0.5   0.5
M  0.25  0.5   0.25
E  0.25  0.25  0.5
```

It is easy to see that our Markov chain has a single recurrent class, which is aperiodic. The balance equations take the form

$$\begin{aligned} \pi_1 &= 0.25\pi_2 + 0.25\pi_3 \\ \pi_2 &= 0.5\pi_1 + 0.5\pi_2 + 0.25\pi_3 \\ \pi_3 &= 0.5\pi_1 + 0.25\pi_2 + 0.5\pi_3 \end{aligned}$$

and solving these with the constraint $\sum_i \pi_i = 1$ gives

$$\pi_1 = \frac{1}{5} \quad \pi_2 = \pi_3 = \frac{2}{5}$$

or use steadyTests.py on exams.txt.

5. Snakes and Ladders.

Run absorbSnakes.py.

Expected time to absorption from square 5 is 8.36

6. It's easy to see that the Markov chain described by the transition diagram in Fig. B.1 captures exactly what we are looking for.

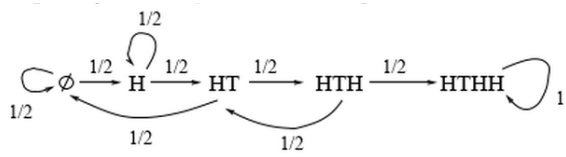


Figure B.1. Reaching HTHH

Rename the states 0, H, HT, HTH, HTHH as 0, 1, 2, 3, 4 respectively. and let v_i be the average number of steps required for the state 4 to be reached if the starting state is i. Writing first-step (backwards) equations we have

$$\begin{aligned} v_0 &= 1 + 0.5v_0 + 0.5v_1 \\ v_1 &= 1 + 0.5v_1 + 0.5v_2 \\ v_2 &= 1 + 0.5v_0 + 0.5v_3 \\ v_3 &= 1 + 0.5v_3 + 0.5v_4 \end{aligned}$$

Also. obviously, $v_4 = 0$. Solving, we find

$$v_3 = 8, \quad v_2 = 14, \quad v_1 = 16, \quad v_0 = 18.$$

So the answer is: "it takes on average 18 coin tosses to see the pattern HTHH for the first time".

Or run absorbFair.py. with fair.txt. and look at t vector

7. Say that a score $x - y$ means that the server has x points and the other player y. If the current score is $3 - 3$ the next score is either $4 - 3$ or $3 - 4$. In either case, the game must continue until one of the players is ahead by 2 points. So let us say that i represents the difference $x - y$. We model the situation by the Markov chain in Fig. B.2.

Let v_1 be the probability that the server wins, i.e. that state 2 is reached before state -2. First-step equations yield:

$$v_i = 0.6v_{i+1} + 0.4v_{i-1}, \quad -1 \le x \le 1$$

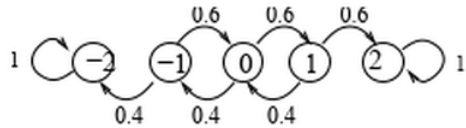


Figure B.2. Playing Tennis

In other words,

$$\begin{aligned} v_{-1} &= 0.6v_0 + 0.4v_{-2} \\ v_0 &= 0.6v_1 + 0.4v_{-1} \\ v_1 &= 0.6v_2 + 0.4v_0 \end{aligned}$$

Of course,

$$v_{-2} = 0, \quad v_2 = 1$$

Solving, we find

$$v_0 = \frac{0.6^2}{1 - 2 \times 0.6 \times 0.4} \approx 0.69, \quad v_1 \approx 0.88, \quad v_{-1} \approx 0.42$$

Or run absorbTests.py on tennis.txt. and look at second column of B

8. (a)

```
        1    2    0    3
   1    0   2/3  1/3   0
P = 2   2/3  0    0    1/3
   0    0    0    1    0
   3    0    0    0    1
```

(b)

```
        1     2
N = 1 9/5    6/5
   2 6/5    9/5

        0     3
B = 1 3/5   2/5
   2 2/5   3/5

t = 1  3
   2  3
```

Obtained by running absorbTests.py on spinner.txt.

(c) The game will last on the average 3 moves.

(d) If Mary deals, the probability that John wins the game is 3/5.

9. The problem can be solved by writing first step equations for the Markov chain representing the number of red cards remaining.

The transition probabilities are:

$$p(3,2) = 3/6, \quad p(2,1) = 2/5, \quad p(1,0) = 1/4,$$
$$p(i,i) = 1 - p(i,i-1), \quad i = 3,2,1,$$
$$p(0,0) = 1$$

Let v_i be the average number of steps till the process ends if the initial state is i. Then

$$\begin{aligned} v_3 &= 1 + 3/6v_3 + 3/6v_2 \\ v_2 &= 1 + 3/5v_2 + 2/5v_1 \\ v_1 &= 1 + 3/4v_1 \end{aligned}$$

Alternatively run absorbReds.py on reds.txt to get $v_3 = 8.5$.

B.26 Chapters 1 to 7

1. Let the number of people who have heard the joke be `inf` and the number of people who have told it to someone who heard it `em`. (`inf` stands for infected, to indicate similarity to models of epidemics, and `em` stands for embarrassed, for having told a familiar joke.)

The next person to be told could be one who heard it already, other than the person telling it, or someone new. We model this by taking an integer at random from 2..n. If the number is $\leq$ `inf` we add 1 to `em`, otherwise we add 1 to `inf`. When `em` = `inf` the process stops and we record how many people have not heard the joke. The experiment is repeated m times. 11 repetitions of Joke-q1.py gave the median output 0.2035 and mean output 0.2032. That is, about 20% of the population does not hear the joke.

If n is large and we set `inf/n` = x and `em/n` = y then the two quantities grow approximately in accordance with the differential equation $dy/dx = z/(1-z)$, $y(0) = 0$, until y becomes equal to x, which happens when $2x = \ln(1-x)$. Then $1-x$, the proportion of people who did not hear the joke, is 0.2031878699799799...

2. You will get all numbers of the form $2^m 3^n - 1$.

5. The following program checks Watson's claim with interpretation a):

checkit-q5.py

We have 33/70=0.47142857.... On the other hand, five runs of with max=10000 resulted in 0.4564, 0.4549, 0.4637, 0.4600, 0.4564, which is close to, but consistently below 0.47. So this does not seem to be the right interpretation.

By trying interpretation b) we get 0.472. This seems to be the correct interpretation. One can get the exact value by algebraic computation. One can show that the required probability is the same as for the following simpler problem: let S be a sphere with north pole N. Pick two points A, B inside S at random. Find the probability that $\bigtriangleup NAB$ is acute. If one takes the north pole as the origin of polar coordinates, evaluating the triple integrals representing the probability of

getting an obtuse angle at the north pole, or at another vertex, becomes a laborious exercise in third-term calculus.

Restricting the random points to a cube or a sphere is arbitrary. The following argument seems to reduce the completely unrestricted problem to the simplified problem above.

Let A be that one of the three points chosen which is farthest from the origin 0. Then all we know about B and C is that they randomly chosen points inside the sphere about O with radius OA. Thus the probability that $\bigtriangleup ABC$ is acute is the same as the probability that a triangle formed by the North Pole of a sphere and two points chosen at random inside it is acute.

The weakness of this argument is that it tacitly uses the notion of selecting a point at random from an infinite space, without giving preference to any part of the space. Assuming that this can be done leads to contradictions. For example, take the two dimensional case of our problem. Let A be the first point we pick, B the second, C the third. Consider the probability P that the angle at C is $> 45°$. This will happen if C is in the union of the interiors of the two circles through A and B with radii $AB/\sqrt{2}$. We can place infinitely many disjoint copies of this region in the plane. The probability that C will be in any one of these ought to be P also. The sum of these probabilities must be < 1, hence P can not be positive, it must be 0. Thus the probability that the angle at C is $> 45°$ is 0. But there is nothing special about the last point C in our problem. The three points could be selected by three people simultaneously, and they could tell us in random order. So the probability that a random triangle has any angle $> 45°$ is 0, which is absurd.

6. The following program first computes the list f[1..n]. Then for input q, it prints out $f(q)$.

selfDescrSeq-q6.py

We get $c * 10000^d \approx 356$, $c * 5000^d \approx 232$. Hence $2^d \approx 356/232 = 89/58$, $d \approx \ln(89/58)/\ln(2) = 0.61775...$, $c \approx 356/10000^d = 1.23035....$ So $f(n) \sim 1.23035n^{0.61775}$. Check $f(n) = \lfloor cn^d + 0.5 \rfloor$ with the correct value of $f(n)$. Can you do better?

8. a) mergeSortComp-q8.py

b) $\text{com(n)} = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$

11. ThreePlayers-q11.py

Simulation did not help with four players.

12. intertwinedFunctions-q12.py

If you plot the functions $F(n)$ and $M(n)$ you will observe that both are strongly clustering about the same straight line through the origin: $F(n) \sim tn, M(n) \sim tn$. For t we get $tn = n - t^2(n-1)$, or, for large n, $t^2 + t = 1$ with the solution $t = (\sqrt{5}-1)/2$. The program finds $F(i), M(i), \lfloor it + 0.5 \rfloor, \lfloor (i+1)t \rfloor$ from k to n. The agreement is exact with a few exceptions, when the difference between any pair is at most 1.

14. a) $g(a,b,c) = ab + bc + ca$;
piles-q14a.py
b) $g(a,b,c,d) = ab + ac + ad + bc + bd + cd$;
c) $g(x_1,...,x_n) = \sum_{i<k} x_i x_k$
15. isPoly-q15.py
19. trans-q19.py
20. The point (x,y) can be reached from $(1,1)$ if and only if gcd(x,y) $= 2^n$, $n = 0,1,2,...$
22. alg-q22.py
24. armstrong-q24.py
26. twinperm-q26.py
26. b) p[10000] $= 0.36787024156 \approx e^{-1} = 0.36787944117....$ This is also the probability of getting a random permutation of {1,..., n} without a fixed point. See Ex. 3 of this section. But the convergence to the asymptotic value e^{-1} is slower.
27. For $z = 26861$ we have $\pi_1(x) = 1473$, $\pi_3(x) = 1472$
28. c) (1) is exact for $n \geq 1 + s(b-1)$, where $2s$ is the word length and b is the base.
32. pentagon-q32.py picks the negative terms to be inverted in random order.

We find that the algorithm always stops and the number of steps is $20n - 10$ no matter how we pick the next operation, although the sets of numbers which come up do depend on that.

The key to the proof is to find an integer-valued, non-negative function f$(x_1, x_2, x_3, x_4, x_5)$ of the vertex labels whose value decreases when the given operation is performed. All but one of the students who solved the problem found the same function,

$$f(x_1,x_2,x_3,x_4,x_5) = \sum_{i=1}^{5} (x_i - x_{i+2})^2, \quad \text{where } x_6 = x_1,\ x_7 = x_2.$$

If, say, x_4 is flipped, $f_{\text{new}} - f_{\text{old}} = 2sx_4 < 0$. A decreasing sequence of non-negative integers must be finite, so the algorithm must stop.

Bernhard Chazelle (Princeton) gave an argument which shows why the number of steps is determined by the initial values. He considers the infinite multiset S of all sums $s(i,j) = x_i + \cdots + x_j$ with $1 \leq i \leq 5$ and $i \leq j$, where $x_6 = x_1$, etc. A multiset is a set which can have equal elements. In this multiset one element changes from negative to positive and all the others are either unchanged or switched with others when we perform an operation. For instance, if x_4 is changed from negative to positive, and the new values are denoted by primes, $s'(4,4) = -s(4,4)$ changes from negative to positive whereas $s'(4,5) = s(5,5)$, $s'(5,5) = s(4,5)$, $s'(4,6) = s(5,6)$, $s'(5,6) = s(4,6)$, etc. There are only finitely

many negative elements in S, since $s > 0$. The number of steps until the finish is the number of negative elements of S. The x_i need not be integers for this argument to be valid.

Appendix C

Summary of Notation and Formulas

- $a|b$: *a divides b*, i.e. b is a *multiple* of a for integers a, b.
- $a..b$: the set $\{a, a+1, ..., b-1, b\}$ of integers.
- $\lfloor x \rfloor$: *floor* of x, i.e. the largest integer $\leq x$. Usually denoted by $[x]$.
- $\lceil x \rceil$: *ceiling* of x , i.e. the smallest integer $\geq x$.
- gcd(a, b): *greatest common divisor* of the integers a, b.
- lcm(a, b): *least common multiple* of the integers a, b.
- $p \Rightarrow q$: The statement p *implies* the statement q.
- $p \Leftrightarrow q$: p is true *if and only if* q is true.
- $\mapsto$: mapping or function symbol. $a \mapsto b$: a is mapped into b.
- $f(n) \sim g(n)$: $f(n)$ is *asymptotically equal* to $g(n)$, i.e. $f(n)/g(n) \to 1$ for $n \to \infty$.
- $f(n) = O(g(n))$: There is a constant C such that $|f(n)| \leq Cg(n)$.
- *Harmonic series*: $1 + \frac{1}{2} + \frac{1}{3} + \cdots$. The n-th *harmonic number* $H_n = 1 + \frac{1}{2} + \cdots \frac{1}{n}$ is the n-th partial sum of the series.

- $\phi(n) = n\prod_p(1-1/p)$, where the product ranges over all distinct prime factors of n. This is the number of integers in $1..n$ which are relatively prime to n. $\phi(n)$ is called *Euler's phi function*.
- $\sum_{n\geq 1} 1/n^2 = \pi^2/6$, a classic result due to Euler.
- $\binom{n}{s}$: the number of s-subsets of an n-set

$$\binom{n}{s} = \frac{n}{s}\binom{n-1}{s-1} = \frac{n(n-1)...(n-s+1)}{s(s-1)...1} = \frac{n!}{s!(n-s)!}$$

is called *n choose s*, or the *binomial coefficient n over s*. Here $n!$, pronounced *n factorial*, is the product $1\cdot 2\cdot 3\cdots n$, with $0! = 1$.

Appendix D

A Summary of Python

As we said in the preface, we've assumed that you'd already been introduced to Python before you picked up this book. So the following amounts to a 'cheat sheet' for the Python features used here. Many Python topics aren't covered, such as class inheritance and decorators, since we don't use them. The appendix ends with three examples to show off Python in complete programs.

If you're looking for a Python tutorial, the official one at `https://docs.python.org/3/tutorial/` is excellent. We also recommend two online text books:

- *Think Python* by Allen B. Downey at `https://greenteapress.com/wp/think-python-2e/`
- *Automate the Boring Stuff with Python* by Al Sweigart at `https://automatetheboringstuff.com/`

D.1 Comments

```
# Single line comments start with a hash
'''
  Multiline strings can be written
  using three 's or "s.
'''
```

D.2 Variables

```
# A variable name is a.. zA..Z followed by a..zA..Z_0..9
```

```
age = 25        # integer; unlimited size
height = 1.65   # float; 64-bit

x, y = 25, 1e-9  # multiple assignment
x, y = y, x      # swap two values

isStudent = True  # (or False); a boolean
x = None    # None is a constant meaning 'undefined'
```

D.3 Operators

```
# Arithmetic: +, /, -, *
x = 5 / 2   # division; x = 2.5
x = 5 // 2  # integer division; x = 2
r = 5 % 2  # modulus; r = 1
s = 5 ** 2  # exponentiation; s = 25

x += 3  # same as x = x + 3
x -= 2  # same as x = x - 2

# Equality is ==
1 == 1  # => True
2 == 1  # => False
# Inequality
1 != 1  # => False
2 != 1  # => True

# Also: >, <, >=, <=

# Logical ops
g = True and False  # There's also: or
i = not True
```

D.4 Strings

```
"This is a string."
'This is also a string.'
name = "Alice"

"Hello " + "world!"  # => "Hello world!"
"Hello world!"[0]  # => 'H'

len("This is a string")  # => 16

# f-strings (f == formatting)
name = "Jane"
```

```
f"She said her name is {name}."
             # => "She said her name is Jane"
```

D.5 I/O

```
s = input("What is s? ")   # prompt for a string
x = int(input("x? "))   # read in an integer
x, y = map(float(input("x y? ").split()))
     # read in two floats separated by white space

name = "Alice"
print("Hello", name)   # Hello Alice and a newline
print("Hello", name, end = ' ')  # space instead of newline

age = 25
print("My name is", name, "age", age)
# My name is Alice age 25

print("Her age is " + str(age) + " years")
   # Her age is 25 years
print(f"My name is {name} and I am {age} years old.")
   # My name is Alice and I am 25 years old.

# f-strings for formatting numerical output
val = 12.3456789
print(f"val = {val:.3f}")  # prints 12.346
print(f"val = {val:9.3f}")  # prints ___12.346
```

D.6 Lists

```
fs = ["apple", "banana", "orange", "cherry"]
print(fs[0])  # apple
print(fs[1])  # banana
print(fs[-1]  # cherry; counts from right

fs[1] = "kiwi"
print(fs)     # ['apple', 'kiwi', 'cherry']

xs = [1]*10    # create a 10-element list, full of 1's

lst.append(item)   # add item to end of lst
lst.extend(lst2)   # add all of list lst2 to end
lst = lst + lst2   # concatenation

"banana" in fs    # True
```

```
fs.index("banana")  # 1
fs.index("durian")  # raises a ValueError exception

lst.insert(idx, val) # insert val at index
lst.remove(val)      # remove first item with value = val
x = lst.pop(idx)     # remove item at index and return its value

lst.sort()
xsSort = sorted(lst)    # create a sorted copy

lst.reverse()
revlst = reversed(lst)  # create a reversed copy

r = len(fs) # 4

lst = [1, 2, 3, 4]
min(lst)  # 1
max(lst)  # 4
sum(lst)  # 20
```

D.6.1 List Slices.

```
a = [0, 1, 2, 3, 4, 5]
a[1:]    # [1,2,3,4,5]
a[:5]    # [0,1,2,3,4]
a[:-2]   # [0,1,2,3]
a[1:3]   # [1,2]
a[1:-1]  # [1,2,3,4]

b=a[:]   # Shallow copy of a
```

D.6.2 2D Lists.

```
joeMarks = [97, 90, 92, 87]
jillMarks = [92, 79, 85, 88]
andrewMarks = [6, 13, 7, -1]
classMarks = [joeMarks, jillMarks, andrewMarks]
print(classMarks)

print(classMarks[2])     # row
# [6, 13, 7, -1]
print(classMarks[0][3])  # row column
# 87
```

D.6.3 List Comprehensions.

```
ns = [1, 2, 3, 4, 5]
squares = [n**2 for n in ns]
print(squares)  # [1, 4, 9, 16, 25]
```

```
[x for x in [3, 4, 5, 6, 7] if x > 5]  # => [6, 7]

xs = [x for x in [1,2,3,4,5,6,7,8,9] if x % 2 == 0]
# xs = [2,4,6,8]
```

D.7 Tuples

```
# Like lists but cannot be modified; they are immutable
tup = (1, 2, 3)
tup[0]      # => 1
tup[0] = 3  # Raises a TypeError

# You can use most of the list operations
len(tup)         # => 3
tup + (4, 5, 6)  # => (1, 2, 3, 4, 5, 6)
tup[:2]          # => (1, 2)
2 in tup         # => True
```

D.8 Dictionaries

```
# Dictionaries store key-value pairs
d =  {'Mary':'555-6789', 'Jenny':'867-5309',
      'John':'555-1234', 'Bob':'444-4321'}
print(d["John"])  # '555-1234'

d.keys()   # return ['Mary', 'Jenny', 'John', 'Bob']
d.values()
  # return ['555-6789', '867-5309', '555-1234', '444-4321']

for key in sorted(phoneNos.keys()):
  print(key, phoneNos[key])

v = d.pop('John')
# remove 'John':'555-1234', and v = '555-1234'
```

D.9 Sets

```
s1 = set {"keyl","key2"}
s2 = {1,9,3,0}
s3 = set()  # same as {}

nums1 = {1, 2, 3, 4}
nums2 = {2, 3, 5}
# set ops
```

```
nums1 & nums2    # intersection: {2, 3}
nums1 | nums2    # union: {1, 2, 3, 4, 5}
nums1 - nums2    # difference: {1, 4}
nums1 ^ nums2    # xor: {1, 4, 5}
2 in nums1     # True
10 in nums1    # False
```

D.10 If Statements

```
age = 5
# prints "age is smaller than 10"
if age > 10:
  print("age is totally bigger than 10.")
elif age < 10:
  print("age is smaller than 10.")
else:
  print("age is indeed 10.")

# Indentation is significant in Python!
# Convention is to use four spaces, not tabs.
# But we prefer to use two spaces.
```

D.11 Loops

D.11.1 For Loops.

```
for i in [0,1,2,3,4]): # iterate over lists
  print(i)

animals = ["dog", "cat", "mouse"]
for i, value in enumerate(animals):
    print(i, value)
```

D.11.2 Loops using range().

```
# range([start,] stop, [, step])

for i in range(5):   # [0,1,2,3,4]
  print(i)

for i in range(1,5):   # [1,2,3,4]
  print(i)

for i in range(0,10,2):   # [0, 2, 4, 6, 8]
  print(i)

for i in range(4,-1,-1):   # [4, 3, 2, 1, 0]
```

```
  print(i)

'''
  One of the most common beginner's mistakes
  is to forget that range() does not include
  the last element.
'''
```

D.11.3 while Loops.

```
x = 0
while x < 4:
  print(x)
  x += 1

while True:
  print("Hello", end = ' ')
```

D.11.4 Inside a Loop.

```
for i in [1, 2, 3, 4]:
  if i == 2:
    continue   # jump to start of next loop
  if i == 3:
    break      # end  loop
  print i      # prints => 1, 3
```

D.12 Functions

```
def adder(x, y, z=0):
  sum = x + y + z
  return sum

r = adder(1, 2)      # r = 3
r = adder(1, 2, 3)   # r = 6

# Return multiple values
def pows(x, y):
  return x**y, y**x

x = 2
y = 3
p1, p2 = pows(x, y)     # p1 = 8, p2 = 9
```

D.12.1 Function Names.

```
def add(x,y):
  return x + y
```

```
def subtract(x,y):
  return x - y

def calc(fn, x, y):     # function name argument
  return fn(x, y)       # call function

z = calc(add, 5, 10)      # z = 15
z = calc(subtract, 5, 10) # z = -5
```

D.12.2 Global Variables.

```
'''
  Global variables can be seen by every function,
  but can not be changed unless the function uses
  the global keyword.
'''

def tax(p, tax_rate):
  global taxCount
  total = p + (p * tax_rate)
  taxCount += 1
  return total

taxCount = 0
p = float(input("Enter a price: "))
p = tax(p, 0.06)
p = tax(p, 0.06)
p = tax(p, 0.06)
print(f"Price (after {taxCount} calls = {p}")
```

D.12.3 Lambda Functions.

```
f = lambda x: x**2
print(f(3))  # Output: 9

g = (lambda x, y: x**2 + y**2)
g(2, 1)  # => 5
```

D.12.4 map and filter.

```
def add10(x):
  return x + 10
list(map(add10, [1, 2, 3]))   # [11, 12, 13]

isOdd = lambda x: x%2 == 1
list(filter(isOdd, [3, 4, 5, 6, 7]))  # [3, 5, 7]
list(filter((lambda x: x%2 == 0), [3, 4, 5, 6, 7]))  # [4,6]
```

D.13 Exceptions

```
try:
  x = 1/0
except ZeroDivisionError:
  print("Cannot divide by zero")

try:
  database.update()
except Exception as e:
  print(e.msg)
  database.abort()
finally:  # always do this
  database.commit()
```

D.14 File I/O

```
# Read a file
with open("myfile.txt", "r") as f:
  contents = f.read()
print(contents)

# Write to a file
with open("myfile.txt", "w") as f:
  f.write("Hello, World!")
```

D.15 Modules

```
import math
print(math.sqrt(16))  # => 4.0

from math import ceil, floor
print(ceil(3.7))   # => 4
print(floor(3.7))  # => 3

# You can import all functions from a module
from math import *

# shorten module names
import math as m
math.sqrt(16) == m.sqrt(16)  # => True

'''
  Python modules are just ordinary Python files. You
  can write your own, and import them. The name of the
  module is the name of the file.
```

```
'''
from frange import *      # import everything from frange.py

'''
  Inside a module file, use __name__ check so test
  code is only run when the file is called directly
'''
if __name__ == '__main__':
  # test frange functions
```

D.16 Classes

```
class Person:
  def __init__ (self, name, age):
    self.name = name   # properties
    self.age = age

  def greet(self):  # methods
    print("Hello, my name is " + self.name)

alice = Person("Alice", 25)
alice.greet()    # Hello, my name is Alice
alice.age = 26
```

D.17 Three Examples

These examples can be found at `http://coe.psu.ac.th/~ad/explore/code/D_Python/`.

D.17.1 `caesar.py`. caesar.py uses brute force on the encrypted text "nzohfu gur rarzl ng avtug" to crack its Caesar cipher (`https://en.wikipedia.org/wiki/Caesar_cipher`). The code illustrates the use of for-loops, if-statements, and f-strings in print().

```
codedMsg = 'nzohfu gur rarzl ng avtug'

for i in range(26):
  guess = ''
  for ch in codedMsg:
    if ch != ' ':
      # Shift the letter forward by i
      if ord(ch) + i <= 122:
        ch = chr(ord(ch) + i)
      else:  # Subtract 26 if we go beyond z
        ch = chr(ord(ch)+i-26)
    guess = guess + ch
  print(f"{chr(i+65):2s}: {guess}")
```

Listing D.1. Implementing the Caesar cipher

The output:

```
> python caesar.py
A : nzohfu gur rarzl ng avtug
B : oapigv hvs sbsam oh bwuvh
       : # lines not shown
M : zlatrg sgd dmdlx zs mhfgs
N : ambush the enemy at night
O : bncvti uif fofnz bu ojhiu
       : # lines not shown
Y : lxmfds esp pypxj le ytrse
Z : mynget ftq qzqyk mf zustf
```

The 'N' line reveals the plain text message.

D.17.2 `sinApprox.py`. sinApprox.py calculates the sine of an angle using its Taylor series approximation to N terms, and prints the result along with the math.sin() answer. The program illustrates the use of input conversion, while-loops, and the math module.

The Taylor series for sine:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

```
import math
N = 25

degs = float(input("degrees? "))
x = math.radians(degs)

i = 1
sum = x
sign = 1
while i < N:
  sign = -sign
  i += 2
  term = sign*x**i/math.factorial(i)
  sum += term

print(f"     sin(x) = {sum} ({N} terms)")
print("math.sin(x) =", math.sin(x))
```

Listing D.2. Approximating a sine

Typical output:

```
> python sinApprox.py
degrees? 30
```

```
     sin(x) = 0.49999999999999994 (25 terms)
math.sin(x) = 0.49999999999999994

> python sinApprox.py
degrees? 45
     sin(x) = 0.7071067811865475 (25 terms)
math.sin(x) = 0.7071067811865476
```

D.17.3 `romans.py`. romans.py is a small module containing functions for converting integers into Roman numerals (`https://en.wikipedia.org/wiki/Roman_numerals`), and vice versa. The module contains a test-rig after a `__name__` test.

```
# 1000 <->"M", 900 <->"CM", 500 <->"D"...
decimalDens = [1000,900,500,400, 100, 90, 50, 40, 10, 9, 5,4,1]
romanDens   = ["M","CM","D","CD","C","XC","L","XL","X","IX","V","IV",
    "I"]

def num2Roman(dec):
  # sanity checks
  if dec <= 0:
    raise ValueError("It must be a positive")
  elif dec >= 4000:
    raise ValueError("It must be lower than MMMM(4000)")
  return toRoman(dec, "", decimalDens, romanDens)


def toRoman(num, s, decs, romans):
  if decs != []:
    if num < decs[0]:
      # deal with the rest of the denomination
      return toRoman(num, s, decs[1:], romans[1:])
    else:
      # reduce this denomation till num < desc[0]
      return toRoman(num-decs[0], s+romans[0], decs, romans)
  else:
    return s


def roman2Num(str):
  str = str.upper()
  res = 0
  i = 0
  while (i < len(str)):
    s1 = toDecimal(str[i])
    if (i+1 < len(str)):
      s2 = toDecimal(str[i+1])
      if (s1 >= s2):
        res += s1
        i += 1
      else:  # s1 < s2
```

```
        res += (s2 - s1)
        i += 2
    else:
      res += s1
      i += 1
  return res

def toDecimal(ch):
  # Roman symbol to decimal
  try:
    idx = romanDens.index(ch)
    return decimalDens[idx]
  except ValueError:
    print(ch, "not recognized; using 1")
    return 1



if __name__ == "__main__":
  rs = []
  for n in [12, 44, 57, 111, 1910, 2024]:
    r = num2Roman(n)
    rs.append(r)
    print(n, r)
  print("---------------")
  for rom in rs:
    print(rom, roman2Num(rom))
```

Listing D.3. Integers to/from Roman numerals

When the program is run from the command line, the test-rig is called:

```
> python romans.py
12 XII
44 XLIV
57 LVII
111 CXI
1910 MCMX
2024 MMXXIV
---------------
XII 12
XLIV 44
LVII 57
CXI 111
MCMX 1910
MMXXIV 2024
```

If the module is imported (in this case, to the Python REPL), then the test-rig is not executed, but the functions are available:

```
>>> from romans import *
>>> num2Roman(1999)
```

```
'MCMXCIX'
>>> roman2Num("MCMXCIX")
1999
```

Appendix E

Numbers in Python

This appendix provides an overview of numerical support in Python, mostly concentrating on the problems that arise from the implementation of reals as floating-point numbers. The code examples can be found at `http://coe.psu.ac.th/~ad/explore/code/E_Nums/`.

The python documentation includes an excellent section on this topic, "Floating Point Arithmetic: Issues and Limitations" (`https://docs.python.org/3/tutorial/floatingpoint.html`), and the standard academic reference is 'What Every Computer Scientist Should Know About Floating-Point Arithmetic' by David Goldberg [**Gol91**] (`https://people.cs.pitt.edu/~cho/cs1541/current/handouts/goldberg.pdf`).

E.1 The int and float Types

An integer is initially assigned a machine word (e.g. 64 or 32 bits), but if the number doesn't fit, or grows too big, then it's automatically promoted to a 'long' which can be as large as your RAM allows. For example:

```
>>> import math
>>> math.factorial(100)
933262154439441526816992388562667004907159682643816214685929638952175
999932299156089414639761565182862536979208272237582511852109168640000
0000000000000000000
```

Details on the integer type are available via the sys module:

```
>>> import sys
>>> sys.int_info
sys.int_info(bits_per_digit=30, sizeof_digit=4)
```

```
>>> sys.maxsize
9223372036854775807
```

sys.maxsize is the largest regular integer, before promotion kicks in.

Things aren't so rosy for Python floats. Almost all platforms map them to IEEE 754 binary64 doubles (`https://en.wikipedia.org/wiki/IEEE_754`). This supports very large and small numbers, but there are limits:

```
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15,
mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

We'll describe some of the consequences of these limitations in what follows.

E.2 The math Module

Table E.1 (located at the end of this appendix) lists the functions in the math module, along with comments about some of the more obscure ones. They are grouped into six categories: number-theoretic, trigonometric, angular conversion, hyperbolic, special functions (e.g. gamma()), and constants. Full information can be found in Python's documentation at `https://docs.python.org/3/library/math.html`.

E.3 Numerical Errors

It's useful to distinguish between two types of errors in numerical calculations:

- **Approximation errors**. For example, you can generate the exponential, e^x, using a Taylor series:

$$y = \sum_{n=0}^{n} \frac{x^n}{n!}$$

  but the result will always be approximate since the terms from $n + 1$ to ∞ aren't being evaluated.

- **Roundoff errors**. This kind of error appears every time a calculation is carried out using floating-point numbers. None of the IEEE 754 formats offers infinite precision, and so some information will inevitably be lost. For instance, we know that $\sqrt{2}^2 - 2 = 0$, but Python disagrees:

```
>>> (math.sqrt(2))**2 - 2
4.440892098500626e-16
```

E.4 Representing Reals

The general IEEE 754 standard represents a real number x in *normal* form using three values: s, f, and e, such that:

$$x = (-1)^s \times 1.f \times 2^{(e-bias)}$$

s is the sign bit, $1.f$ denotes that only the fractional part (i.e. the digits in f) is stored, without the leading 1 (which is sometimes called the hidden bit). e is stored, but the actual exponent is $e - bias$, where bias is a predetermined offset which means that e is always positive, and so can be represented by an unsigned integer.

In the IEEE double format (used by Python floats), a 64-bit word is divided up as follows: 1 bit for the sign s, 11 bits for the exponent e, and 52 bits for the fraction of the mantissa f (see Fig. E.1).

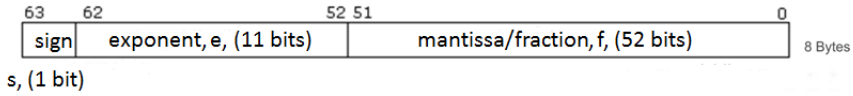


Figure E.1. The IEEE754 64-bit standard for floats

e is an 11-bit unsigned integer ranging from 0 to 2047, with the bias set to 1023 ($2^{10} - 1$) so the float's actual exponent can range between -1022 and +1023. (The exponents for -1023 (all 0s) and +1024 (all 1s) are reserved for special numbers.)

The fraction uses 52 bits: $m_{51}, m_{50}, , m_{49}, ..., m_1, m_0$, which are utilized in the float as:

$$1.f = 1 + m_{51} \times 2^{-1} + m_{50} \times 2^{-2} \cdots + m_0 \times 2^{-52}$$

For example, the binary version of 13256.625 is:

$$13256.625_{10} \equiv 11001111001000.101_2$$

Its normalized form is:

$$11001111001000.101_2 = 1.1001111001000101_2 \times 2^{13}.$$

The first digit of the normal form is always 1, so isn't stored in the fractional part of the machine word. Also, the stored exponent is offset by the bias, so $13 + 1023 = 1036_{10} = 10000001100_2$ is placed in the word.

floatUtils.py contains a printIEEE754() function that can perform these conversions on a supplied float:

```
>>> from floatUtils import *
>>> printIEEE754(13256.625)
IEEE754 of 13256.625 =
0 : 10000001100 : 1001111001000101000000000000000000000000000000000000000000
```

```
  Sign == +; Exponent == 13; mantissa == 1.6182403564453125
  Decimal value: 13256.6250000000000000
```

The final line of the output prints the binary string as a Python Decimal, which utilizes 28 digits of precision by default. This is useful for confirming whether the float really does have an exact binary value. If it does then the generated Decimal will be the same as the supplied float.

E.4.1 Largest Number. The largest possible fractional part (f) of the machine word is all 1's, so $1.f$ will be 1.111...111, which we can approximate as 2. The biggest possible stored exponent is 2046, which means that the largest true exponent is 2046 - 1023 = 1023. Therefore the largest number is approximately:

$$+1 \times 2 \times 2^{1023} = 2^{1024} \approx 1.8 \times 10^{308}$$

The biggest float is stored in sys.float_info.max, which can be examined with printIEEE754():

```
>>> printIEEE754(sys.float_info.max)
IEEE754 of 1.7976931348623157e+308 =
0 : 11111111110 : 1111111111111111111111111111111111111111111111111111
  Sign == +; Exponent == 1023; mantissa == 1.9999999999999998
  Decimal value: 1.797693134862315708145274237E+308
```

Since float has a maximum value, then it's possible for float calculations to overflow. The following code from floatTests.py shows how to catch an overflow as an exception:

```
v = 2
while True:
  print(f"exp({v}): ", end='')
  try:
    print(math.exp(v))
  except Exception as e:
    print(type(e).__name__, e.args)  # or just print(e)
    break
  v *= 2
```

The output:

```
exp(2): 7.38905609893065
exp(4): 54.598150033144236
  : # more lines, not shown
exp(256): 1.5114276650041035e+111
exp(512): 2.2844135865397565e+222
exp(1024): OverflowError ('math range error',)
```

E.4.2 Subnormals. When $e = 0$, IEEE 754 employs an alternative format to represent numbers less than the smallest normal number (2.23e-308). *Subnormals* are encoded as:

$$x = (-1)^s \times 0.f \times 2^{-1022}$$

The smallest possible fraction is 2^{-52} which when multiplied to the exponent (always 2^{-1022}) gives the smallest number, 2^{-1074}, which is $\approx 4.94 \times 10^{-324}$.

If we try to generate a number that's larger than 1.8×10^{308}, we get overflow. If we try to represent a number that's smaller than the subnormal 4.94×10^{-324}, then we get underflow. These two extremes are illustrated in Fig. E.2.

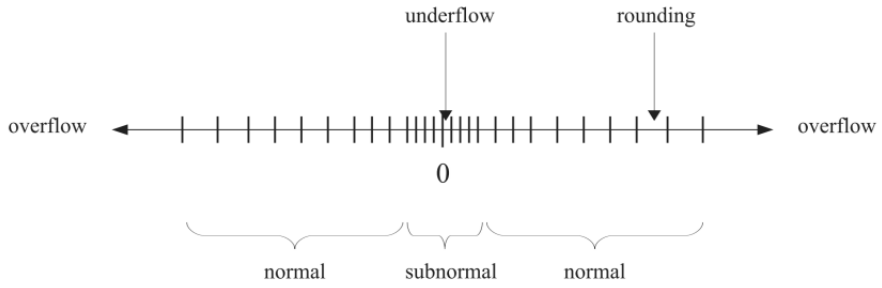


Figure E.2. Representing floating-point numbers

The following code snippet from floatTests.py starts with the smallest *normal* float, and keeps dividing it by 100 to show the switch to subnormals.

```
f = sys.float_info.min
print("float min: ", f)

while f != 0:
  f = f/100
  print("Divided by 100:", f)  # gradual underflow
```

The output:

```
float min:  2.2250738585072014e-308
Divided by 100: 2.2250738585072e-310
Divided by 100: 2.225073858507e-312
Divided by 100: 2.2250738583e-314
Divided by 100: 2.22507384e-316
Divided by 100: 2.225074e-318
Divided by 100: 2.2253e-320
Divided by 100: 2.2e-322
Divided by 100: 0.0
```

printIEEE754() is able to distinguish between the normal and subnormal formats when converting a float:

```
>>> printIEEE754(sys.float_info.min)
IEEE754 of 2.2250738585072014e-308 =
0 : 00000000001 : 0000000000000000000000000000000000000000000000000000
```

```
  Sign == +; Exponent == -1022; mantissa == 1.0
  Decimal value: 2.225073858507201383090232717E-308

>>> printIEEE754(2.225074e-318)
IEEE754 of 2.225074e-318 =
0 : 00000000000 : 0000000000000000000000000000000001101101111100111000
Subnormal:  Sign == +; Exponent == -1022; mantissa == 1.0000000827403710e-10
          Decimal value: 2.225074042610637936353595215E-318
```

E.4.3 Zero, Infinity, NaN. There are several special values in the standard, shown in Fig. E.3.

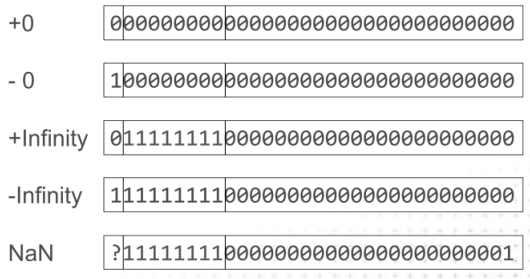


Figure E.3. Floating-point special values

Most arithmetic operations that produce 0 are given the value +0, with -0 reserved for very tiny negative numbers that are rounded to 0 due to precision limits.

NaN (Not a Number) is the result of indeterminate operations such as 0/0, $\infty - \infty$, or x + NaN.

E.4.4 Number Precision. Fig. E.2 also depicts the unequal effects of precision on number spacing. Even when the exponent is large, there is still only 52 bits to assign to the fraction, and so the distance between exactly representable number will be relatively large. But when the exponent is smaller, the fraction is able to denote more closely packed numbers.

This can be observed by using math.nextafter(x,y), introduced in Python 3.9, which returns the next float value after x when moving towards y. This can be employed alongside math.ulp(x) which returns the value of the least significant bit of x. ULP stands for 'unit of least precision', or 'unit in the last place'.

The following code reports next float values and ULPs for values between 1e-5 and 1e10:

```
for i in range(-5, 10):
  nxt = math.nextafter(10**i, float('inf'))
```

```
  print(f"next after {10**i} = {nxt}")
  print(" ulp =", math.ulp(nxt))
```

The output:

```
next after 1e-05 = 1.0000000000000003e-05
 ulp = 1.6940658945086007e-21
next after 0.0001 = 0.00010000000000000002
 ulp = 1.3552527156068805e-20
next after 0.001 = 0.0010000000000000002
 ulp = 2.168404344971009e-19
    :  # more lines; not shown
next after 10000000 = 10000000.000000002
 ulp = 1.862645149230957e-09
next after 100000000 = 100000000.00000001
 ulp = 1.4901161193847656e-08
next after 1000000000 = 1000000000.0000001
 ulp = 1.1920928955078125e-07
```

The general point is that the precision gap (the ULP) between representable numbers gets wider as larger numbers are considered. This is due to the number of bits in the fractional part of the number remaining constant.

Assume that a number x is stored as $\pm a \times 2^b$, where $\frac{1}{2} \leq a < 1$. Since the fractional part of a occupies 52 bits, there's a possible error in a in the interval $\pm\frac{1}{2} \times 2^{-52} = \pm 2^{-53}$. Thus the stored value is actually $a + E$, where $|E| \leq 2^{-53}$. It follows that the value of x is:

$$\pm(a + E) \times 2^b = \pm a \times 2^b \cdot (1 \pm \frac{E}{a}) = x(1 + \epsilon),$$

where

$$\epsilon = \frac{E}{a}, \quad \text{and} \quad |\epsilon| \leq \frac{1}{a} \times 2^{-53} \leq 2^{-52},$$

So a number x is actually stored as $x(1 + \epsilon)$, where $|\epsilon| \leq p = 2^{-52}$. p is the precision, which in terms of decimal digits, is approximately:

```
>>> import math
>>> math.floor( math.log10(2**52) )
15
```

Only the first 15 or 16 digits of a float can be accurate in the best case. Unfortunately, values may become a lot less accurate when they result from math operations involving other numbers.

Suppose we have x_1 and x_2, which are stored as $x_1(1 + \epsilon_1)$ and $x_2(1 + \epsilon_2)$, where $|\epsilon_1| \leq p$ and $|\epsilon_2| \leq p$ (recall that $p = 2^{-52}$). We multiply these numbers together to obtain $y = x_1 x_2$. The actual value for y will be

$$\begin{aligned} x_1(1 + \epsilon_1)x_2(1 + \epsilon_2) &= x_1 x_2(1 + \epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2) \\ &\simeq y(1 + \epsilon_1 + \epsilon_2) \end{aligned}$$

since $\epsilon_1\epsilon_2$ is small compared with the other terms involved.

Thus y is $y(1+\delta)$, where $\delta \simeq \epsilon_1 + \epsilon_2$, and so $|\delta| \leq 2p$. In other words, the possible error for y may be double that of x_1 and x_2. However, this is an upper limit, and in many cases it may be much smaller. For instance if ϵ_1 and ϵ_2 have opposite signs then they will partially cancel out one another.

If instead we consider $y = x_1/x_2$, the result is:

$$\frac{x_1(1+\epsilon_1)}{x_2(1+\epsilon_2)} = \frac{x_1}{x_2}(1+\epsilon_1)(1-\epsilon_2+\epsilon_2^2-\epsilon_2^3+\cdots),$$

using the binomial expansion of $(1+\epsilon_2)^{-1}$. Ignoring terms involving products of two or more ϵ's, this gives

$$y(1+\epsilon_1-\epsilon_2).$$

Thus y is $y(1+\delta)$, where $\delta \simeq \epsilon_1 - \epsilon_2$. However, since ϵ_1 and ϵ_2 can be either positive or negative, we again have the worst case that $\delta \leq 2p$.

Addition and subtraction are a little more difficult to analyze. If $y = x_1 + x_2$ then y is

$$x_1(1+\epsilon_1) + x_2(1+\epsilon_2) = x_1 + x_2 + x_1\epsilon_1 + x_2\epsilon_2 = y(1+\delta),$$

with

$$\delta = \frac{x_1\epsilon_1 + x_2\epsilon_2}{x_1 + x_2}. \tag{E.1}$$

We know that $-p \leq \epsilon_1 \leq p$ and $-p \leq \epsilon_2 \leq p$; so if x_1 and x_2 are both positive, we can write

$$-\frac{x_1 p + x_2 p}{x_1 + x_2} \leq \delta \leq \frac{x_1 p + x_2 p}{x_1 + x_2}$$

or $|\delta| \leq p$. Since the limit for $|\delta|$ is the same as the limit for $|\epsilon|$ there is no additional loss of accuracy when two positive numbers are added together.

We can extend this argument to the case when x_1 and x_2 are both negative, but *not* to the situation when x_1 and x_2 are opposite signs, i.e. when we are doing subtraction. If x_2 is close to $-x_1$, the denominator $x_1 + x_2$ in Equ. (E.1) is nearly zero, so δ could be *very* large. It follows that we may suffer a substantial loss of accuracy when subtracting two nearly equal numbers. This situation is probably the most common precision problem in computational work.

E.4.5 Floating-point Arithmetic. The preceding discussion doesn't include errors caused by how floating-point arithmetic is implemented. IEEE 754 addition and subtraction require that the numbers have the same exponent size, which is achieved by shifting the bits of the fractional part of one of the numbers to the right.

We'll illustrate this process in a simple way by adding the decimals 123456.7 and 0.009876543 (without converting them to binary), while assuming that a number can only hold 7 significant digits. First, the numbers are converted to exponent form as 1.234567×10^5 and 9.876543×10^{-3}. The smaller second number

is shifted right by 8 places, so its exponent matches that of the first number, resulting in $0.0000000987643 \times 10^5$. The numbers are added, producing $1.23456709876543 \times 10^5$. But, since only seven digits can be stored, the result is rounded to 1.234567×10^5, which causes the contribution of the second number to be lost completely.

The loss of digits may be worse when two similarly-sized numbers are subtracted. As an example, consider 123456.787654 - 123456.712345. Once again let's assume that only seven significant digits can be stored, so the subtraction will involve 1.234568×10^5 (note the rounding) and 1.234567×10^5. The result is 0.000001×10^5 which is normalized to 1×10^{-1}, or just 0.1. Of course, the actual result should be 0.075309. Substituting these values into Equ. (E.1) gives us a δ of $0.1/0.075309 \simeq 1.328$, or 133%. Essentially, the computed answer contains no correct digits.

The problems become harder to detect when we remember that 'simple' floating-point numbers, such as 0.1, 0.2, and 0.3, may have rounding issues. Consider:

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 0.3
False
```

There are two problems here: the rounding of the numbers when stored, and the loss of accuracy when they are added.

Calls to printIEEE754() reveal the details:

```
IEEE754 of 0.1 =
0 : 01111111011 : 1001100110011001100110011001100110011001100110011001100110011010
  Sign == +; Exponent == -4; mantissa == 1.6
  Decimal value: 0.1000000000000000055511151231

IEEE754 of 0.2 =
0 : 01111111100 : 1001100110011001100110011001100110011001100110011001100110011010
  Sign == +; Exponent == -3; mantissa == 1.6
  Decimal value: 0.2000000000000000111022302462

IEEE754 of 0.3 =
0 : 01111111101 : 0011001100110011001100110011001100110011001100110011001100110011
  Sign == +; Exponent == -2; mantissa == 1.2
  Decimal value: 0.29999999999999998888977697538
```

None of the numbers can be represented exactly in binary, so lose some accuracy through rounding. This is usually not apparent when the floats are printed since only at most 15 digits are reported:

```
>>> 0.1
0.1
>>> 0.2
0.2
```

```
>>> 0.3
0.3
```

However, the floating-point binary versions of 0.1 and 0.2 (which we'll call a and b) are:

$$a = 1.1001100110011001100110011001100110011001100110011010 \times 10^{-4}$$
$$b = 1.1001100110011001100110011001100110011001100110011010 \times 10^{-3}$$

Before adding them, the mantissa of the smaller number a (0.1) is shifted 1 bit to the right so that both number's exponents are the same:

$$a = 0.1100110011001100110011001100110011001100110011001101 \times 10^{-3}$$

$a + b$ produces c (the value approximating 0.3):

$$c = 10.0110011001100110011001100110011001100110011001100111 \times 10^{-3}$$

This is normalized by shifting it 1 bit to the right:

$$c = 1.0011001100110011001100110011001100110011001100111 \times 10^{-2}$$

Internally, the leading 1 isn't stored, but more importantly the fractional part now has 53 bits, so the result must be rounded up to:

$$c = 1.0011001100110011001100110011001100110011001100111\underline{000} \times 10^{-2}$$

The changed digits have been underlined. The last bit is now discarded, leaving the fraction as:

```
0011001100110011001100110011001100110011001100110100
```

This is *not* the same as the printIEEE754(0.3) output shown above, but is the floating-point result of 0.1 + 0.2:

```
>>> printIEEE754(0.1+0.2)
IEEE754 of 0.30000000000000004 =
0 : 01111111101 : 0011001100110011001100110011001100110011001100110100
  Sign == +; Exponent == -2; mantissa == 1.2000000000000002
  Decimal value: 0.3000000000000000444089209850
```

One way of avoiding this problem is by comparing floats using less decimal places (i.e. less than 15 significant digits). For instance:

```
>>> round(0.1 + 0.2, 5) == round(0.3, 5)
True
```

E.4.6 The Dangers of Equality. One common place where floats cause problems is within tests for equality, often inside if-branches and while loops. For instance:

```
print("\nCount down to 0:")
a = 2
while a != 0:
  print(a, end= ' ')
  a -= 0.1
  if a < -2:
    print("\nReached -2!")
    break
```

The lack of explicit typing in Python contributes to the confusion since a starts as an integer but is cast to a float when it's incremented by 0.1. As we saw above, 0.1 is not represented exactly in binary, and the small error will gradually increase as the loop carries out more additions, making it very unlikely that a will land exactly on 0. This is shown in the output:

```
Count down to 0:
2 1.9 1.7999999999999998 1.6999999999999997 1.5999999999999996
1.4999999999999996 1.3999999999999995 1.2999999999999994 ...
0.09999999999999937 -6.38378239159465e-16 -0.10000000000000064
-0.20000000000000065 -0.30000000000000066 -0.4000000000000007
... -1.500000000000001 -1.6000000000000012 1.7000000000000013
-1.8000000000000014 -1.9000000000000015
Reached -2!
```

E.4.7 Ordering Matters. The result of operations involving floating-point numbers may depend on the order in which those operations are carried out. An example:

```
>>> (0.7 + 0.1) + 0.3
1.0999999999999999
>>> 0.7 + (0.1 + 0.3)
1.1
```

A more dramatic difference:

```
>>> xt = 1e20
>>> yt = -1e20
>>> zt = 1
>>> (xt + yt) + zt
1.0
>>> xt + (yt + zt)
0.0
```

This second example is really performing *subtraction* of similar numbers, and so it's to be expected that the rounding error may increase significantly. In the

first bracketed calculation, the two large numbers, xt and yt, cancel each other and zt becomes the answer. In the second version, adding 1 to the negative large number yt rounds to yt. Then, xt and yt cancel each other out.

One fix for summing problems is to use math.fsum(), which sums a list while avoiding loss of precision by tracking the intermediate partial sums. The following contrasts the results of sum() and fsum():

```
>>> xt = 1e20
>>> yt = -1e20
>>> zt = 1
>>> sum([xt, yt, zt])
1.0
>>> sum([yt, zt, xt])
0.0
>>> import math
>>> math.fsum([xt, yt, zt])
1.0
>>> math.fsum([yt, zt, xt])
1.0
```

E.4.8 Rearranging Expressions. Two versions of the same expression:

$$\text{poor}(x) = \frac{1}{\sqrt{x^2+1}-x} \quad \text{and} \quad \text{good}(x) = \sqrt{x^2+1}+x$$

encoded as two functions in exprEval.py:

```
def poor_eval(x):
    return 1/(math.sqrt(x**2 + 1) - x)

def good_eval(x):
    return (math.sqrt(x**2 + 1) + x)


for v in [1000, 10000, 1000000, 10000000]:
  print(f"{v:8d} poor: {poor_eval(v):.8f}; good: {good_eval(v):.8f}")
```

Listing E.1. Evaluating two versions of the same expression

The loop is never completed since 100000000 evaluates to a zero denominator in poor_eval():

```
> python exprEval.py
    1000 poor: 2000.00049981; good: 2000.00050000
   10000 poor: 19999.99977765; good: 20000.00005000
 1000000 poor: 1999984.77112922; good: 2000000.00000050
10000000 poor: 19884107.85185185; good: 20000000.00000005
Traceback:
  File "exprEval.py", line 15, in <module>
```

```
  File "exprEval.py", line 8, in poor_eval
ZeroDivisionError: float division by zero
```

The earlier results also point to an obvious blooming of errors in poor_eval() compared to the more accurate good_eval(). The reason lies in poor_eval()'s use of subtraction. For large values of x, two almost identical values are being subtracted, producing a number close to 0 with a large relative error.

E.4.9 Computing the Exponential Function. A direct implementation of the exponential function using the Taylor/Maclaurin series will sum $x^n/n!$ for each term:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

This suffers from (at least) two problems: (a) calculating the power and the factorial is costly, and (b) both x^n and $n!$ can become very large (potentially overflowing) even though the required term ratio is small.

Instead, we take advantage of the fact that the n-th term in the expansion is related to the $(n-1)$-th term:

$$\frac{x^n}{n!} = \frac{x}{n}\frac{x^{n-1}}{(n-1)!}$$

There are two main ways to terminate a summation: (a) test for when a new term is "small enough", or (b) test for when the running total reaches a particular value. We'll employ (a) and terminate the loop when adding the n-th term to the total doesn't change it (due to the precision limits of floats). calcExp() is shown in Listing E.2 (compexp.py).

```
def calcExp(x):
  n = 0
  oldsum, tot, term = 0, 1, 1
  while tot != oldsum:
    oldsum = tot
    n += 1
    term *= x/n
    tot += term
    # tot += (x**n)/math.factorial(n)
  print(n, "iterations")
  return tot
```

Listing E.2. Calculating exp()

It's instructive to compare calcExp() and math.exp() for different x values, starting with $x = 2$.

```
>>> import math
>>> from compexp import *
>>> calcExp(2)
23 iterations
```

```
7.389056098930649
>>> math.exp(2)
7.38905609893065
```

There's agreement to about 15 decimal digits, which is to be expected when dealing with Python floats.

For x = 20, more iterations are needed, but the answer is again in agreement with math.exp() to 15 decimal digits.

```
>>> calcExp(20)
68 iterations
485165195.40979046
>>> math.exp(20)
485165195.4097903
```

The output for $x = -20$ tells a different story:

```
>>> calcExp(-20)
95 iterations
6.147561828914626e-09
>>> math.exp(-20)
2.061153622438558e-09
```

Even more iterations are required, but the real problem is that the function produces a sum that is totally wrong (although of the right order of magnitude). This is due to the way that the terms in the series have alternating signs, and so cancellation occurs between numbers of comparable magnitudes:

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots$$

Fortunately, there's an easy fix for this particular problem: take advantage of $e^{-x} = 1/e^x$:

```
>>> math.exp(-20)
2.061153622438558e-09
>>> x = calcExp(20)
68 iterations
>>> 1/x
2.061153622438557e-09
```

The Taylor expansion for positive x doesn't suffer from cancellations, and supplies 15 correct digits. Dividing 1 by this large number leads to an answer also with 15 correct digits.

The main part of compexp.py (Listing E.3; compexp.py) graphs the relative differences in the outputs between calcExp(x) and math.exp(x) for x values between 0 and 30:

```
xs = list(frange(0, 30, 0.1))
ys = []
```

```
for x in xs:
  ys.append( (calcExp(x) - math.exp(x))/math.exp(x) )
  # ys.append( (calcExp(D(x)) - D(x).exp())/D(x).exp() )

plt.xlabel("x")
plt.ylabel("error")
plt.plot(xs, ys)

plt.title("Relative Diff Errors for exp()s")
plt.show()
```

Listing E.3. Graphing math.exp() and calcExp()

Fig. E.4 confirms that the relative error stays in the range $\pm 1e-15$, the precision of floats.

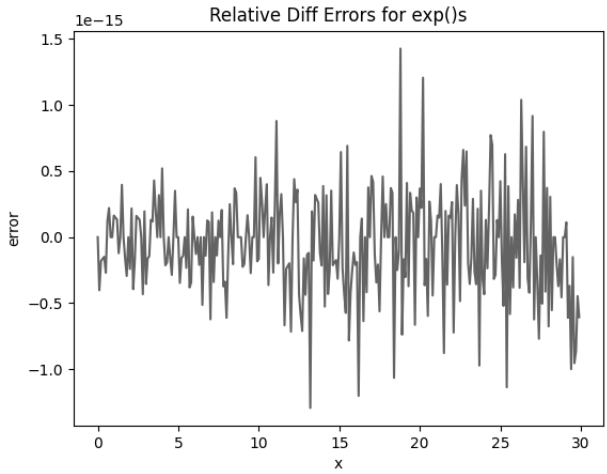


Figure E.4. Relative differences between math.exp() and calcExp()

E.5 Avoiding Float Problems

The simplest way to reduce the errors caused by the 'limited' precision of floats (i.e. ±1e-15) is to switch to Python's Decimal or Fraction types. Another solution is to employ a third-party library, such as mpmath.

E.5.1 Decimal. The Decimal module (`https://docs.python.org/3/library/decimal.html`) allows decimals to be represented exactly so that, for example, 0.1 + 0.1 + 0.1 == 0.3 is true. Unlike float, Decimal has no maximum, although RAM space and running time will constrain larger calculations. Also

a Decimal's precision (which defaults to 28 places) can be adjusted, and the programmer has many more rounding options.

The module is based on the Decimal Arithmetic Specification developed by IBM (`https://speleotrove.com/decimal/decarith.html`).

It's best if you initialize Decimals using strings:

```
import decimal
from decimal import Decimal as D

x = D('0.1')
print("Decimal str 0.1:", x)
y = D('0.3')
res = x + x + x
print("Decimal str 0.1 + 0.1 + 0.1 == 0.3? ", (res == y), res)

x = D(0.1)
print("\nDecimal float 0.1:", x)
y = D(0.3)
res = x + x + x
print("Decimal float 0.1 + 0.1 + 0.1 == 0.3? ", (res == y), res)
```

The first block of code manipulates its decimals exactly, but the second part has the same problems as float, albeit with greater precision (28 places for Decimal vs. 15 for float).

```
Decimal str 0.1: 0.1
Decimal str 0.1 + 0.1 + 0.1 == 0.3?  True 0.3

Decimal float 0.1:
0.1000000000000000055511151231257827021181583404541015625
Decimal float 0.1 + 0.1 + 0.1 == 0.3?  False
0.30000000000000000166533453694
```

A useful feature for code portability is that integers are cast to decimals when they're used with Decimals. This means that many functions can be reused unchanged for both float and decimal calculations. Consider calcExp() (Listing E.2): no changes are needed to that function except for calling it with a Decimal argument.

```
>>> import math
>>> import decimal
>>> from decimal import Decimal as D
>>> from compexp import *

>>> calcExp(2)      # float call
23 iterations
7.389056098930649
>>> calcExp(D(2))   # Decimal call
```

```
34 iterations
Decimal('7.389056098930650227230427460')

>>> calcExp(20)     # float call
68 iterations
485165195.40979046
>>> calcExp(D(20))  # Decimal call
88 iterations
Decimal('485165195.4097902779691068300')
```

Since the Decimal precision is larger, more iterations are required before the loop inside calcExp() terminates.

The increased precision for Decimal *may* mean that the rounding errors will be small enough not to swamp the calculation, as in the case of exp(-20):

```
>>> math.exp(-20)
2.061153622438558e-09
>>> calcExp(-20)
95 iterations
6.147561828914626e-09   # wrong
>>> calcExp(D(-20))
112 iterations
Decimal('2.061153622431177804227533375E-9')   # correct-ish
```

Decimal precision can be changed:

```
>>> decimal.getcontext().prec = 100
>>> calcExp(D(-20))
194 iterations
Decimal('2.061153622438557827965940380155820976375807275599103692972244661629164023784559353283074184159606035E-9')   # even more correct-ish
```

Wolfram Alpha (`https://www.wolframalpha.com`) reports e^{-20} to be

```
2.061153622438557827965940380155820976375807275599103692972244661629164023784559353283074184159606035E-9
```

which suggests that calcExp() is correct to around 85 decimal digits when Decimal precision is set to 100.

decimal.quantize() rounds a number to a fixed exponent. This method is useful for monetary applications, especially when rounding flags are utilized as well.

```
d = D("10.005")
print("Value:", d)  # 10.005
print("2 dps rounded:", d.quantize(D("1.00")))   # 10.00
print("2 dps rounded up:", d.quantize(D("1.00"),
                      decimal.ROUND_HALF_UP))   # 10.01
```

`decimal.ROUND_HALF_UP` ensures that a number is rounded up if its neighboring digits are equidistant, a criteria used in most business math.

A drawback of the Decimal module is a lack of math operations, although it does have logarithm, exponent, and square root functions. However, the 'Recipes' section of the documentation (`https://docs.python.org/3/library/decimal.html#recipes`) has Taylor series approximations for calculating π, sin(), and cos(), which we've copied to a local file, decTrig.py. We've also added a function for calculating arctan(), which has proved useful (e.g. see Sec. 6.7.8).

E.5.2 Fraction. For number problems involving rationals, it may be simpler to encode them as Fractions in Python:

```
>>> from fractions import Fraction as F
>>> F(3,10)
Fraction(3, 10)
>>> F(3,7) - F(5,9)
Fraction(-8, 63)
>>> F(3,10) * F(10,3) - 1
Fraction(0, 1)
>>> F(3,10) * F(10,3) - 1 == 0
True
```

These examples show that $\frac{3}{10}$ is represented exactly (recall that float has problems with 0.3), and calculations such as $\frac{3}{10} \times \frac{10}{3}$ are exact. It's also possible to combine fractions and integers.

E.5.3 Other Numeric Modules in Python. Python's math module is built atop the `numbers` class (`https://docs.python.org/3/library/numeric.html`), which also underpins classes for complex numbers (`cmath`), pseudo-random numbers (`random`) and statistics (`statistics`).

E.5.4 Third-party Libraries. **Mpmath** (`https://mpmath.org/`) supports arbitrary-precision similar to Python's Decimal, but unlike Decimal also implements functions across a wide range of math topics, including numerical integration and differentiation, limits and summations of infinite series, root-finding, Chebyshev approximations, fourier and taylor series, ODEs, and linear algebra.

Numpy (`https://numpy.org/`) supports large, multi-dimensional arrays and matrices, and a large assortment of routines for fast operations on those arrays, including sorting, selecting, I/O, shape manipulation, discrete Fourier transforms, linear algebra, and statistical operations.

As we mentioned in the preface, we've (mostly) avoided using numpy in this book since Python lists are almost as expressive in the 1D and 2D cases. Also, numpy programming with arrays employs element-by-element operations, which makes it difficult (in an untyped language) to know what an operation like $a * b$ means. The APL-style of programming that this encourages is very elegant, but also quite confusing for a beginner, especially when the concept of *nested*

arrays is utilized. Its support in MATLAB and the Wolfram Language (the programming language of Mathematica) is one reason why the second author think these tools are too complex for beginners. Nevertheless, numpy is the clear winner for large data, with some measurements showing its arrays to be 50x faster than lists.

The numpy `ndarray` have become the de-facto tool for multi-dimensional data in Python, and so numpy is employed by many other libraries.

Sympy (`https://www.sympy.org/`) focuses on symbolic mathematics, including algebraic manipulation, calculus, and equation solving, using symbolic rather than numerical techniques.

Scipy (`https://scipy.org/`) aims to offer support for all types of scientific and engineering applications. It's built on top of numpy, so includes routines for manipulating arrays, matrices, and other kinds of multidimensional data, and adds a large number of algorithms, such as: Airy functions, elliptic functions and integrals, a variety of statistical functions and distributions, Fresnel integrals, Legendre, hypergeometric, and spheroidal functions.

Panda (`https://pandas.pydata.org/`) provides tools for analyzing tabular data, including functions for importing and cleaning, statistical analysis, and visualization. Its name derives from the term "panel data", otherwise known as longitudinal data.

Exercises

(1) Fermat's last theorem states that no three positive integers x, y and z can satisfy the equation $x^n + y^n - z^n = 0$ for any integer $n > 2$. Explain this apparent counter-example:

```
>>> 844487.0**5 + 1288439.0**5 - 1318202.0**5
0.0
```

Another version of this problem, proposed by noted mathematician Homer Simpson is $3987^{12} + 3987^{12} = 3987^{12}$ (`https://boingboing.net/2014/10/17/homers-last-theorem.html`). Sadly, even when this is converted to powers of floats, it still (correctly) reports false. Why the difference from the first equation?

(2) The functions $f(x) = (1 - \cos^2 x)/x^2$ and $g(x) = \sin^2 x/x^2$ are mathematically indistinguishable, but plotted in the region $-0.001 \le x \le 0.001$ show a marked difference. Explain why.

(3) A direct implementation of Heron's formula for the area of a triangle,

$$A = \sqrt{s(s-a)(s-b)(s-c)} \quad \text{where } s = \frac{1}{2}(a+b+c),$$

is inaccurate if one side is very much smaller than the other two (i.e. resulting in "needle-shaped" triangles). Why? Demonstrate that the following reformulation gives a more accurate result by considering a triangle with sides $(10^{-13}, 1, 1)$, which has area 5×10^{-14}:

$$A = \frac{1}{4}\sqrt{(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))},$$

The sides are labeled so that $a \geq b \geq c$.

Functions	Comments
Number-theoretic	
exp(x)	
exp2(x)	2^x
log(x, base), log2(x), log10(x)	
log1p(x)	Natural log of $1 + x$
pow(x, y), sqrt(x), isqrt(n)	
cbrt(x)	$x^{\frac{1}{3}}$
ceil(x), floor(x), trunc(x), fabs(x), copysign(x, y)	
fmod(x, y), modf(x)	
factorial(n)	
gcd(integers), lcm(integers)	
comb(n, k)	Returns the number of ways to choose k items from n items without repetition and **without** order.
perm(n, k)	Returns the number of ways to choose k items from n items without repetition and **with** order.
prod(iterable), sumprod(p, q)	
fsum(iterable)	Avoids loss of precision by tracking multiple intermediate partial sums.
isclose(a, b), isfinite(x), isinf(x), isnan(x)	
frexp(x)	Returns the mantissa and exponent of the internal representation of a float.
ldexp(x, i)	Returns $x \times (2^i)$. The inverse of frexp().
nextafter(x, y, steps), ulp(x)	
remainder(x, y)	Returns IEEE 754-style remainder of x/y.
Trigonometric	
sin(x), cos(x), tan(x)	
asin(x), acos(x), atan(x), atan2(y, x)	
Angular Conversion	
degrees(x), radians(x)	
Hyperbolic	
sinh(x), cosh(x), tanh(x)	
asinh(x), acosh(x), atanh(x)	
Special Functions	
erf(x)	
erfc(x)	Complementary error function at x.
gamma(x)	
lgamma(x)	Natural log of Gamma function at x.
Constants	
pi, e, tau	
inf, nan	

Table E.1. Math module functions

Appendix F

Matplotlib

John D. Hunter, a neurobiologist, began developing matplotlib (`https://matplotlib.org/`) in 2003, originally to emulate Mathworks' MATLAB. Since then it has grown into an amazing plotting library, backed by a large community of developers.

One issue when learning matplotlib is that there are two main ways to write code (and matplotlib's documentation and examples use both):

(1) Explicitly initialize variables representing the chart figure and axes, and call functions on them:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
xs = [2, 4, 6, 8]
ys = [3, 5, 7, 9]
ax.plot(xs, ys)
ax.set_title('Simple plot')
plt.show()
```

Listing F.1. Plot a line using an explicit figure and axis

There are several ways to create these variables:

```
fig = plt.figure()
ax = fig.subplots()
```

or

```
ax = plt.gca()   # gc means "get current"
fig = plt.gcf()
```

(2) Create a chart using the functions in the `pyplot` submodule (`https://matplotlib.org/stable/tutorials/pyplot.html`):

```
import matplotlib.pyplot as plt

xs = [2, 4, 6, 8]
ys = [3, 5, 7, 9]
plt.plot(xs, ys)
plt.title('Simple plot')
plt.show()
```

Listing F.2. Plot a line using pyplot

We've used this second style in this text, except in a few cases.

Needless to say, Listing F.1 (simpleFigAx.py) and Listing F.2 (simplePyPlot.py) both produce the same chart (Fig. F.1).

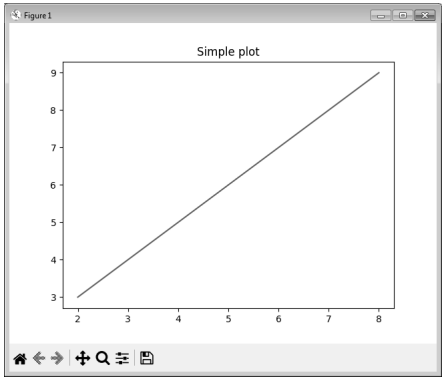


Figure F.1. A line chart

It's worth noting the buttons at the bottom of every matplotlib window. They offer useful features like panning, zooming, and saving of the graph.

There's an older coding style (now deprecated) based around the `pylab` interface, which can still be found in some textbooks and online:

```
from pylab import *
xs = [2, 4, 6, 8]
ys = [3, 5, 7, 9]
plot(xs, ys)
ptitle('Simple plot')
show()
```

Programs of this type can usually be easily edited to use `pyplot` functions instead.

Matplotlib is built on top of numpy, and so a great number of examples (including many in matplotlib's own documentation) involve the plotting of data stored in numpy arrays:

```
import numpy as np
x = np.array([1, 2, 3, 4])
y = x*2
plt.plot(x,y)
```

We prefer to stick with Python lists where possible, although numpy becomes inescapable when dealing with 3D graphs, as we'll see later.

The matplotlib online documentation is excellent. Good starting points include:

- The quick start guide: `https://matplotlib.org/stable/users/explain/quick_start.html`
- The user guide: `https://matplotlib.org/stable/users/`
- The tutorials page: `https://matplotlib.org/stable/tutorials/`
- A collection of printable 'cheatsheets': `https://matplotlib.org/cheatsheets/`
- A page listing the main plot types (`https://matplotlib.org/stable/plot_types/`) in five categories: pairwise data (i.e. graphing (x,y) coordinates), statistical distributions, gridded data, irregularly gridded data, and 3D and volumetric data.
- An official gallery of examples: `https://matplotlib.org/stable/gallery/`
- An unofficial gallery of examples: `https://python-graph-gallery.com/best-python-chart-examples/`

F.1 The Big Three

The 'big three' are line plots, scatter plots, and bar charts.

F.1.1 The Line Plot.

Listing F.2 (simplePyPlot.py) above shows how to draw a line using lists of coordinates. More commonly these are generated using an equation, as in Listing F.3 (parabola.py):

```
import matplotlib.pyplot as plt
from frange import *

xs = linspace(-2, 2, 100)
ys = [x**2 for x in xs]
plt.plot(xs, ys, label="y=x^2")
plt.title("Parabola")
plt.xlabel("x")
plt.ylabel("y")
```

```
plt.legend()
plt.show()
```

Listing F.3. Plot a parabola

We've used our linspace() function imported from frange.py, but you can utilize numpy's linspace() instead if you wish. It generates the requested number of floats equally spaced out across a specified range. For example:

```
>>> from frange import *
>>> linspace(-2, 2, 10)
[-2.0, -1.5555555555555556, -1.1111111111111112,
-0.6666666666666667, -0.2222222222222232,
0.22222222222222232, 0.6666666666666665,
1.1111111111111107, 1.5555555555555554, 2.0]
```

The plot (Fig. F.2) includes a label, drawn on the graph using plt.legend(), and the x- and y- axes are labeled.

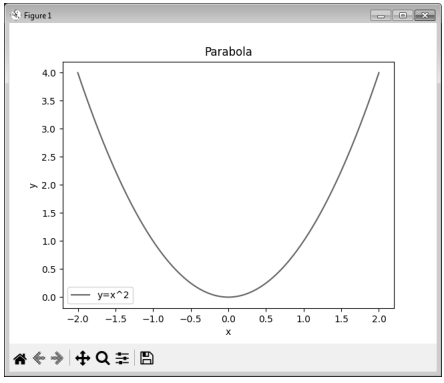


Figure F.2. A line plot of a parabola

F.1.2 The Scatter Plot. Scatter plots also utilize lists of coordinates, but plotted with scatter().

```
import random
import matplotlib.pyplot as plt

xs = [ random.gauss(0, 1) for _ in range(1000)]
ys = [ random.gauss(0, 1) for _ in range(1000)]
plt.scatter(xs, ys, s=5)  # s is point size
plt.show()
```

Listing F.4. A Gaussian scattering

The graph produced by Listing F.4 (randScatter.py) is shown in Fig. F.3 (from now on we'll crop the matplotlib buttons to reduce the image size).

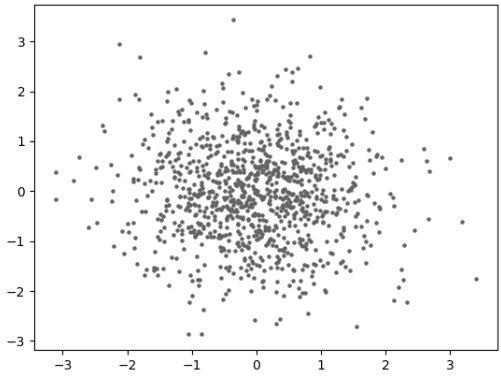


Figure F.3. A scatter plot of Gaussian numbers

F.1.3 The Bar Chart. Listing F.5 (days13.py) uses a bar chart to shows the day distribution of the 13th across the months in the years 1800 to 2201. bar() is supplied with the bar names and a list of heights.

```
from datetime import date
import matplotlib.pyplot as plt


WEEKDAYS = ['Mon','Tue','Wed','Thur','Fri','Sat','Sun']
days13 = [0]*7
for year in range(1800, 2201):
  for month in range(1,13):
    days13[ date(year, month, 13).weekday()] += 1
print(*days13)
plt.bar(WEEKDAYS, days13)  # labels, heights
plt.title('The 13th Days')
plt.show()
```

Listing F.5. A tally of days numbered 13

Fig. F.4 is far from ideal, and we'll look at how to improve it in a later section.

F.2 Common Chart Variants

There are three common chart variants: plotting several curves in a single chart (to aid their comparison), adding error bars to line and bar plots, and changing one or both axes to use a logarithmic scale.

F.2.1 Multiple Curves in a Single Graph. Listing F.6 (taylorcos.py) draws the cosine curve and two Taylor series approximations of degrees 2 and 4 in the same chart (Fig. F.5).

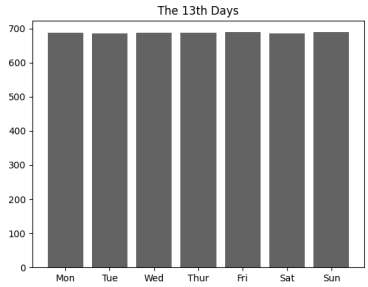


Figure F.4. A bar chart of 13th's

```
xs = linspace(-6, 6, 50)

# Plot y = cos(x)
ys = [ math.cos(x) for x in xs]
plt.plot(xs, ys, 'b', label ='cos(x)')  # solid blue

# Plot degree 2 Taylor polynomial
ys2 = [ (1 - x**2/2) for x in xs]
plt.plot(xs, ys2, 'r-.', label ='Degree 2')  # red dot dashes

# Plot degree 4 Taylor polynomial
ys4 = [(1 - x**2/2 + x**4/24) for x in xs]
plt.plot(xs, ys4, 'g:', label ='Degree 4')  # green dots

plt.grid(True, linestyle =':')  # a dotted grid pattern
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.xlim([-6, 6])
plt.ylim([-4, 4])

plt.legend()
plt.title('Taylor Polynomials of cos(x) at x = 0')
plt.show()
```

Listing F.6. Cosine and approximations

There are three calls to plot(), and to distinguish between the drawn curves we've used different line colors, patterns, and labels, which are printed on the graph by legend(). Two other additions are the use of grid lines and the limiting of the x- and y- axis ranges.

F.2.2 Error Bars. Error bars are most often added to line plots and bar charts. Since the coding is similar in both cases, we'll just explain the parabolic line graph here (Listing F.7; errParabola.py).

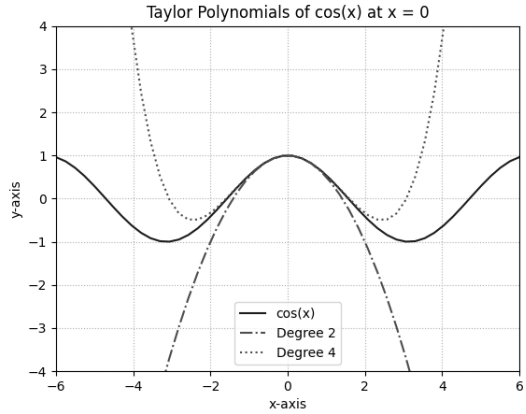


Figure F.5. Multiple lines in a graph

```
n = 10
xs = linspace(-2, 2, n)
ys = [x**2 for x in xs]
plt.plot(xs, ys, label="y=x^2")

yErrs = linspace(0.2, 0.5, n)
plt.errorbar(xs, ys, yerr = yErrs, fmt ='o')

plt.title("Parabola with Error Bars")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```

Listing F.7. A parabola with error bars

The lengths of the error bars are passed to errorbar() along with how the error symbol should be drawn (see Fig. F.6).

F.2.3 Logarithmic Axes. Listing F.8 (loglog.py) contains three alternative ways to plot the same coordinates – using plot() as seen previously, and also calls to semilogx(), semilogy(), and loglog() which automatically scale the data on the x-axis, or y-axis, or both. Note that there's no need to modify the coordinate data.

```
xs = linspace(1, 10, 500)
ys = [x**3 for x in xs]
# plt.plot(xs, ys)

# plt.semilogx(xs, ys)  # x-axis log
# plt.semilogy(xs, ys)  # y-axis log
plt.loglog(xs, ys)   # both axes use logs
```

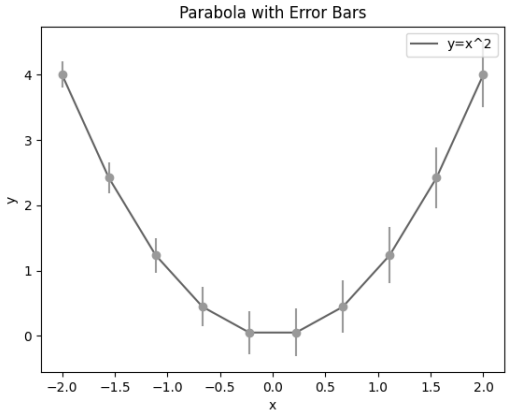


Figure F.6. Error bars on a parabola

```
plt.title("Log plot")
plt.show()
```

Listing F.8. Log scaling of axes

F.3 Customizing a Plot

There are many, many ways to adjust how graph elements are drawn. When looking for a particular matplotlib function for the task, it helps to know what an element is called. Fig. F.7, taken from the matplotlib documentation, lists the main ones.

Listing F.9 (parabolaCustom.py) includes the adjustments that we use most frequently in the main text.

```
xs = linspace(-2, 2, 10)
ys = [x**2 for x in xs]

fig = plt.figure(figsize = (10, 8)) # bigger window

plt.axis('equal')  # same scaling used on both axes
plt.xlim(-2.5, 2.5)
plt.minorticks_on()
plt.xticks( list(frange(-2.5, 3, 0.5)))

plt.axhline(0, color="Green")  # horizontal line
plt.axvline(0, color="Green")  # vertical line

plt.plot(xs, ys, alpha = 0.4, label ='$y = x^2$', # Latex
```

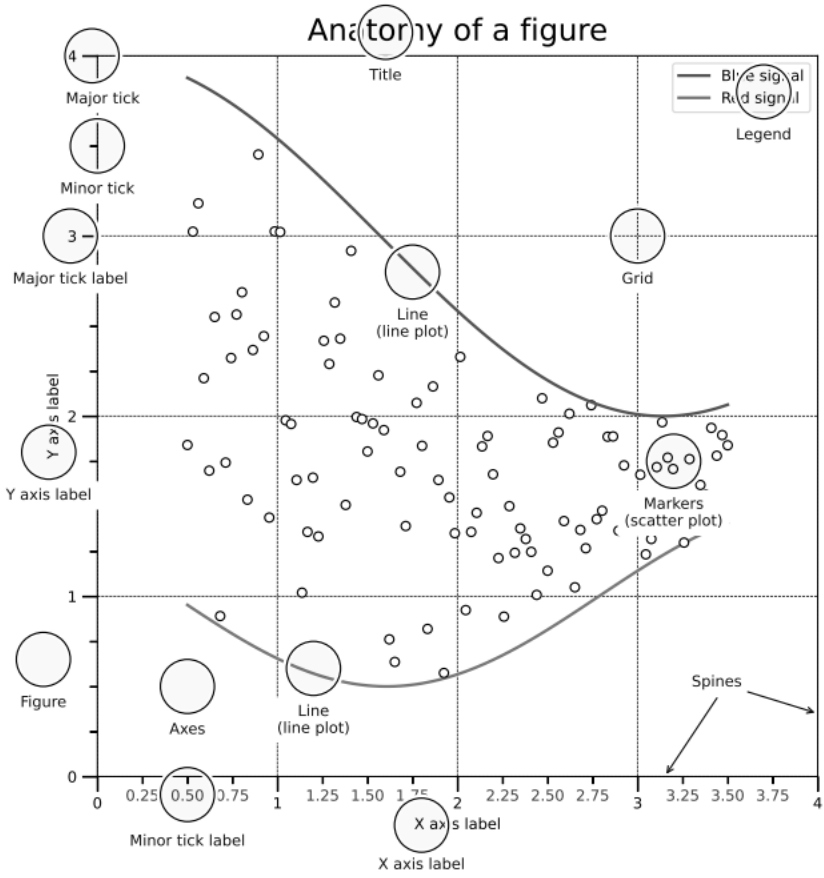


Figure F.7. Names of figure elements

```
                color ='red', linestyle ='--',
                linewidth = 2, marker ='D',
                markersize = 5, mfc ='blue', mec ='blue')
                      # marker face and edge colors

plt.title('Parabola plot', fontsize = 20) # bigger title
plt.xlabel('x')
plt.ylabel('y')
fig.canvas.manager.set_window_title('Parabola plot')
             # give the window title bar a better name

fig.text(0.8, 0.15, 'Some info',
     fontsize = 12, color ='green',
```

```
      ha ='right', va ='bottom',
      alpha = 0.7)     # draw text at (0.8, 0.15) offset

plt.grid(alpha =.6, linestyle ='--')  # grid lines
plt.legend(loc="upper left")      # move the legend

print("Saving plot to savedplot.png")
plt.savefig("savedplot.png", dpi=300)

plt.show()
```

Listing F.9. Customization code

It helps to cross-reference the source with what it generates in Fig. F.8.

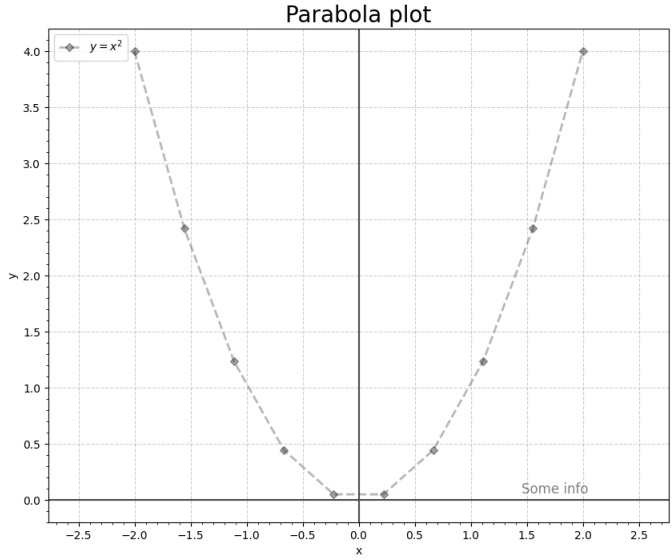


Figure F.8. Customization results

Perhaps the hardest customizations to remember are the marker line and point styles (e.g. '–' and 'D' in Listing F.9). The common ones are listed in Tables F.1 and F.2.

The bar chart back in Fig. F.4 is hard to read, but there are two customizations that can improve its look: the y-axis doesn't need to start at 0, and the bars can be labeled with their numerical values. The relevant lines are shown in Listing F.10 (daysCustom.py), and the resulting graph is Fig. F.9.

```
MIN = 680
bots=[MIN]*7
```

Label	Style
-	Solid
–	Dashed
:	Dotted
-.	Dash-dot

Table F.1. Marker line styles

Label	Marker	Style
.	·	Point
o	∘	Circle
+	+	Plus
x	×	Cross
D	⋄	Diamond
v	▽	Downward Triangle
^	△	Upward Triangle
s	□	Square
*	★	Star

Table F.2. Marker point styles

```
ds = [val-MIN for val in days13]   # cut off the heights

plt.bar(WEEKDAYS, ds, width=0.4, bottom=bots)
                           # set vertical baseline

xlocs, xlabs = plt.xticks()
for i, v in enumerate(days13):
  plt.text(xlocs[i] - 0.2, v + 0.08, str(v))
```

Listing F.10. Customizing a bar chart

The trickiest part of the code is determining the positions of the labels that appear above the bars. The x and y values are obtained in different ways. The x is retrieved via a call to xticks() which returns the positions of the tick marks on the x-axis. The y value is calculated from the heights in the `days13` list. Both are slightly modified so that the text is better positioned.

Another complication is that the heights of the bars on the graph do *not* come from `days13`, but from the modified `ds` list which holds smaller heights. The bars are drawn correctly since the bar() call includes a bottom argument which sets the start of the y-axis.

A minor change is that we prefer the bars to be narrower, which is achieved by setting the width argument in bar().

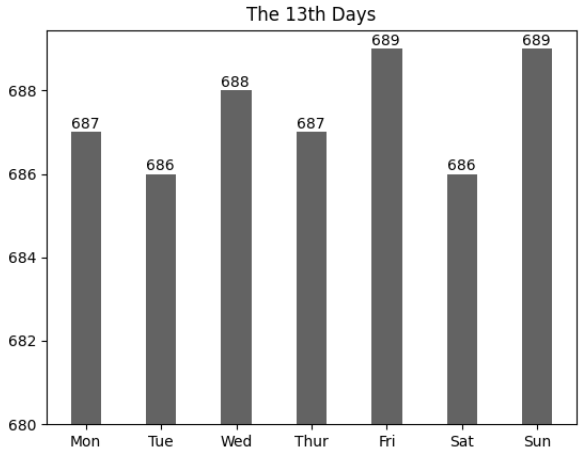


Figure F.9. Customized bar chart of 13th's

F.4 Other Useful Plot Types

Matplotlib supports a very large variety of plot types (e.g look at `https://matplotlib.org/stable/plot_types/`). In this section, we'll give brief examples of five that we've used – the histogram, the box plot, the polar chart, the contour lines plot, and the heat map.

F.4.1 The Histogram. Listing F.11 (heights.py) reads in the heights of high school boys and girls from a CSV file, and creates two box plots. The means and standard deviations of the two groups are also reported as vertical lines – a solid line for the mean, and two dashed lines for mean ± stdev. The graph is shown in Fig. F.10.

Listing F.11 utilizes buildDist() to load data from the CSV file into a dictionary of key-list pairs. The keys are the data's column names, and each list holds all the values from a column as floats.

```
def buildDist(fnm):
  columns = defaultdict(list)
  with open(fnm) as f:
    reader = csv.DictReader(f)
    for row in reader:
      # read a row as {column1: value1, column2: value2,...}
      for (k,v) in row.items():
        columns[k].append(float(v))
  return columns

def stats(label, heights, col):
  # report the mean and stdev for the supplied heights data
```

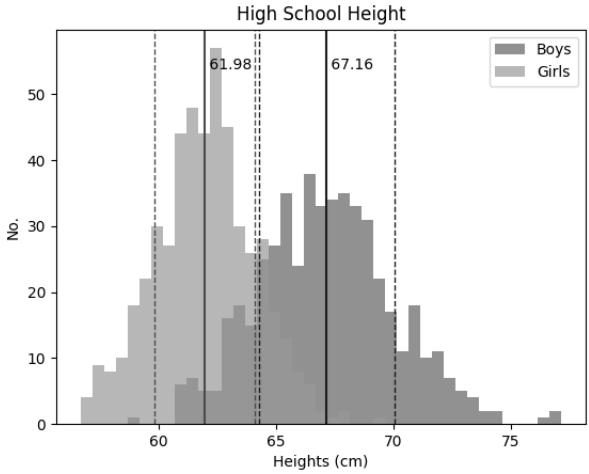


Figure F.10. Histograms of boys and girls heights

```
  mean = statistics.mean(heights)
  stdev = statistics.stdev(heights)
  print("Statistics ("+ str(label) +")")
  print(f"Mean: {mean:.2f} ; stdev: {stdev:.2f}")
  minY, maxY = plt.ylim()
  plt.axvline(mean, color=col)
  plt.text(mean, maxY*0.9, f" {mean:.2f}")
  plt.axvline(mean+stdev, color=col, ls='dashed', lw=1)
  plt.axvline(mean-stdev, color=col, ls='dashed', lw=1)

# load data
columns =  buildDist('hs_heights_pair.csv')
boys = columns['boys']  # use CSV data column labels
girls = columns['girls']

# calculate a range for the box plots bins, and
# split into boxes of width 0.5
minHeight = min([min(boys),min(girls)])
maxHeight = max([max(boys),max(girls)])
bins = list(frange(minHeight, maxHeight+0.5, 0.5))

plt.hist(boys, bins, label="Boys", alpha=0.7)
plt.hist(girls, bins, label="Girls", alpha=0.7)

# report the means and stdevs for the two groups
stats("Boys", boys, "blue")
stats("Girls", girls, "green")

plt.xlabel('Heights (cm)')
```

```
plt.ylabel('No.')
plt.title('High School Height')
plt.legend()
plt.show()
```

Listing F.11. Building a histogram

The CSV file, `hs_heights_pair.csv`, begins like so:

```
boys,girls
64.44,62.47
67.4,63.27
63.93,62.99
   : # many more lines, not shown
```

This format means that the columns dictionary returned by buildDist() will contains two lists, `boys` and `girls`.

A difficult aspect of histograms can be how to decide on the bins range, and the size of each bin. Listing F.11 automates the first task by finding the minimum and maximum values in the two data lists; however, the bin size is fixed at 0.5.

It's often useful to include mean and standard derivation information with a histogram, which is handled by stats(). It prints the information to standard output, and draws lines on the graph along with the mean values.

F.4.2 The Box Plot. A box plot summarizes data having a minimum, first quartile, median, third quartile, and maximum. These elements are illustrated in Fig. F.11.

A box extends from the first data quartile (Q1) to the third (Q3), which is termed the Interquartile Range (IQR). A line goes through the box to mark the median, and 'whisker' lines extend from the top and bottom to 'maximum' and 'minimum' values. These aren't the smallest and largest numbers in the data, but positions marking Q1 - 1.5*IQR and Q3 + 1.5*IQR, with any data outside these ranges being classed as outliers. The display of the mean is an optional extra, but sometimes useful.

The output from Listing F.12 (boxplots.py) is shown in Fig. F.12.

```
data1 = [ random.gauss(100, 10) for _ in range(200)]
data2 = [ random.gauss(90, 20) for _ in range(200)]
data3 = [ random.gauss(80, 30) for _ in range(200)]
data4 = [ random.gauss(70, 40) for _ in range(200)]
ds = [data1, data2, data3, data4]

plt.boxplot(ds, showmeans=True)  # include means
plt.xticks([1, 2, 3, 4], ["Jim", "John", "Joan", "Jane"])
                          # change the x-axis labels
plt.show()
```

Listing F.12. Drawing box plots

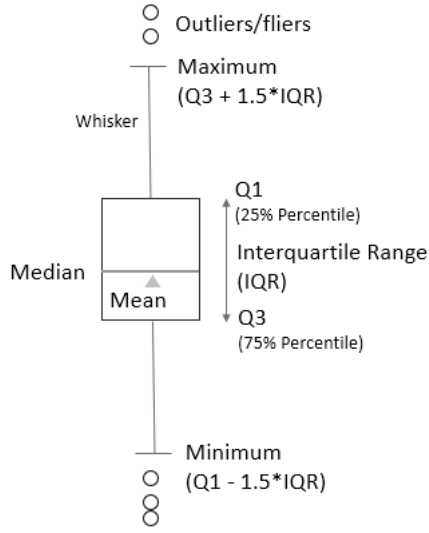


Figure F.11. Elements of a box plot

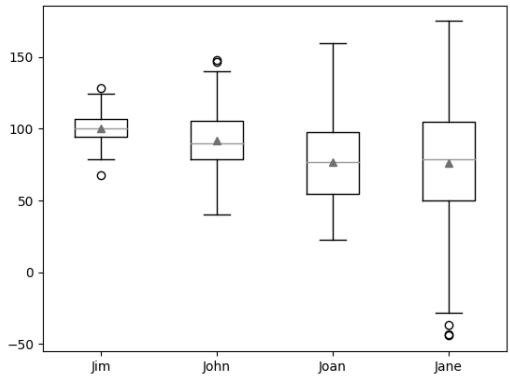


Figure F.12. Student box plots

F.4.3 The Polar Chart. Polar plots utilize the polar() function which is passed lists of θs and radii. Listing F.13 (butterfly.py) generates the butterfly curve shown in Fig. F.13.

```
n = 5 # no. of loops
# generate radii from angles
thetas = linspace(0, 2*math.pi*n, 100*n)
```

```
rs = [ math.exp(math.sin(theta)) - 2*math.cos(4*theta) +
       math.sin(theta/12)**5
                    for theta in thetas]
plt.polar(thetas, rs)
plt.title("Butterfly Curve")
plt.show()
```

Listing F.13. Generating a butterfly

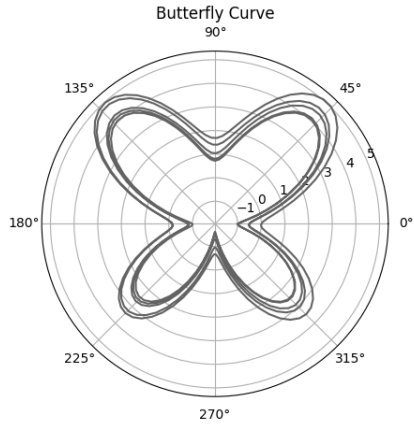


Figure F.13. The butterfly curve

Listing F.14 (spiral.py) shows how to customize the polar axes (e.g. reduce the number of radial ticks and move the radial labels). This requires access to the axis variable, which means that the polar plot must be drawn using the axis plot() function:

```
# generate angles from the radii
rs = linspace(0, 2, num=100)
thetas = [ 2*math.pi*r for r in rs]

ax = plt.subplot(projection='polar')
ax.plot(thetas, rs)
```

Listing F.14. Part of spiral.py

F.4.4 The Contour Lines Plot. A contour plot is our first 3D graph, albeit one using a 2D mapping. It's a member of matplotlib's "Gridded Data" class of plots, which also include filled contour plots and quiver charts (`https://matplotlib.org/stable/plot_types/arrays/`).

Listing F.15 (contourLines.py) plots the difference of two 2D Gaussian functions, with labels drawn on their contour lines (see Fig. F.14).

```
y = np.arange(-2.0, 2.0, 0.025)
X, Y = np.meshgrid(x, y)   # numpy output required

# build curve for generating z-coords
Z1 = np.exp(-X**2 - Y**2)  # 2D Gaussian
Z2 = np.exp(-(X-1)**2 - (Y -1)**2)
Z = (Z1 - Z2) * 2  # difference of Gaussians

cts = plt.contour(X, Y, Z)   # , colors="k")
plt.clabel(cts, inline=True, fontsize=10)
plt.title('Labeled Contour Lines')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Listing F.15. Labeled contour lines representing a 3D equation

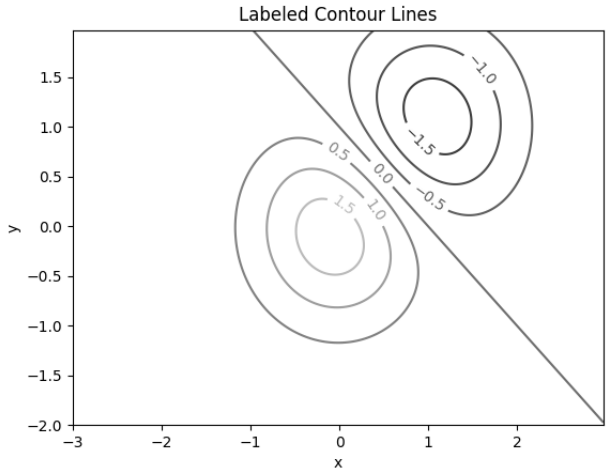


Figure F.14. A contour lines plot

A useful variant is to add the argument `colors="k"` to contour(), which draws the contours in black, with negatives represented by dashed lines.

The definition of the equation requires that the x- and y- coordinates be organized into a numpy meshgrid, which changes the x- and y- axis data into 2D arrays; the essential idea is shown in Fig. F.15.

This permits the equation to be written very succinctly since the X and Y numpy arrays are automatically combined element-by element. The corresponding equation is:

$$z = 2 \times (\exp^{(-x^2-y^2)} - \exp^{(-(x-1)^2-(y-1)^2)})$$

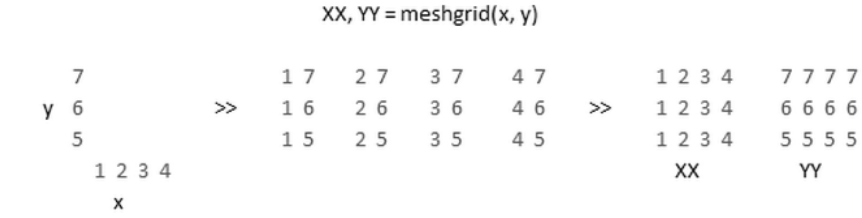


Figure F.15. Converting data into a numpy meshgrid

F.4.5 The Heat Map. A heat map is a way of displaying 3D data in a 2D form when the z- values can be stored as a 2D array or list.

Listing F.16 (colorMap.py) produces a *categorical* heat map (Fig. F.16) since the x- and y- data are categories (in this case, student names and subjects).

```
nms = ['Sam','Jim','John','Sue','Mike','Jane']
subjs = ['Maths','Physics','English','Chemistry',
         'History','Comp Sci']
marks = [[50, 74, 40, 59,90, 98],
         [72, 85, 64, 33, 47, 87],
         [52, 97, 44, 73, 17, 56],
         [69, 45, 89, 79,70, 48],
         [87, 65, 56, 86, 72, 68],
         [90, 29, 78, 66, 50, 32]]

# display the categories as x- and y- axis labels
plt.xticks(ticks=range(len(nms)), labels=nms, rotation=90)
plt.yticks(ticks=range(len(subjs)),labels=subjs)

hm=plt.imshow(marks,cmap='Blues_r',interpolation="nearest")
plt.colorbar(hm)
plt.title("Students and their Subject Marks")
plt.show()
```

Listing F.16. A blue color map relating students, subjects, and their scores

The map is displayed using imshow() which draws a 2D list (or numpy array) as a colored checkboard. In fact, imshow() can also be passed a RGB image for rendering (as the function name suggests).

Fig. F.16 is colored using a reversed version of one of matplotlib's sequential colormaps, 'Blues'. Other map categories include diverging, cyclic, and qualitative; for details see `https://matplotlib.org/stable/users/explain/colors/colormaps.html`. It also includes a color bar, which explains how colors are assigned to values.

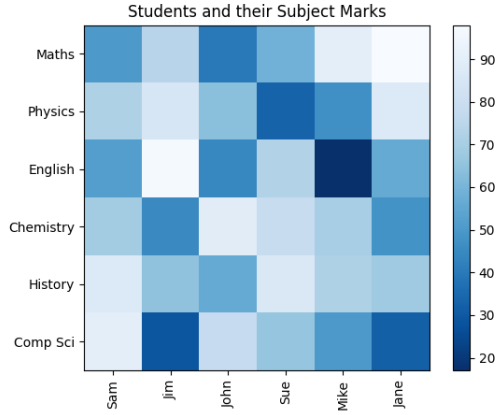


Figure F.16. A heat map of student marks

F.5 3D Plots

This section describes four types of 3D charts: line, scatter, wireframe, and surface plots. Several more can be found at `https://matplotlib.org/stable/plot_types/3D/`, and there are numerous examples at `https://matplotlib.org/stable/gallery/mplot3d/`.

One very nice feature common to all 3D plots is that matplotlib's window controls allow a figure to be rotated in addition to being panned and zoomed.

F.5.1 3D Lines or Points. The generation of 3D line charts and scatter plots is so similar that we've combined the examples into one program, Listing F.17 (points3D.py). If plot() is called then a line appears as in Fig. F.17(a), whereas scatter() generates Fig. F.17(b).

```
ax = plt.figure().add_subplot(projection ='3d')

# define data
zs = linspace(0, 1, 100)
xs = [ z * math.sin(25 * z) for z in zs]
ys = [ z * math.cos(25 * z) for z in zs]

plt.plot(xs, ys, zs)     # line
# plt.scatter(xs, ys, zs=zs, s=70)  # points; s is size

plt.title('3D plot')
plt.xlabel('x')
plt.ylabel('y')
ax.set_zlabel('z')    # no plt.zlabel()
plt.show()
```

Listing F.17. Rendering 3D points

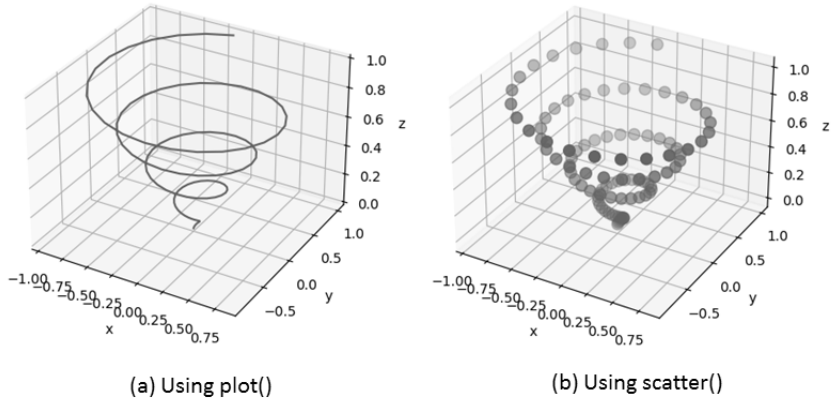


Figure F.17. 3D line or points

For plot() and scatter() to generate 3D plots rather than their usual 2D graphs, it's necessary to set the projection attribute to '3d'. Also, z-axis labeling must be set via an axis function.

F.5.2 A Wireframe Plot. Producing a wireframe, and the surface plot of the next section, requires that the x- and y- data be represented by a numpy meshgrid (which we met when drawing contour lines). Listing F.18 (wire3D.py) produces the graph in Fig. F.18.

```
X = np.linspace(-1, 2, 20)  # increase step for more detail
Y = np.linspace(-1, 2, 20)
X, Y = np.meshgrid(X, Y)  # numpy output required
Z = np.sin(np.sqrt(X**2 + Y**2))

ax = plt.axes(projection = '3d')
ax.plot_wireframe(X, Y, Z)
plt.title('3D Wireframe Plot')
plt.xlabel('x')
plt.ylabel('y')
ax.set_zlabel('z')  # no plt.zlabel()
plt.show()
```

Listing F.18. A wireframe plot of a 3D equation

The distance between the wires is controlled by the intervals between the x- and y- values. It's also possible to set the spacing via 'count' or 'stride' arguments to plot_wireframe().

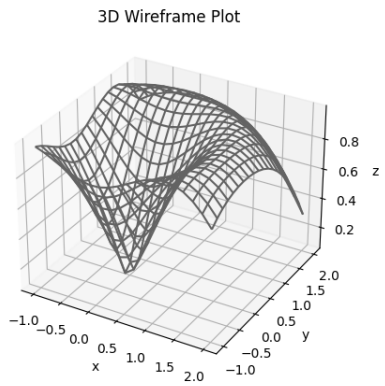


Figure F.18. A wireframe

F.5.3 A Surface Plot. A surface plot can be thought of as a filled and colored wireframe, and its equation uses a meshgrid in the same way. The differences come in the optional coloring arguments to plot_surface(). The simplest approach is to use a predefined colormap (which we met when using heat maps). It's also a good idea to adjust the color and thickness of the wires.

Listing F.19 (surface3D.py) produces the graph in Fig. F.19.

```
x = np.linspace(-10, 10, 40)
y = np.linspace(-10, 10, 40)
X, Y = np.meshgrid(x, y)    # numpy output required
Z = fn(X, Y)

ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z,
    cmap="Blues_r", edgecolor="Black", lw=0.2)

plt.title('Surface Plot of fn'+str(n))
plt.xlabel('x')
plt.ylabel('y')
ax.set_zlabel('z')  # no plt.zlabel()
plt.show()
```

Listing F.19. Generate a 3D mesh of a function

We haven't shown all of the code in Listing F.19. There are seven equations defined as Python functions, and the user must input a number to select a function to generate the surface. Function no. 6 shown in Fig. F.19 is:

```
def fn6(x, y):   # monkey saddle
  r = np.sqrt(x*x + y*y)
  return x*y*(x-y)*(x+y)/r
```

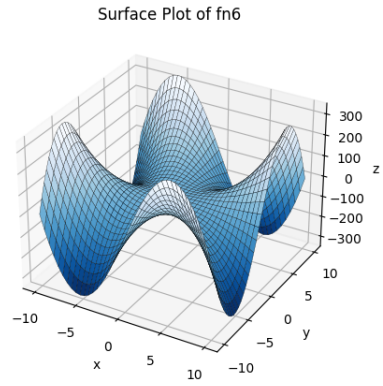


Figure F.19. A surface plot of a monkey saddle

F.6 Animation

animBall.py utilizes matplotlib's FuncAnimation() to animate the trajectory of a ball as a series of points plotted over time. Fig. F.20 shows two frames from the animation.

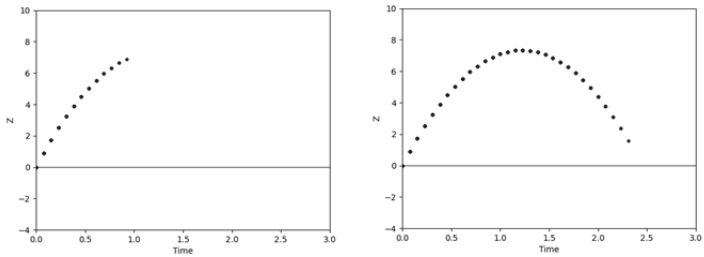


Figure F.20. Two frames from the ball animation

The important code from animBall.py:

```
ts = linspace(0, 3, 40)
g = -9.81
v0 = 12
zs = [((g*t**2)/2 + v0*t) for t in ts]

fig, ax = plt.subplots()
anim = animation.FuncAnimation(fig, update,
                frames=len(ts)-1, interval=50) # ms
```

After all of the points have been generated, FuncAnimation() automatically calls update() every 50 ms. The function only draws *some* of those points depending on the current animation frame counter.

```
def update(i):
  if i == 0:  # reset for next play
    ax.clear()
    ax.set(xlim=[0, 3], ylim=[-4, 10],
           xlabel='Time', ylabel='Z')
    ax.axhline(y=0, color='k', alpha=0.5)
  ax.scatter(ts[:i], zs[:i], c="b", s=10)
```

The current frame number, *i*, is used on the last line to call scatter() with the data sliced to only include the first *i* points.

Although the function called by FuncAnimation() can have any name, it can only receive the frame number as an input argument. That means that the rest of the plotting data (e.g. the `ts` and `zs` lists, and `ax`) must be globals.

An extra feature of `animBall.py` is the ability for the user to pause/resume the animation by clicking anywhere inside the window. This is implemented by attaching a listener to button press events:

```
fig.canvas.mpl_connect('button_press_event', onClick)
animRunning = True
```

When a press event is detected, onClick() is called:

```
def onClick(event):
  global animRunning
  if animRunning:
    anim.event_source.stop()
    animRunning = False
  else:
    anim.event_source.start()
    animRunning = True
```

As with update(), the function can be called anything but can only accept an event value. As a consequence, the `animRunning` boolean must be declared global since its value is toggled inside the function. Event processing is disabled or re-enabled, which stops or restarts FuncAnimation().

The matplotlib user guide contains a good description of its animation features (`https://matplotlib.org/stable/users/explain/animations/animations.html#animations`), and the examples page has a section dedicated to animations (`https://matplotlib.org/stable/gallery/index.html#animation`). The button press functionality is part of matplotlib's event handling features, which has its own section on the examples page (`https://matplotlib.org/stable/gallery/event_handling/`).

F.7 Widgets

Widgets are GUI controls, such as sliders and buttons, which can be used to control and modify a plot interactively. These controls work in a similar way to FuncAnimation() (see the previous section), but instead of responding to time events they're triggered by the user clicking on the control; a function is then called to modify the plot.

Listing F.20 (sineSlider.py) draws the sine wave shown in Fig. F.21, and allows its frequency to be adjusted via a slider.

```
def update(value):
  ys = [ math.sin(value * math.pi * x) for x in thetas]
  line.set_ydata(ys)
  fig.canvas.draw_idle()

fig, ax = plt.subplots()
plt.subplots_adjust(bottom=0.2)
thetas = linspace(0, math.pi, 300)
ys = [ math.sin(5 * math.pi * x) for x in thetas]
line, = ax.plot(thetas, ys)

# frequency slider
freqAxis = plt.axes([0.25, 0.1, 0.65, 0.03])
freqSlider = Slider(freqAxis, 'Frequency [Hz]',
                            0, 10, valinit=5)
                         # min, max, initial value
freqSlider.on_changed(update)
plt.show()
```

Listing F.20. Using a slider to control the frequency of a sine wave

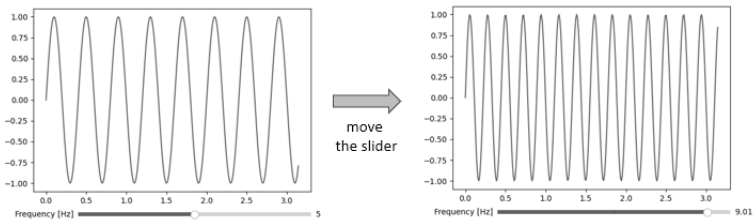


Figure F.21. Changing sine wave frequency

Setting up a control typically takes three lines of code: a line to specify the widget size in terms of axis values, the creation of the control, and the linking of the control to a function.

The matplotlib API documentation has a detailed description of widget features (`https://matplotlib.org/stable/api/widgets_api.html`), and the

examples page has a dedicated section on the topic (`https://matplotlib.org/stable/gallery/widgets/`).

Appendix G

Turtle Graphics

Turtle graphics were originally developed as part of the children's programming language LOGO by Wally Feurzeig, Daniel Bobrow, and Seymour Papert at MIT in the late 1960s [**AD86**]. Turtles went on to play a key role in the "constructionist learning" philosophy proposed by Papert in his seminal textbook *Mindstorms* [**Pap20**].

A turtle moves across the screen with a pen in its teeth (`https://www.turtleconservationsociety.org.my/do-turtles-have-teeth/`). Commands direct the turtle to lift the pen or lower it onto the screen, turn some number of degrees left or right, or move a specified distance forwards or backwards.

At any given moment, the turtle is located at a specific position, specified by (x, y) coordinates. The origin, $(0, 0)$, is at the center of the window, which is also the turtle's 'home'.

Another turtle attribute is its heading – the direction it is currently facing. The turtle's initial heading is 0 degrees, which is due east. The degrees of the heading increase as it turns to the left, so 90 degrees is due north.

In addition to its position and heading, other turtle settings include the pen's drawing color, the width of the line that is drawn, and whether the pen is up or down (i.e. is drawing or not).

The Python turtle documentation is excellent (`https://docs.python.org/3/library/turtle.html`), but we've included two tables of the commands that we've used the most (Tables G.1 and G.2) at the end of this appendix. They are by no means comprehensive, but cover the important functions.

G.1 Drawing Polygons

Almost all of our turtle code starts in the same way as Listing G.1 (drawPolys.py):

```
if __name__ == "__main__":
  rotAngle = int(input("angle? "))

  # create a turtle
  t = Turtle()
  t.hideturtle()
  t.speed(0)
  scr = t.getscreen()
  scr.title('Draw Patterns')

  polyCircles(t, rotAngle)

  scr.exitonclick()
```

Listing G.1. The start of a turtle program

An invisible, fast-moving turtle called *t* is created, and a reference is passed to the drawing function(s) (in this case, polyCircles()). A reference to the window is obtained as *scr*, and this is used to set the title and to keep the screen (the window) open after the drawing is finished. exitonclick() returns when it detects a mouse click inside the window, and lets the program finish.

polyCircles() calls rotatePolys() four times to draw a series of polygons rotated through 360 degrees.

```
from turtle import Turtle

LENGTH = 100

def drawPolygon(t, numSides):
  # draw a regular polygon with a specified no. of sides
  lineLen = LENGTH/(2*math.sin(math.pi/numSides))
  if (numSides > 2) and (lineLen > 0):
    angle = 360 / numSides
    for i in range(numSides):
      t.forward(lineLen)
      t.right(angle)

def rotatePolys(t, numSides, rotAngle):
  # draw a series of polygons in a circle,
  # spaced out at rotAngle degrees
  numShapes = 360 // rotAngle
  for shape in range(numShapes):
    drawPolygon(t, numSides)
    t.right(rotAngle)

def polyCircles(t, rotAngle):
  # draw four circles of polys in different colors
```

```
  t.pensize(3)
  t.pencolor('blue')
  rotatePolys(t, 3, rotAngle) # triangles
  t.pencolor('red')
  rotatePolys(t, 4, rotAngle) # squares
  t.pencolor('green')
  rotatePolys(t, 5, rotAngle) # penatgons
  t.pencolor('orange')
  rotatePolys(t, 6, rotAngle) # hexagons
```

Listing G.2. Drawing polygons

The end result is shown in Fig. G.1. This was produced after the user entered 10 degrees as the value for `rotAngle`.

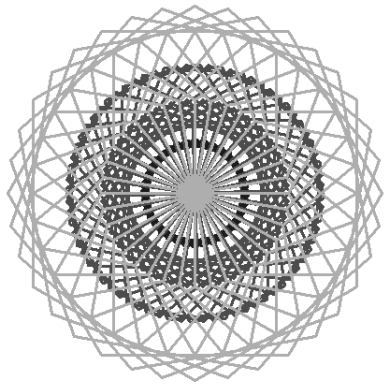


Figure G.1. Circles of polygons

G.2 Responding to Mouse Clicks

Another common top-level form for a turtle program is to use onclick() and mainloop() to listen for mouse clicks to trigger function calls (Listing G.3; drawClick.py).

```
from turtle import Turtle

def drawClick(x, y):
  t.goto(x, y)
  t.write(str(x)+","+str(y))

t = Turtle()
t.hideturtle()
t.speed(0)
scr = t.getscreen()
```

```
scr.title('DrawToClicks')
scr.onclick(drawClick)
scr.mainloop()
```

Listing G.3. Responding to mouse clicks

A mouse click causes drawClick() to execute, which moves the turtle from its current position to the click point, drawing a line in the process, and writing the coordinate details to the window (see Fig. G.2). Note, that it's not possible to pass a turtle reference to drawClick(), so *t* must be global.

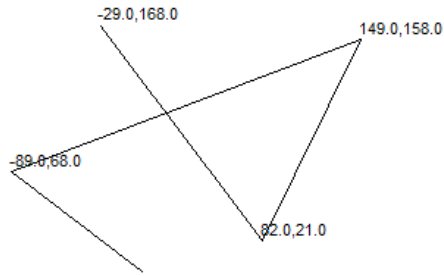


Figure G.2. Lines drawn by clicking

This approach means that the program must be terminated by clicking on the turtle window's close box.

G.3 Beautiful Recursion

Listing G.4 (spirals.py) shows that recursion is easy to illustrate with turtle graphics. The patterns in Fig. G.3 were generated by calling spiral() with slightly different arguments (some examples are commented out in the code).

```
def spiral(t, sideLen, angle, scaleFactor, minLength):
  # Draw a spiral recursively."""
  if sideLen >= minLength:
    t.forward(sideLen)
    t.left(angle)
    spiral(t, sideLen*scaleFactor, angle,
                scaleFactor, minLength)

t = Turtle()
t.hideturtle()
t.speed(0)
```

```
scr = t.getscreen()
scr.title('Draw Spiral')
# spiral(t, 200, 90, 0.8, 10)
# spiral(t, 200, 72, 0.97, 10)
# spiral(t, 200, 80, 0.95, 10)
# spiral(t, 200, 121, 0.95, 15)
spiral(t, 200, 95, 0.93, 10)
scr.exitonclick()
```

Listing G.4. Draw a spiral

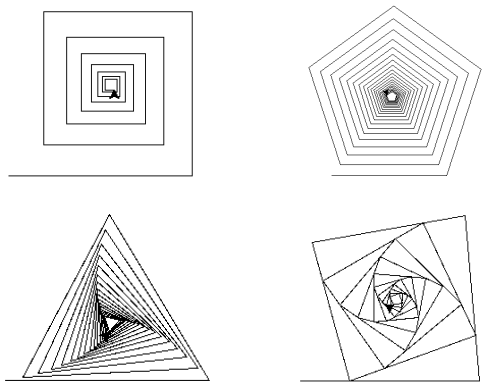


Figure G.3. Some possible spirals

G.4 Multiple Turtles

Many turtle examples don't bother creating an explicit turtle, *t*, as we have done, because a default turtle is always available. We prefer the explicit approach since it makes it clearer what's being manipulated by our functions.

It's possible to create multiple turtles, as in Listing G.5 (drunks.py), which also shows Python's random module making turtle behavior less predictable.

```
MAX_STEPS = 50
STEP_SIZE = 50
colors=["red","orange","green","blue","purple","violet"]

n = int(input("n? "))
# create n turtles
ts = [ Turtle() for _ in range(n)]
for i in range(n):
  ts[i].setheading(360*random.random())
```

```
  ts[i].speed(0)
  ts[i].pencolor(colors[i % len(colors)])
scr = ts[0].getscreen()
scr.title('Drunk Turtles')

# move randomly for max steps
for t in range(MAX_STEPS):
  for i in range(n):
    ts[i].left(360 * random.random())
    ts[i].forward(random.randint(STEP_SIZE//5, STEP_SIZE))

scr.exitonclick()
```

Listing G.5. Drunk turtles on the move

When the user creates five turtles (each with a different colored pen), something like Fig. G.4 is generated.

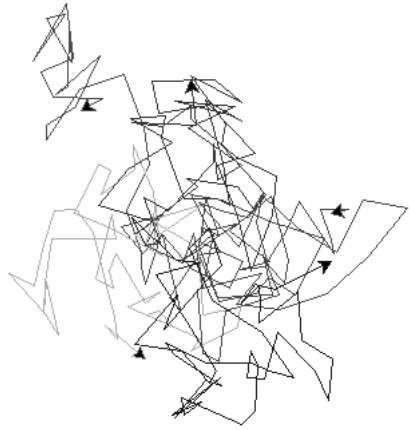


Figure G.4. Five drunk turtles

G.5 LOGO Flowers

The last program of this section produces the most complex turtle drawing we've seen (Fig. G.5), although it only uses basic features.

The program was originally written in LOGO by William Weinreb for *BYTE Magazine*, Nov. 1982 (`https://archive.org/details/byte-magazine-1982-11/page/n119/mode/2up`). We've translated the code into Python, which you can find in `flowers.py`.

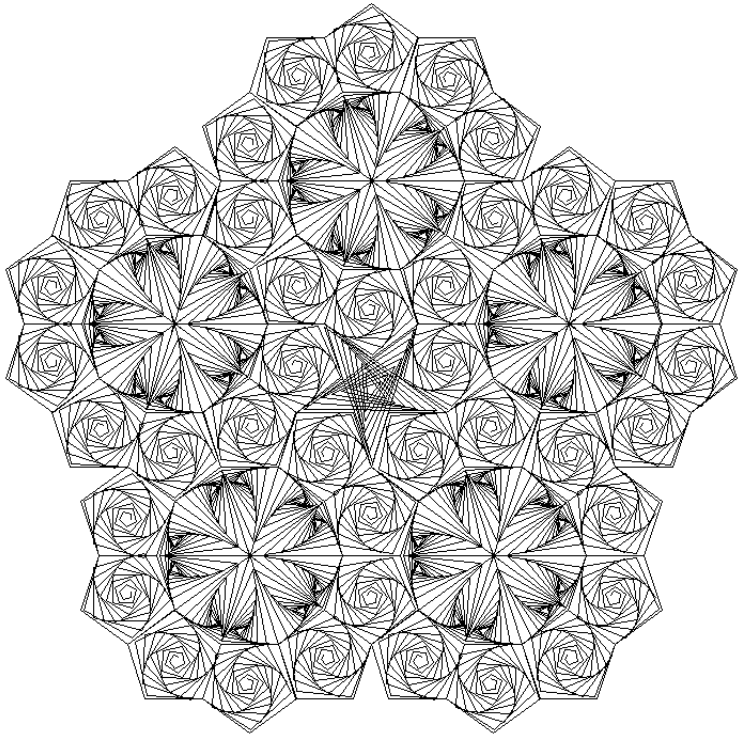


Figure G.5. A bunch of flowers

Turtle Methods	Description
t = Turtle()	Creates a new Turtle object and opens its window.
t.penup(); t.pendown()	Raises/lowers *t*'s pen from/to the drawing surface.
t.isdown()	Returns True if *t*'s pen is down, otherwise False.
t.left(degrees); t.right(degrees)	Rotates *t* to the left or right by the specified degrees.
t.forward(distance)	Moves *t* the specified distance in the current direction.
t.backward(distance)	Moves *t* the specified distance in the opposite of the current direction.
t.hideturtle(); t.showturtle()	Makes *t* invisible or visible (but doesn't affect the pen).
t.goto(x, y)	Moves *t* to the specified position.
t.position()	Returns the current (x, y) position of *t*.
t.xcor(); t.ycor()	Returns the turtle's x or y coordinate.
t.heading()	Returns the current direction of *t*.
t.setheading(degrees)	Points *t* in the indicated direction, which is specified in degrees. East is 0 degrees, north is 90 degrees, west is 180 degrees, and south is 270 degrees.
t.home()	Moves *t* to the center of the window and then points *t* east.
t.pencolor(r,g,b); t.pencolor(string)	Changes the pen color of *t* to the specified RGB value or string, such as 'red'. Returns the current color of *t* when the arguments are omitted.
t.pensize(width)	Sets the pen's drawing line to the given width.
t.speed(s)	Sets the turtle's speed: 1 is slowest, 10 is fast, 0 is the fastest.
t.write(message)	Writes message at *t*'s current position.

Table G.1. Commonly used turtle functions

Screen Methods	Description
scr = t.getscreen()	Returns a reference to the turtle's drawing window.
scr.setup(width, height, x, y)	Sets the size and location of the window.
scr.setworldcoordinates(x1, y1, x2, y2)	Sets the coordinates of the window to $(x1, y1)$ at the lower left and $(x2, y2)$ at the upper right.
scr.exitonclick()	Causes the window to close when clicked.
scr.onclick(function)	Sets up an automatic call to a function whenever a mouse click is detected; the coordinates of the click are passed to the function.
scr.mainloop()	Makes the program enter an event processing loop to listen for mouse clicks and other events.

Table G.2. Commonly used turtle screen functions

Appendix H

Graphviz

Graphviz is a simple, but powerful, package for drawing graphs (both undirected and directed). It was originally developed at AT&T Labs to process diagrams described by the DOT language (`https://graphviz.org/`). The Python version of Graphviz (`https://graphviz.readthedocs.io/en/stable/`) is somewhat different in that it offers an API for creating nodes, edges, and clusters (subgraphs), along with many attributes and layout managers for tweaking the look of a graph. The API can generate DOT files, but all of our examples use the API to create and display graphs directly.

An important issue concerning the distinction between Graphviz and its Python version is that much of the online help about the package uses examples written in the DOT language, which isn't of immediate help when looking for a particular method in the Python API.

The great strength of Graphviz is that the difficult task of laying out the nodes and edges of a graph is left to the software. However, this is also sometimes a weakness, since it's not easy to wrest layout control away from Graphviz; if you want to position lots of graph elements in specific places then Graphviz may not be the right tool for the job.

H.1 A Simple Undirected Graph

The code in Listing H.1 (simpGraph.py) generates an undirected graph using Graphviz's default 'dot' layout engine – the nodes are drawn in a top-to-bottom order based on the ordering of the edges in the code.

```
import graphviz
```

```
TEMP_FNM = 'temp_graph'

g = graphviz.Graph(format='png')  # use Graph or Digraph
# g.attr(rankdir='LR')     # to lay out left-to-right
g.attr(label='A Simple Graph')

g.edge('1', '2');  g.edge('2', '3'); g.edge('3', '1')
g.edge('1', '4');  g.edge('4', '5'); g.edge('4', '6')
g.edge('4', '7');  g.edge('6', '8'); g.edge('8', '9')
g.edge('9', '10'); g.edge('9', '7'); g.edge('10', '7')
g.edge('6', '7')

g.render(filename=TEMP_FNM, view=True)
```

Listing H.1. Generate a Top-to-Bottom Undirected Graph

This program displays the image shown in Fig. H.1, which is also saved to `temp_graph.png`.

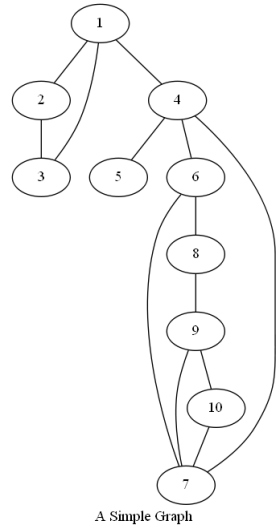


Figure H.1. A Top-to-Bottom Undirected Graph

Note that Graphviz has inferred the graph's nodes from the strings used in the edge() calls.

The `cleanup=True` argument of render() removes any unnecessary files, which in this case will be a DOT text file called `temp_graph`. It contains:

```
graph {
        label="A Simple Graph"
        1 -- 2;  2 -- 3; 3 -- 1
```

```
        1 -- 4;  4 -- 5; 4 -- 6
        4 -- 7;  6 -- 8; 8 -- 9
        9 -- 10; 9 -- 7; 10 -- 7
        6 -- 7
}
```

Actually, we've added line separators (';') to the code to reduce its length.

It's sometimes useful to retain this file (i.e. by removing the `cleanup=True` argument from render()) since it can be employed in other Graphviz tools, such as the 'Make a Graph' website at `https://graphs.grevian.org/graph`.

One easy change to Listing H.1 is to set the direction of the layout to be left-to-right by uncommenting the line:

```
g.attr(rankdir='LR')
```

Rerunning the program produces Fig. H.2.

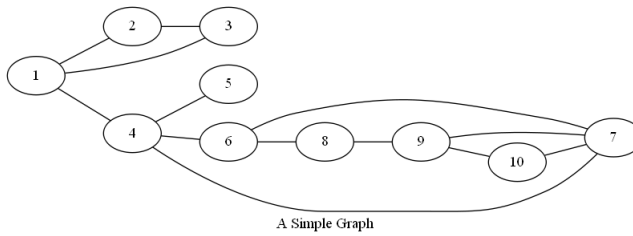


Figure H.2. The Left-to-Right Undirected Graph

'rankdir' is one of *many* graph attributes, which are listed in full at `https://graphviz.org/docs/graph/`.

H.1.1 The Directed Version. Converting the undirected graph to a directed version is another one-line change to Listing H.1 – replacing `graphviz.Graph()` by `graphvis.Digraph()`:

```
g = graphviz.Digraph(format='png')
```

Fig. H.3 shows both the top-down and the left-right versions of this directed graph.

H.2 Using Nodes

It's often best to explicitly define graph nodes by calling `node()` since this gives us access to a node's attributes such as its shape, color, and labeling. Listing H.2 (tennis.py) is an example.

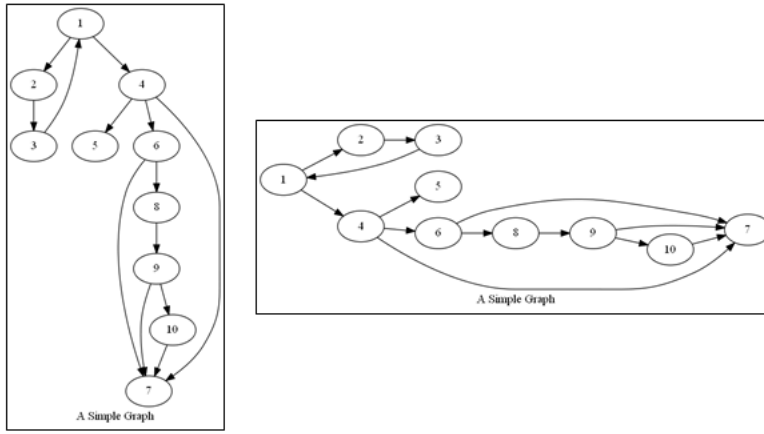


Figure H.3. Top-Bottom and Left-Right Directed Graphs

```
import graphviz as gv

TEMP_FNM = 'temp_graph'

t = gv.Digraph(format='png')
t.attr(label='Tennis Tournament')
t.attr(rankdir='LR')
t.attr('node', shape='box')

# player nodes
t.node('Player 1',shape='none'); t.node('Player 2',shape='none')
t.node('Player 3',shape='none'); t.node('Player 4',shape='none')
t.node('Player 5',shape='none'); t.node('Player 6',shape='none')
t.node('Player 7',shape='none'); t.node('Player 8',shape='none')

# game nodes
t.node('Game 1'); t.node('Game 2'); t.node('Game 3')
t.node('Game 4'); t.node('Game 5'); t.node('Game 6')
t.node('Game 7')

# link players and games
t.edge('Player 1','Game 1'); t.edge('Player 2','Game 1')
t.edge('Player 3','Game 2'); t.edge('Player 4','Game 2')
t.edge('Player 5','Game 3'); t.edge('Player 6','Game 3')
t.edge('Player 7','Game 4'); t.edge('Player 8','Game 4')
t.edge('Game 1', 'Game 5'); t.edge('Game 2', 'Game 5')
t.edge('Game 3', 'Game 6'); t.edge('Game 4', 'Game 6')
t.edge('Game 5', 'Game 7'); t.edge('Game 6', 'Game 7')
```

```
t.render(filename=TEMP_FNM, view=True, cleanup=True)
```

Listing H.2. Drawing a Tennis Tournament

This program displays the details of the players and games in a tennis tournament in Fig. H.4. Note that the list of players is slightly out of order.

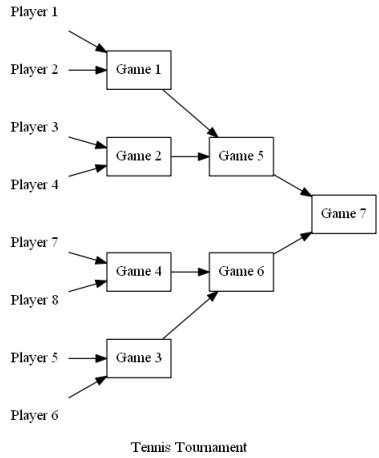


Figure H.4. A Slightly Misordered Tennis Tournament

Node attributes can be set globally by calling the attr() function with a 'node' argument. In `tennis.py`, all the nodes are set to be box-shaped with:

```
t.attr('node', shape='box')
```

A complete list of node attributes can be found at `https://graphviz.org/docs/nodes/`.

The player strings on the left of Fig. H.4 are implemented as nodes with no shape:

```
t.node('Player 1', shape='none')
```

Fixing the ordering of the labels so they are in increasing order going down the page illustrates the trickiness of overriding Graphviz's layout engines.

The most commonly used technique is to link nodes with *invisible* edges to encourage the engine to position the nodes in a particular order. Unfortunately, this isn't sufficient in this case since we're utilizing a left-to-right ordering for the graph, but the players should be positioned top-down. This distinction can be made by placing the invisible edges in a subgraph with their own rank and direction. The new lines in tennis2.py are:

```
with t.subgraph() as ps:
  ps.attr(rankdir='TB', rank='same')
  ps.edge('Player 1', 'Player 2', style='invis')
  ps.edge('Player 2', 'Player 3', style='invis')
  ps.edge('Player 3', 'Player 4', style='invis')
  ps.edge('Player 5', 'Player 6', style='invis')
  ps.edge('Player 6', 'Player 7', style='invis')
  ps.edge('Player 7', 'Player 8', style='invis')
```

The ps subgraph is inside the tournament's graph (t), and the edges all have the same rank in top-to-bottom (rankdir='TB') order. Running tennis2.py produces Fig. H.5.

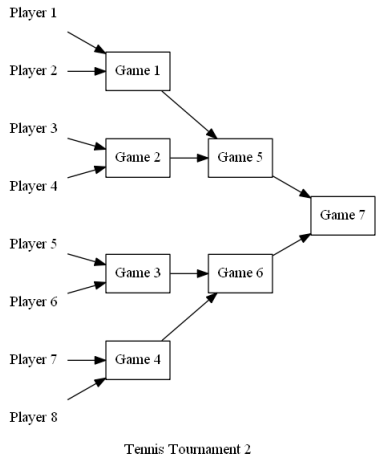


Figure H.5. An Ordered Tennis Tournament

H.2.1 A Probability Tree. The probability tree in Fig. H.6 shows a few other aspects of node rendering.

probs.py generates a top-down directed graph with circular nodes, but the ones at level 2 are colored blue. Less obviously, several nodes use the same labels ('W', 'R', and 'G') which require their node names (which must be unique) to be different from their labels. This is done using code like:

```
tree.node(pNode, label=col, shape='circle', color='blue')
```

pNode is a unique string generated using a counter at run time, while col is one of the three letters.

The probabilities beneath each leaf node in Fig. H.6 are drawn as no-shape nodes connected to the leaves above by invisible edges. This is implemented in probs.py using:

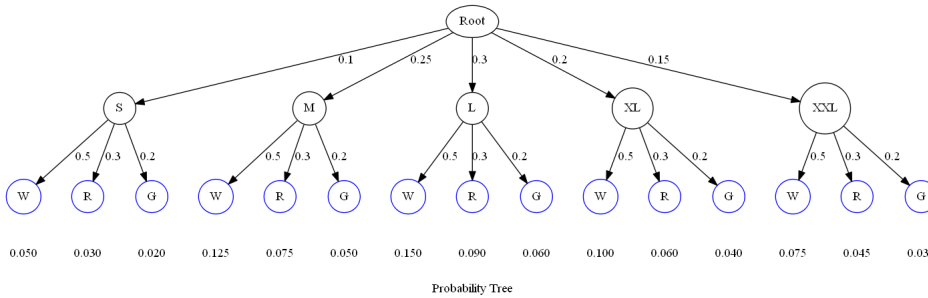


Figure H.6. A Probability Tree

```
# total prob in 'none' shape linked by invisible edge to pNode
tree.node(tNode, label=f"{(sprob*cprob):.3f}",
                    labelloc='t', shape='none')
tree.edge(pNode, tNode, len='0.5', weight='3', style='invis')
                  # shorter and more vertical
```

The invisible edges are shortened using the `len` attribute, and their weight is increased with `weight`. The weight makes it more likely that the probability text will be directly below the letter nodes rather than be spaced out evenly across the row. A complete list of edge attributes can be found at `https://graphviz.org/docs/edges/`.

H.3 More Layouts

Graphviz has a growing list of layout engines (see `https://graphviz.org/docs/layouts/`). `dot` is the default one, best suited to hierarchical drawing such as the trees in the previous section. More general graphs may benefit from being laid out with `neato`, `fdp`, or `sfdp`. Graphs arranged in a circle are best positioned with `circo`.

engines.py uses the same edge data as simpGraph.py, but lets the user decide on the layout engine for drawing it. Fig. H.7 shows the images produced when the user enters `dot`, `neato`, `fdp`, `sfdp`, or `circo`.

The layout managers can be adjusted using various graph attributes (see `https://graphviz.org/docs/graph/`). engines.py uses:

```
g.attr(rankdir='LR')      # for dot
g.attr(overlap='false')   # avoids node overlapping
g.attr(splines='true')    # helps edges avoid crossing nodes
g.attr(oneblock='true')   # for circo
```

The `splines` attribute lets the layout manager employ splines to curve edges around nodes; this avoids overlapping but can slow down rendering when the

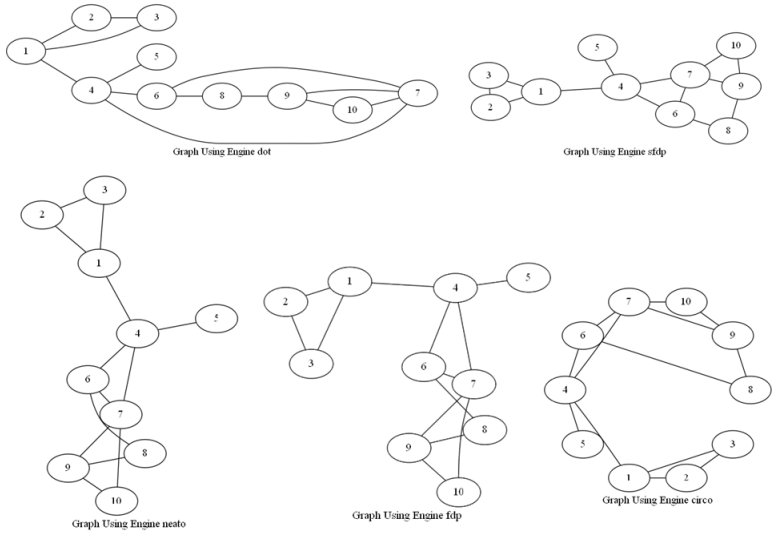


Figure H.7. The Same Graph; Five Different Layouts.
Top row: `dot`, `sfdp`; Bottom row: `neato`, `fdp`, `circo`

graph is large. `oneblock` tells `circo` that all the nodes are to be grouped into a single circle.

H.4 More Attributes

Listing H.3 (arcs.py) draws the directed graph in Fig. H.8, which illustrates a few more node and edge attributes.

```
import graphviz

def ns(name, val):
  return '<<I>'+name+'</I><SUB>'+str(val)+'</SUB>>'

TEMP_FNM = 'temp_graph'
ALPHA_BLUE = '#0096FF80'  # 50% alpha

dg = graphviz.Digraph(format='png', engine='neato',
      node_attr={'shape':'point'},
      edge_attr={'fontsize':'12',
                 'arrowhead':'open',
                 # 'color':'cadetblue3'
                 'color':ALPHA_BLUE
                })
dg.attr(label='Directed Graph')
```

```
for v in ['v1','v2','v3','v4','v5','v6','v7']:
  dg.node(v, '', xlabel=v)

dg.edge('v1','v2',label=ns('e',1))
dg.edge('v2','v3',label=ns('e',2))
dg.edge('v3','v4',label=ns('e',3))
dg.edge('v4','v6',label=ns('e',4))
dg.edge('v6','v5',label=ns('e',6))
dg.edge('v6','v7',label=ns('e',7))
dg.edge('v6','v1',label=ns('e',8))
dg.edge('v2','v7',label=ns('e',9))
dg.edge('v3','v5',label=ns('e',5))

dg.render(filename=TEMP_FNM, view=True, cleanup=True)
```

Listing H.3. Generate a Fancy Directed Graph

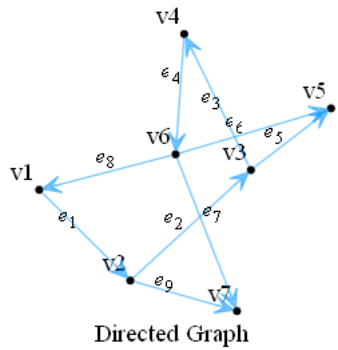


Figure H.8. Directed Graph Using Attributes

The node and edge attributes are set with the `node_attr` and `edge_attr` parameters of Digraph(), which allow multiple attributes to be set at once via a key:value map; this requires less code than calling `dg.attr('node', ...)` and `dg.attr('edge', ...)` multiple times. There's also a `graph_attr` parameter which isn't used in this example.

The API structure for Digraph() (and Graph()) are described in the Graphviz Python documentation at `https://graphviz.readthedocs.io/en/stable/api.html`. However, the meaning of the attributes, such as the possible node shapes, is explained in the Graphviz package documentation (e.g. at `https://graphviz.org/doc/info/shapes.html`).

The nodes are drawn by arcs.py as small black points, so their labels must be placed along side with the `xlabel` attribute (`https://graphviz.org/docs/nodes/`).

The edge labels use a smaller font size and subscripts. The latter is best archived by writing the label using a snippet of HTML, which is done by ns() in arcs.py. In fact, the HTML is more accurately 'HTML-like' since a lot of Graphviz-specific elements have been added; the details can be found at `https://graphviz.org/doc/info/shapes.html#html`. Another major use of this feature is to subdivide box node shapes with horizontal and vertical lines to organize record or tabular data (see the 'More Shapes' section below for an example).

The edge colors are fixed in two ways – using a preexisting color (`cadetblue3`) which is commented out, or with a hexadecimal consisting of values for the RGB and alpha components. Color names are listed at `https://graphviz.org/doc/info/colors.html`, while various color schemes are detailed at `https://graphviz.org/docs/attr-types/color/`. Our `ALPHA_BLUE` sets the alpha part of the edges to 0x80, which is roughly 50% transparent, making it a little easier to read the node labels.

H.5 Clusters: An Automaton

A cluster is a subgraph with a visual appearance (by default, a rectangle), and support for many drawing attributes. A cluster is created in the same way as a subgraph (i.e. with subgraph()) but must have a name that begins with the string 'cluster'.

Listing H.4 (automaton.py) draws the automaton in Fig. H.9 which includes a cluster holding the 's1' and 's2' nodes.

```
import graphviz

TEMP_FNM = 'temp_graph'

dg = graphviz.Digraph(format='png')
dg.attr(rankdir='LR')
dg.attr(label='An Automaton')
dg.attr('edge', fontcolor='blue')

# start arrow with no node
dg.node('', shape='none')
dg.edge('', 's0', label='Start')

# nodes
dg.node('s0', shape='doublecircle')
dg.node('s3', shape='doublecircle')

with dg.subgraph(name='cluster1') as c1:
  c1.attr(label='Internal', style='rounded,dashed')
  c1.node('s1')   # nodes inside a cluster
  c1.node('s2')
```

```
# transitions
dg.edge('s0','s0',label='0'); dg.edge('s0','s1',label='1')
dg.edge('s1','s2',label='1'); dg.edge('s2','s3',label='0')
dg.edge('s3','s3',label='0, 1')
dg.edge('s2','s2',label='1'); dg.edge('s1','s3',label='0')

dg.render(filename=TEMP_FNM, view=True, cleanup=True)
```

Listing H.4. Drawing an Automaton

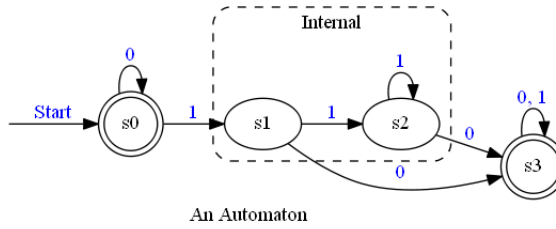


Figure H.9. An Automaton with a Cluster

The cluster code only utilizes the `label` and `style` attributes, but many other elements can be adjusted, as listed at `https://graphviz.org/docs/clusters/`.

Note that although the 's1' and 's2' nodes are defined inside the cluster subgraph `c1`, the edge code only employs their string names.

H.5.1 More Clusters: a Statechart.

`states.py` draws the nested statechart shown in Fig. H.10.

The outer cluster `c` is created with:

```
with dg.subgraph(name='cluster_op') as c:
  c.attr(label='Operational', style='rounded')
    : # create nodes, an inner cluster, and edges
```

The inner cluster `p` is created with:

```
  with c.subgraph(name='cluster_pul') as p:
    p.attr(label='Pulsing', style='rounded')
      :  # define three nodes and edges
```

Note that `p` is defined as a subgraph of the outer cluster `c`.

The only completely new feature here are the two dashed arrows that link the inner cluster to the 'Standby' node; they are implemented using:

```
  # edges to/from cluster p
  c.edge('standby', 'pulsingStart',
```

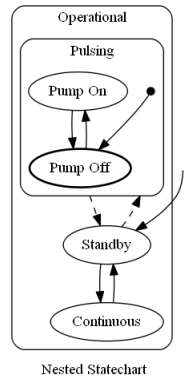


Figure H.10. A Nested Statechart

```
                lhead='cluster_pul', style='dashed')
  c.edge('pOff', 'standby',
                ltail='cluster_pul', style='dashed')
```

The calls to `edge()` link the arrows to the inner cluster using its string name 'cluster_pul' not the cluster variable `p`.

H.6 More Shapes

Listing H.5 (`someShapes.py`) draws a variety of node shapes in Fig. H.11.

```
import graphviz

TEMP_FNM = 'temp_graph'

dg = graphviz.Digraph(format='png')
dg.attr(rankdir='LR')   # LR or TB
dg.attr(label='Some Shapes')

dg.node('n1', '{record|{a|b|<f0>c}}', shape='record')
   # uses graph's rankdir to interpret '|'

dg.node('n2', label=(
  '''<<table border="0" cellborder="1" cellspacing="0">
      <tr>
        <td colspan="3">table</td>
      </tr>
      <tr>
        <td port="f0">d</td>
        <td port="f1">e</td>
        <td port="f2">f</td>
      </tr>
```

```
    </table>
  >''' ), shape='plaintext')

dg.edge('n1:f0', 'n2:f2')

# pentagon
dg.node('P', '5-sided', shape='polygon', sides='5')
# hexagon
dg.node('H', '6-sided', shape='polygon', sides='6')
# distorted hexagon
dg.node('E','Distorted',shape='polygon',sides='6',distortion='0.7')
# skewed hexagon
dg.node('F', 'Skewed', shape='polygon', sides='6', skew='0.7')
# rotated hexagon
dg.node('G', 'Rot 30',shape='polygon',sides='6',orientation='30')
dg.edges(['PH', 'HE', 'EF', 'FG'])
   # e.g. PH means link nodes P --> H

dg.render(filename=TEMP_FNM, view=True, cleanup=True)
```

Listing H.5. Generating Different Node Shapes

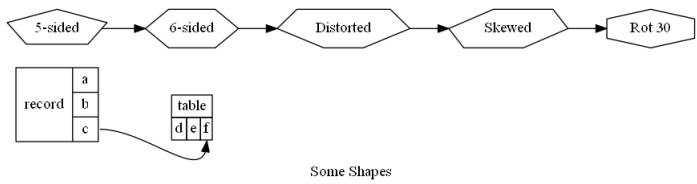


Figure H.11. Different Node Shapes

The bottom row of Fig. H.11 is generated first by the code, using a node record and HTML. (More complex examples of these two approaches can be found at `https://graphviz.readthedocs.io/en/stable/examples.html`). Note that the edge between the nodes employs `f` fields in the labels.

Although the record coding is considerably shorter, the meaning of its '|' separator is tied to the layout direction (in this case left-to-right, `LR`). If the layout is changed to top-to-bottom (`TB`), the HTML node isn't affected, but the record's organization changes as shown in Fig. H.12.

The top row in Fig. H.11 utilizes a 'polygon' shape combined with various attributes. The first shape is 5-sided, but the others are hexagons. Details on the many shape types and polygon attributes are given at `https://graphviz.org/doc/info/shapes.html`.

H.6.1 Images: A Circuit Diagram. Probably the easiest way to utilize an arbitrary shape as a node is to save its outline as an image. `circuits.py` loads the three PNG images shown in Fig. H.13.

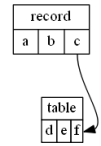


Figure H.12. A top-bottom Record and HTML Node



Figure H.13. AND, OR, and NOT Gate Images

The relevant snippets of code are:

```
# gates using images for AND, OR, and NOT
g.node('AND', image='andGate.png',imagescale='true',shape='none')
g.node('OR', image='orGate.png',imagescale='true',shape='none')
g.node('NOT\l', image='notGate.png',imagescale='true',shape='none')
```

The default shape (an ellipse) is switched off, and the loaded image is scaled to fit its space.

A drawback is that edges only extend to the default shape's boundary, so are unlikely to touch the image, as shown in Fig. H.14.

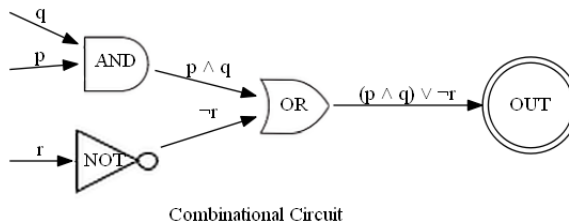


Figure H.14. A Combinatorial Circuit

Another occasional issue is positioning a node's label on top of an image. For instance, we had to left-justify the 'NOT' string using `\l`.

Appendix I

Matrices

The purpose of this appendix is to briefly review matrices in sufficient detail to cover their use in other parts of the book such as in computer graphics, computational geometry, and Markov chains. This is very much *not* a comprehensive discussion of the subject. For that we suggest most textbooks on linear algebra or engineering maths, which will at least have a chapter or two about matrices; a good single text is Richard Bronson's *Schaum's Outline Of Theory And Problems Of Matrix Operations* [**Bro88**], which can be found in the Internet Archive at `https://archive.org/details/schaum-s-outlines-matrix-operations`.

A second aim is to introduce Mat.py, a simple class for manipulating matrices. We've chosen this approach rather than rely on Numpy matrices so that its easier to understand the algorithms used, especially for obtaining a determinant or a matrix inverse. Numpy is vastly superior in terms of functionality and efficiency, but these advantages are less important for the small matrices used here.

Mat.py is copiously documented, and comes with an extensive test-rig that shows off its features. To try it out, download the code, and run `python Mat.py`.

I.1 What is a Matrix?

A *matrix* is a set of real or complex numbers arranged in rows and columns to form a rectangular array. A matrix having m rows and n columns is called an $m \times n$ matrix and is referred to as having *order* $m \times n$.

A matrix is indicated by writing the array within brackets. For example:

$$\begin{pmatrix} 5 & 7 & 2 \\ 6 & 3 & 8 \end{pmatrix}$$

is a 2×3 matrix. Note that the number of rows is stated first and the number of columns second.

A whole matrix can be denoted by a single letter printed in bold type. For example, let

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}.$$

Similarly, the vextor $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$.

With Mat.py, you can create a matrix either by supplying a 2D list of values or by reading the data from a text file, such as m1.txt:

```
>>> from Mat import Mat
>>> a = Mat([[1,2,3],[4,5,6],[7,8,9]])
>>> print(a)
 1.00    2.00    3.00
 4.00    5.00    6.00
 7.00    8.00    9.00

>>> b = Mat.readSq('m1.txt')
Reading square: m1.txt
>>> print(b)
 0.80    0.19    0.01
 0.20    0.70    0.10
 0.10    0.20    0.70
```

I.2 Equality

Two matrices are said to be equal if corresponding elements throughout are equal. Thus, the two matrices must also be of the same order. So, if

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} 4 & 6 & 5 \\ 2 & 3 & 7 \end{pmatrix}$$

then $a_{11} = 4; a_{12} = 6; a_{13} = 5; a_{21} = 2$; etc.

I.3 Addition and Subtraction

To be added or subtracted, two matrices must be of the same order. The sum or difference is then determined by adding or subtracting corresponding elements. For example:

$$\begin{pmatrix} 4 & 2 & 3 \\ 5 & 7 & 6 \end{pmatrix} + \begin{pmatrix} 1 & 8 & 9 \\ 3 & 5 & 4 \end{pmatrix} = \begin{pmatrix} 4+1 & 2+8 & 3+9 \\ 5+3 & 7+5 & 6+4 \end{pmatrix} = \begin{pmatrix} 5 & 10 & 12 \\ 8 & 12 & 10 \end{pmatrix}$$

Naturally, Mat.py offers these operations, both as methods called add() and subtract(), and through operator overloading:

```
>>> print(a + b)
 1.80    2.19    3.01
 4.20    5.70    6.10
 7.10    8.20    9.70

>>> print(a - b)
 0.20    1.81    2.99
 3.80    4.30    5.90
 6.90    7.80    8.30
```

I.4 Multiplication

I.4.1 Scalar multiplication. To multiply a matrix by a single number, each individual element of the matrix is multiplied by that value. For example,

$$4 \cdot \begin{pmatrix} 3 & 2 & 5 \\ 6 & 1 & 7 \end{pmatrix} = \begin{pmatrix} 12 & 8 & 20 \\ 24 & 4 & 28 \end{pmatrix}$$

Mat.py offers scalar multiplication through operator overloading:

```
>>> print(5*a)
 5.00    10.00   15.00
 20.00   25.00   30.00
 35.00   40.00   45.00

>>> print(b*2)
 1.60    0.38    0.02
 0.40    1.40    0.20
 0.20    0.40    1.40
```

I.4.2 Multiplication of two matrices. Two matrices can be multiplied together only when the number of columns in the first is equal to the number of rows in the second. For example, if

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

$$\begin{aligned} \text{then } \mathbf{A} \cdot \mathbf{b} &= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \\ &= \begin{pmatrix} a_{11}b_1 + a_{12}b_2 + a_{13}b_3 \\ a_{21}b_1 + a_{22}b_2 + a_{23}b_3 \end{pmatrix} \end{aligned}$$

Each element in the top *row* of **A** is multiplied by the corresponding element in the first *column* of **b** and the products added. Similarly, the second row of

the product is found by multiplying each element in the second row of **A** by the corresponding element in the first column of **b**.

In general, if **A** is an $m \times n$ matrix and **B** is an $n \times p$ matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

then the matrix product $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ is the $m \times p$ matrix

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

where

$$\begin{aligned} c_{ij} &= a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} \\ &= \textstyle\sum_{k=1}^{n} a_{ik}b_{kj}, \quad \text{for } i = 1, \ldots, m \text{ and } j = 1, \ldots, p. \\ &= \textstyle\sum_{i=1}^{m} \sum_{j=1}^{p} \sum_{k=1}^{n} a_{ik}b_{kj} \end{aligned}$$

An example:

$$\begin{pmatrix} 4 & 7 & 6 \\ 2 & 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} 8 \\ 5 \\ 9 \end{pmatrix} = \begin{pmatrix} 4 \times 8 + 7 \times 5 + 6 \times 9 \\ 2 \times 8 + 3 \times 5 + 1 \times 9 \end{pmatrix} = \begin{pmatrix} 32 + 45 + 54 \\ 16 + 15 + 9 \end{pmatrix} = \begin{pmatrix} 121 \\ 40 \end{pmatrix}$$

Multiplying this (2×3) matrix and a (3×1) matrix gives a product of order (2×1).

Mat.py has a multiply() method, and also overrides the "*" and "@" operators.

```
>>> print(a*b)
 1.50    2.19    2.31
 4.80    5.46    4.74
 8.10    8.73    7.17

>>> print(b@a)
 1.63    2.63    3.63
 3.70    4.70    5.70
 5.80    6.80    7.80

>>> c = Mat([[1,2],[3,4]])
>>> print(a*c)
ValueError: Wrong dimensions for multiplication.
```

The second example illustrates the non-commutativity of matrix multiplication, and the third example shows the error raised when the matrix sizes do not match.

The code that implements matrix multiplication is very similar to the maths:

```
def multiply(self, other):
  if self.nCols != other.nRows:
      raise ValueError("Wrong dimensions for multiplication.")

  # Initialize the result matrix with zeros
  result = [[0 for _ in range(other.nCols)]
                       for _ in range(self.nRows)]

  # Explicit loops for matrix multiplication
  for i in range(self.nRows):
    for j in range(other.nCols):
      sumVal = 0
      for k in range(self.nCols): # or other.nRows
        sumVal += self.data[i][k] * other.data[k][j]
      result[i][j] = sumVal
  return Mat(result)
```

There is a more 'pythonic' way to write this code, based around list comprehensions, which is commented out in Mat.py.

The three `for`-loops indicate that the running time is cubic and dependent on the sizes of the matrices. If the matrices are $n \times n$ square, then the time is $O(n^3)$. A lot of effort has gone into trying to reduce this due to multiplication's importance, and an algorithm due to Volker Strassen reduces it to $O(n^{\log_2 7})$, which is significant if the matrices are large. A good description can be found in Section 4.2 of Cormen *et. al*'s *Algorithms* [**CLRS22**], and there's a nice visualization of the technique on the Wikipedia page (`https://en.wikipedia.org/wiki/Strassen_algorithm`).

I.5 Transpose of a Matrix

If the rows and columns of a matrix are interchanged, such that:

- the first row becomes the first column,
- the second row becomes the second column,
- the third row becomes the third column, etc.,

then the new matrix is called the *transpose* of the original. If $\mathbf{A}$ is the original, its transpose is denoted by $\tilde{\mathbf{A}}$ or $\mathbf{A}^T$. We shall use the latter. For example, if

$$\mathbf{A} = \begin{pmatrix} 4 & 6 \\ 7 & 9 \\ 2 & 5 \end{pmatrix}$$

then

$$\mathbf{A}^T = \begin{pmatrix} 4 & 7 & 2 \\ 6 & 9 & 5 \end{pmatrix}$$

I.6 Special Matrices

(a) A *square* matrix is a matrix of order $n \times n$. For example:

$$\begin{pmatrix} 1 & 2 & 5 \\ 6 & 8 & 9 \\ 1 & 7 & 4 \end{pmatrix}$$

is a 3×3 matrix.

A square matrix $\mathbf{A}$ is *symmetric* if $a_{ij} = a_{ji}$, e.g.

$$\begin{pmatrix} 1 & 2 & 5 \\ 2 & 8 & 9 \\ 5 & 9 & 4 \end{pmatrix}$$

is symmetrical about the leading diagonal. Note that $\mathbf{A} = \mathbf{A}^T$.

A square matrix $\mathbf{A}$ is *skew-symmetric* if $a_{ij} = -a_{ji}$, e.g.

$$\begin{pmatrix} 0 & 2 & 5 \\ -2 & 0 & 9 \\ -5 & -9 & 0 \end{pmatrix}$$

In that case, $\mathbf{A} = -\mathbf{A}^T$.

(b) A *diagonal* matrix is a square matrix with all elements zero except those on the leading diagonal, thus

$$\begin{pmatrix} 5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 7 \end{pmatrix}$$

(c) A *unit* (or *identity*) matrix is a diagonal matrix in which the elements on the leading diagonal are all unity, i.e.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The unit matrix is denoted by $\mathbf{I}_n$, where n is its size.

(d) A *null* matrix is one whose elements are all zeros, i.e.

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

and is denoted by $\mathbf{0}_n$, where n is its size

I.7 Determinant of a Square Matrix

The determinant $\det \mathbf{A}$ of a $n \times n$ matrix $\mathbf{A}$ is calculated as follows:

(1) if $n = 1$ then $A = |a|$ and $\det A = a$

(2) if $n \geq 2$ then

$$\det \mathbf{A} = \sum_{i=1}^{n} a_{ki}(-1)^{i+k} \det(\mathbf{A}'_{ik})$$

where $\mathbf{A}'_{ik}$ is the *submatrix* of $\mathbf{A}$ with the ith row and kth column deleted. The choice of k doesn't matter because the result will be the same.

This approach, known as Laplace's method after Pierre-Simon Laplace, systematically reduces a determinant of order n to smaller determinants of order $n-1$.

The determinant is often written as

$$\det \mathbf{A} = \begin{vmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & a_{22} & \cdots \\ \vdots & & \end{vmatrix}$$

For a 2×2 matrix, the Laplace equation can be replaced by

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

and for a 3×3 matrix,

$$\begin{vmatrix} A & B & C \\ D & E & F \\ G & H & I \end{vmatrix} = A \begin{vmatrix} E & F \\ H & I \end{vmatrix} - B \begin{vmatrix} D & F \\ G & I \end{vmatrix} + C \begin{vmatrix} D & E \\ G & H \end{vmatrix}$$

A square matrix $\mathbf{A}$ is said to be *singular* if $\det \mathbf{A} = 0$, and non-singular if $\det \mathbf{A} \neq 0$.

The determinant of the following 3×3 matrix is calculated with Laplace's method:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

We chose the first column, $k = 1$, to generate the three submatrices in Fig. I.1.

The result:

$$\begin{aligned} \det(\mathbf{A}) &= (-1)^{1+1}\, 1\, [(5 \cdot 9) - (6 \cdot 8)] \qquad i = 1,\ k = 1 \\ &\quad + (-1)^{2+1}\, 4\, [(2 \cdot 9) - (3 \cdot 8)] \quad i = 2,\ k = 1 \\ &\quad + (-1)^{3+1}\, 7\, [(2 \cdot 6) - (3 \cdot 5)] \quad i = 3,\ k = 1 \\ &= 1(-3) - 4(-6) + 7(-3) \\ &= 0 \end{aligned}$$

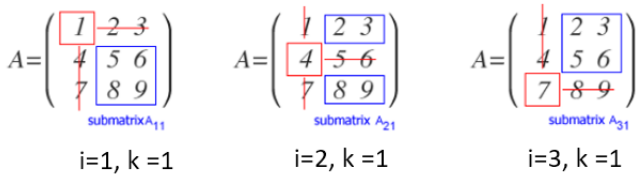


Figure I.1. The Laplace Method Applied to a 3×3 Matrix

I.7.1 Implementation. The implementation of the Laplace method is very close to the maths:

```
def determinant(self):
  if not self.isSquare():
    raise ValueError("Only square matrices have a determinant")

  n = self.nRows
  # base cases
  if n == 1:
    return self.data[0][0]
  if n == 2:
    return self.data[0][0] * self.data[1][1] - \
           self.data[0][1] * self.data[1][0]

  k = 0   # delete kth column
  det = 0
  for i in range(n):
    minor = self.submatrixExcept([i], [k])
    det += self.data[k][i] * (-1)**(i+k) * minor.determinant()
  return det
```

An example using this function:

```
>>> print(a)
 1.00    2.00    3.00
 4.00    5.00    6.00
 7.00    8.00    9.00

>>> print(a.determinant())
0
```

Its big drawback is its running time, $O(n!)$. Fortunately, approaches based on Gaussian elimination (see below) can reduce this to $O(n^3)$. Mat.py includes both versions, but the Gaussian-inspired code is commented out.

I.8 Inverse of a Square Matrix

An $n \times n$ matrix $\mathbf{A}$ is said to be *invertible* if there exists a matrix $\mathbf{A}^{-1}$, called the inverse of $\mathbf{A}$, such that

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}_n$$

One key result from linear algebra is:

$$\mathbf{A} \text{ is invertible } \Leftrightarrow \det(\mathbf{A}) \neq 0$$

We'll prove this by considering the contrapositive:

$$\text{if } \det(\mathbf{A}) = 0, \text{ then } \mathbf{A} \text{ is not invertible}$$

Assume:

$$\det(\mathbf{A}) = 0$$

Now suppose that $\mathbf{A}$ *is* invertible. Then, by definition, there exists $\mathbf{A}^{-1}$ such that:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}_n$$

Take the determinant of both sides:

$$\det(\mathbf{A}\mathbf{A}^{-1}) = \det(\mathbf{I}_n)$$

Using the property of determinants that $\det(\mathbf{A}\mathbf{B}) = \det(\mathbf{A})\det(\mathbf{B})$ (Binet's theorem), then

$$\det(\mathbf{A})\det(\mathbf{A}^{-1}) = \det(\mathbf{I}_n) = 1$$

But we assumed $\det(\mathbf{A}) = 0$, so:

$$0 \cdot \det(\mathbf{A}^{-1}) = 0 \neq 1$$

This is a contradiction. Therefore, our assumption that $\mathbf{A}$ is invertible must be false. Therefore, our contrapositive is true:

$$\text{if } \det(\mathbf{A}) = 0, \text{ then } \mathbf{A} \text{ is not invertible}$$

which means that the positive version is also true:

$$\mathbf{A} \text{ is invertible } \Leftrightarrow \det(\mathbf{A}) \neq 0$$

I.9 Solution of a Set of Linear Equations

Consider the set of linear equations:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \qquad\qquad \vdots \qquad &= \vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

They can be written in matrix form:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

i.e. $\mathbf{Ax} = \mathbf{b}$ where

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix}; \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}; \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

If we multiply both sides of the matrix equation by the inverse of $\mathbf{A}$, we have:

$$\mathbf{A^{-1}Ax} = \mathbf{A^{-1}b}$$

But

$$\mathbf{A^{-1}A} = \mathbf{I}_n$$

so

$$\begin{aligned} \mathbf{I}_n\mathbf{x} &= \mathbf{A^{-1}b} \\ \mathbf{x} &= \mathbf{A^{-1}b} \end{aligned}$$

Therefore, if we calculate the inverse of the matrix of coefficients and pre-multiply matrix $\mathbf{b}$ by it, we'll get a matrix of solutions in $\mathbf{x}$.

I.10 Gaussian Elimination

Gaussian elimination solves a linear system $\mathbf{Ax} = \mathbf{b}$ by transforming it into an equivalent system $\mathbf{Ux} = \mathbf{c}$ with an upper triangular matrix $\mathbf{U}$ (all entries in $\mathbf{U}$ below the diagonal are zero). This is done by applying three types of transformations to the *augmented* matrix $[\mathbf{A} \mid \mathbf{b}]$.

(1) Interchange two equations;

(2) Multiply an equation by some non-zero constant;

(3) Replace an equation with the sum of the same equation and a multiple of another equation.

Once the augmented matrix $[\mathbf{A} \mid \mathbf{b}]$ is transformed into $[\mathbf{U} \mid \mathbf{c}]$, we can solve $\mathbf{Ux} = \mathbf{c}$ using back-substitution.

I.10.1 Example. Suppose that we want to solve

$$\begin{pmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 8 \\ 10 \end{pmatrix}.$$

To keep track of the operations, we'll write them in the following form, $R_2 = R_2 - 2 * R_1$, which means that a new row 2 is computed by subtracting 2 times row 1 from the current row 2.

$$\left(\begin{array}{ccc|c} 2 & 4 & -2 & 2 \\ 4 & 9 & -3 & 8 \\ -2 & -3 & 7 & 10 \end{array}\right) \rightarrow \left(\begin{array}{ccc|c} 2 & 4 & -2 & 2 \\ 0 & 1 & 1 & 4 \\ 0 & 1 & 5 & 12 \end{array}\right) \quad \begin{array}{l} R_2 = R_2 - 2 * R_1 \\ R_3 = R_3 + R_1 \end{array}$$

$$\rightarrow \left(\begin{array}{ccc|c} 2 & 4 & -2 & 2 \\ 0 & 1 & 1 & 4 \\ 0 & 0 & 4 & 8 \end{array}\right) \quad R_3 = R_3 - R_2$$

In other words, the original system is equivalent to:

$$\begin{pmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 8 \end{pmatrix}.$$

This can be solved by back-substitution, producing

$$\begin{aligned} x_3 &= \tfrac{8}{4} = 2 \\ x_2 &= \tfrac{4-1*2}{1} = 2 \\ x_1 &= \tfrac{2-4*2-(-2)*2}{2} = -1 \end{aligned}$$

I.11 Matrix Inversion Using Gaussian Elimination

An $n \times n$ matrix $\mathbf{A}$ can have an inverse only if Gaussian elimination produces a matrix with all diagonal elements equal to 1. For $n = 3$ this looks like:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \xrightarrow{\text{Gaussian elimination}} \begin{bmatrix} 1 & ? & ? \\ 0 & 1 & ? \\ 0 & 0 & 1 \end{bmatrix}$$

Now we can keep going by applying elementary row operations to the triangle in the upper half until we get $\mathbf{I}_n$:

$$\begin{bmatrix} 1 & ? & ? \\ 0 & 1 & ? \\ 0 & 0 & 1 \end{bmatrix} \xrightarrow{\text{Row operations}} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

We carry out this version of Gaussian elimination by forming an $n \times 2n$ matrix, $\mathbf{C}$, from $[\mathbf{A} \mid \mathbf{I}_n]$:

$$\left[\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{array}\right]$$

If the left half of the transformed matrix has a upper-triangular form with a zero on its diagonal, then $\mathbf{A}$ is not invertible. Otherwise keep going until the left half becomes $\mathbf{I}_n$. For $n = 3$ the result will look like:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & b_{11} & b_{12} & b_{13} \\ 0 & 1 & 0 & b_{21} & b_{22} & b_{23} \\ 0 & 0 & 1 & b_{31} & b_{32} & b_{33} \end{array}\right]$$

I.11.1 Example. Find the inverse of

$$\mathbf{A} = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0.5 & 0 & 0.5 \\ 0 & 0.5 & 0.5 \end{bmatrix}$$

Form a 3×6 matrix $\mathbf{C}$ and perform Gaussian elimination on it.

$$\left[\begin{array}{ccc|ccc} 0.5 & 0.5 & 0 & 1 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 & 1 & 0 \\ 0 & 0.5 & 0.5 & 0 & 0 & 1 \end{array}\right] \xrightarrow{R2=R2-R1} \left[\begin{array}{ccc|ccc} 0.5 & 0.5 & 0 & 1 & 0 & 0 \\ 0 & -0.5 & 0.5 & -1 & 1 & 0 \\ 0 & 0.5 & 0.5 & 0 & 0 & 1 \end{array}\right]$$

$$\xrightarrow{R3=R3+R2} \left[\begin{array}{ccc|ccc} 0.5 & 0.5 & 0 & 1 & 0 & 0 \\ 0 & -0.5 & 0.5 & -1 & 1 & 0 \\ 0 & 0 & 1 & -1 & 1 & 1 \end{array}\right]$$

$$\xrightarrow{R1=R1*2} \left[\begin{array}{ccc|ccc} 1 & 1 & 0 & 2 & 0 & 0 \\ 0 & -0.5 & 0.5 & -1 & 1 & 0 \\ 0 & 0 & 1 & -1 & 1 & 1 \end{array}\right]$$

$$\xrightarrow{R2=R2*-2} \left[\begin{array}{ccc|ccc} 1 & 1 & 0 & 2 & 0 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 0 & 1 & -1 & 1 & 1 \end{array}\right]$$

The left half is in upper-triangular form, but we still need to get rid of its nonzero off-diagonal elements:

$$\left[\begin{array}{ccc|ccc} 1 & 1 & 0 & 2 & 0 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 0 & 1 & -1 & 1 & 1 \end{array}\right] \xrightarrow{R2=R2+R3} \left[\begin{array}{ccc|ccc} 1 & 1 & 0 & 2 & 0 & 0 \\ 0 & 1 & 0 & 1 & -1 & 1 \\ 0 & 0 & 1 & -1 & 1 & 1 \end{array}\right]$$

$$\xrightarrow{R1=R1-R2} \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 1 & -1 \\ 0 & 1 & 0 & 1 & -1 & 1 \\ 0 & 0 & 1 & -1 & 1 & 1 \end{array}\right]$$

You should check that

$$\left[\begin{array}{ccc} 1 & 1 & -1 \\ 1 & -1 & 1 \\ -1 & 1 & 1 \end{array}\right]$$

is indeed $\mathbf{A}^{-1}$.

I.11.2 Why Does Gaussian Elimination Work? The initial augmented matrix $[\mathbf{A} \mid \mathbf{I}_n]$ equals the matrix $\mathbf{C}$. For example, for $n = 3$:

$$\left[\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{array}\right] = [\mathbf{A} \mid \mathbf{I}_n] = \mathbf{C}$$

The end result of the Gaussian elimination is the augmented matrix $[\mathbf{I}_n \mid \mathbf{B}]$:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & b_{11} & b_{12} & b_{13} \\ 0 & 1 & 0 & b_{21} & b_{22} & b_{23} \\ 0 & 0 & 1 & b_{31} & b_{32} & b_{33} \end{array}\right] = [\mathbf{I}_n \mid \mathbf{B}]$$

The procedure successively applies elementary row operations to the matrix $\mathbf{C}$. These can be implemented by successively multiplying $\mathbf{C}$ on the left by the elementary matrices $\mathbf{E}_1, \mathbf{E}_2, \mathbf{E}_3, ..., \mathbf{E}_k$:

$$\mathbf{E}_k \cdots \mathbf{E}_3\mathbf{E}_2\mathbf{E}_1\, \mathbf{C} \;=\; \mathbf{E}_k \cdots \mathbf{E}_3\mathbf{E}_2\mathbf{E}_1\, [\mathbf{A} \mid \mathbf{I}_n]$$

These operations work independently on the two matrices, $\mathbf{A}$ and $\mathbf{I}_n$, resulting in:

$$[\mathbf{E}_k \cdots \mathbf{E}_3\mathbf{E}_2\mathbf{E}_1\mathbf{A} \mid \mathbf{E}_k \cdots \mathbf{E}_3\mathbf{E}_2\mathbf{E}_1\mathbf{I}_n] = [\mathbf{I}_n \mid \mathbf{B}]$$

The second argument shows that:

$$\mathbf{B} = \mathbf{E}_k \cdots \mathbf{E}_3\mathbf{E}_2\mathbf{E}_1\mathbf{I}_n = \mathbf{E}_k \cdots \mathbf{E}_3\mathbf{E}_2\mathbf{E}_1.$$

The first argument shows that:

$$(\mathbf{E}_k \cdots \mathbf{E}_3\mathbf{E}_2\mathbf{E}_1)\, \mathbf{A} = \mathbf{B}\mathbf{A} = \mathbf{I}_n.$$

It follows that $\mathbf{B} = \mathbf{A}^{-1}$.

I.11.3 Implementation. The inverse() function in Mat.py starts by creating an augmented matrix with the object's data on the left and an identity matrix on the right. It then enters a loop which considers every column of the data looking for a row with the largest absolute value in column i. This approach is slightly different from the steps outlined above, but makes it less likely for the code to fall foul of rounding errors when using a small pivot. The chosen row is normalized and used to change the other rows, including setting their column i to 0. Once the loop ends, the inverse is extracted from the right half of the augmented matrix and returned.

```
def inverse(self):
  if not self.isSquare():
    raise ValueError("Only square matrices can be inverted")

  n = self.nRows
  # Create augmented matrix with self.data on the left
  # and the identity matrix on the right.
  aug = [self.data[i][:] + \
        [1 if i == j else 0 for j in range(n)]
                                    for i in range(n)]
  for i in range(n):
    # pivot row with largest absolute value in column i.
    pivotRowIdx = i
    maxVal = abs(aug[i][i])
    for r in range(i + 1, n):
      if abs(aug[r][i]) > maxVal:
        maxVal = abs(aug[r][i])
        pivotRowIdx = r

    if maxVal < EPS:
      raise ValueError("Pivot too small")

    if pivotRowIdx != i:
      aug[i], aug[pivotRowIdx] = aug[pivotRowIdx],aug[i]  # Swap rows

    pivot = aug[i][i]
    # Normalize the pivot row.
    for j in range(2 * n):
      aug[i][j] /= pivot

    # Eliminate the current column in all the other rows.
    for r in range(n):
      if r != i:
        factor = aug[r][i]
        for j in range(2 * n):
          aug[r][j] -= factor * aug[i][j]
```

```
    # Extract the right half as the inverse matrix.
    invData = [row[n:] for row in aug]
    return Mat(invData)
```

An example using this function:

```
>>> a = Mat([[0.5, 0.5, 0],[0.5,0,0.5],[0,0.5,0.5]])
>>> print(a)
 0.50    0.50    0.00
 0.50    0.00    0.50
 0.00    0.50    0.50

>>> b = a.inverse()
>>> print(b)
 1.00    1.00   -1.00
 1.00   -1.00    1.00
-1.00    1.00    1.00

>>> print(a*b)
 1.00    0.00    0.00
 0.00    1.00    0.00
 0.00    0.00    1.00
```

Appendix J

Complex Numbers and `cmath`

The square root of a negative number (e.g., $\sqrt{-1}$, $\sqrt{-5}$, $\sqrt{-9}$) is called an imaginary number, and it's convenient to introduce the symbol $i = \sqrt{-1}$ and adopt $\sqrt{-n} = i\sqrt{n}$ as its standard form (e.g. $\sqrt{-5} = i\sqrt{5}$). i has the property $i^2 = -1$, and for higher powers:

$$i^3 = i^2 \cdot i = (-1)i = -i, \quad i^4 = (i^2)^2 = (-1)^2 = 1, \quad i^5 = i^4 \cdot i = i, \text{ etc.}$$

$a+bi$, where a and b are reals, is called a complex number. The first term a is its real part, and the second term bi is the imaginary part. Two complex numbers, $a + bi$ and $c + di$, are equal if and only if $a = c$ and $b = d$.

The conjugate of $a + bi$ is $a - bi$. Multiplying a number to its conjugate produces a real number, the square of its magnitude:

$$(a + bi)(a - bi) = a^2 + b^2$$

The basic algebraic operations:

- **Addition**. Separately add the real parts and imaginary parts. For example,

$$(2 + 3i) + (4 - 5i) = (2 + 4) + (3 - 5)i = 6 - 2i.$$

- **Subtraction**. Separately subtract the real parts and imaginary parts. For example,

$$(2 + 3i) - (4 - 5i) = (2 - 4) + [3 - (-5)]i = -2 + 8i.$$

- **Multiplication**. Perform a multiplication as if each number was a tuple of two values, and replace i^2 by -1 in the result.

$$(a + bi)(c + di) = ac - bd + (ad + bc)i$$

For example,

$$(2+3i)(4-5i) = 8-15(-1)+(-10+12)i = 23+2i.$$

- **Division.** Multiply the fraction's numerator and denominator by the conjugate of the denominator. For example,

$$\frac{2+3i}{4-5i} = \frac{(2+3i)(4+5i)}{(4-5i)(4+5i)} = \frac{(8-15)+(10+12)i}{16+25} = -\frac{7}{41}+\frac{22}{41}i.$$

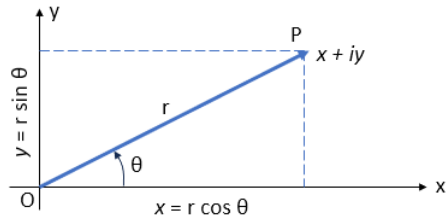


Figure J.1. A Complex Number Graphically

In Fig. J.1, $x+yi$ is represented by the point P at (x, y). It can also be encoded as the directed line segment $\overrightarrow{OP}$. This vector (and hence the complex number) can be described in terms of its length r and the angle θ it makes with the positive x-axis. $r = \sqrt{x^2+y^2}$ is the modulus (magnitude or absolute value) of the complex number, while θ is the phase (amplitude or `arg`), and usually the smallest, positive angle for which $\tan\theta = y/x$.

From Fig. J.1, we see that $x = r\cos\theta$ and $y = r\sin\theta$, so $z = x+yi = r\cos\theta + ir\sin\theta = r(\cos\theta + i\sin\theta)$. We call this the polar or trigonometric form, and $z = x+yi$ the rectangular form.

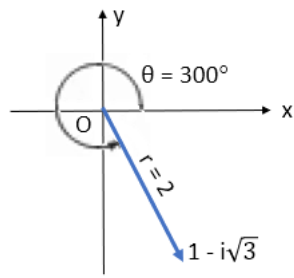


Figure J.2. $1-i\sqrt{3}$ in Polar Form

Consider $z = 1 - i\sqrt{3}$ in Fig. J.2. Its modulus $r = \sqrt{(1)^2 + (-\sqrt{3})^2} = 2$, and since $\tan\theta = y/x = -\sqrt{3}/1 = -\sqrt{3}$, the phase θ is either 120° or 300°. However, since z lies in quadrant IV then $\theta = 300°$, and its polar form is

$$z = r(\cos\theta + i\sin\theta) = 2(\cos 300° + i\sin 300°).$$

Note that z may also be represented by

$$z = 2[\cos(300° + n360°) + i\sin(300° + n360°)]$$

where n is any integer. For instance, we could write $z = 2(cos(-60°)+isin(-60°)$.

J.1 Multiplication in Polar Form

The polar representation is particularly useful when multiplying complex numbers together. If

$$z = \rho(\cos\phi + i\sin\phi),$$

and

$$z' = \rho'(\cos\phi' + i\sin\phi'),$$

then

$$zz' = \rho\rho'\,[(\cos\phi\cos\phi' - \sin\phi\sin\phi') + i(\cos\phi\sin\phi' + \sin\phi\cos\phi')]$$

Using the addition theorems for sine and cosine,

$$\cos\phi\cos\phi' - \sin\phi\sin\phi' = \cos(\phi + \phi'),$$
$$\cos\phi\sin\phi' + \sin\phi\cos\phi' = \sin(\phi + \phi').$$

the equation can be rewritten as:

$$zz' = \rho\rho'\,[\cos(\phi + \phi') + i\sin(\phi + \phi')].$$

This form has modulus $\rho\rho'$ and phase $\phi + \phi'$. In other words, to multiply two complex numbers, we multiply their moduli and *add* their angles, as visualized in Fig. J.3.

J.2 De Moivre's Formula

The polar multiplication formula has a particularly important consequence when $z = z'$, for then we have

$$z^2 = \rho^2(\cos 2\phi + i\sin 2\phi).$$

Multiplying again by z we obtain

$$z^3 = \rho^3(\cos 3\phi + i\sin 3\phi),$$

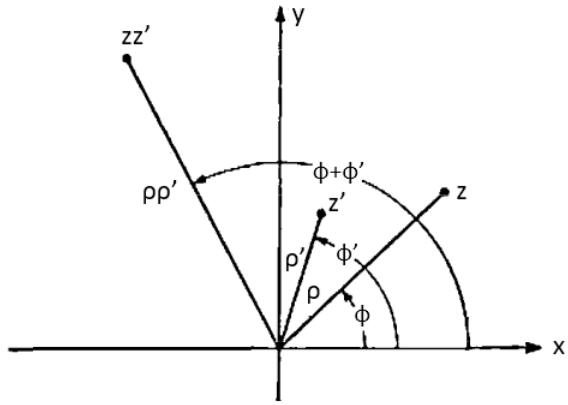


Figure J.3. $1 - i\sqrt{3}$ in Polar Form

and continuing indefinitely in this way,

$$z^n = \rho^n (\cos n\phi + i \sin n\phi), \quad \text{for any integer } n.$$

For example,

$$\begin{aligned}(\sqrt{3} - i)^{10} &= \{2(\cos 330^\circ + i \sin 330^\circ)\}^{10} \\ &= 2^{10}(\cos(10 \cdot 330^\circ) + i \sin(10 \cdot 330^\circ)) \\ &= 1024(\cos 60^\circ + i \sin 60^\circ) = 1024(1/2 + i\sqrt{3}/2) \\ &= 512 + 512i\sqrt{3}.\end{aligned}$$

If z is a point on the unit circle, with $\rho = 1$, we obtain the formula discovered by A. De Moivre (1667–1754):

$$(\cos \phi + i \sin \phi)^n = \cos n\phi + i \sin n\phi.$$

J.3 The nth Root of a Number

A complex number $a + bi = r(\cos \theta + i \sin \theta)$ has exactly n distinct nth roots, which can be readily obtained by applying De Moivre's formula. As an example, let's find the fifth roots of $(4 - 4i)$ by considering its polar version $4\sqrt{2}(\cos 315^\circ + i \sin 315^\circ)$ in its most general form

$$4\sqrt{2}[\cos(315^\circ + k360^\circ) + i \sin(315^\circ + k360^\circ)],$$

where k is any integer, including zero. Using De Moivre's theorem, the fifth root of $(4-4i)$ is

$$\begin{aligned}
&\{4\sqrt{2}[\cos(315^\circ + k360^\circ) + i\sin(315^\circ + k360^\circ)]\}^{1/5} \\
&= (4\sqrt{2})^{1/5}\left(\cos\frac{315^\circ + k360^\circ}{5} + i\sin\frac{315^\circ + k360^\circ}{5}\right) \\
&= \sqrt{2}[\cos(63^\circ + k72^\circ) + i\sin(63^\circ + k72^\circ)].
\end{aligned}$$

Assigning in turn the values $k = 0, 1, 2, \cdots$, we find

$$\begin{aligned}
k = 0: &\quad \sqrt{2}(\cos 63^\circ + i\sin 63^\circ) = R_1 \quad \text{(see Fig. J.4)} \\
k = 1: &\quad \sqrt{2}(\cos 135^\circ + i\sin 135^\circ) = R_2 \\
k = 2: &\quad \sqrt{2}(\cos 207^\circ + i\sin 207^\circ) = R_3 \\
k = 3: &\quad \sqrt{2}(\cos 279^\circ + i\sin 279^\circ) = R_4 \\
k = 4: &\quad \sqrt{2}(\cos 351^\circ + i\sin 351^\circ) = R_5 \\
k = 5: &\quad \sqrt{2}(\cos 423^\circ + i\sin 423^\circ) = \sqrt{2}(\cos 63^\circ + i\sin 63^\circ) = R_1
\end{aligned}$$

The five fifth roots were obtained with $k = 0, 1, 2, 3, 4$, and in the general nth root case by assigning $0, 1, 2, 3, \ldots, n-1$ to k.

The modulus of each of the roots is $\sqrt{2}$ so they all lie on a circle of radius $\sqrt{2}$ centered at the origin. The difference in phase of two consecutive roots is 72° so the roots are equally spaced out around the circle, as in Fig. J.4.

J.4 Euler's Formula

In what follows, the expression $\cos\theta + i\sin\theta$ is so common that we'll abbreviate it to $\operatorname{cis}\theta$, so that, for example,

$$\operatorname{cis}\frac{4}{3}\pi = \operatorname{cis}240^\circ = \cos 240^\circ + i\sin 240^\circ = -\frac{1}{2} - \frac{1}{2}i\sqrt{3}.$$

When

$$z = \operatorname{cis}\theta = \cos\theta + i\sin\theta,$$

then

$$dz/d\theta = -\sin\theta + i\cos\theta = iz$$

We often see differential equations of this sort in physics: $dy/dt = ky$, and the solution is $y = Ae^{kt}$ (where A is the value of y when $t = 0$). By analogy, it isn't too unreasonable to hope that the solution of our derivative might be something like

$$z = Ae^{i\theta}, \quad \text{i.e.,} \quad \operatorname{cis}\theta = Ae^{i\theta}$$

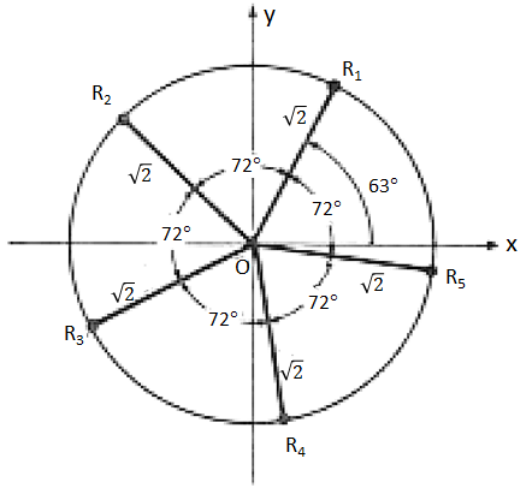


Figure J.4. The Fifth Roots of $4 - 4i$

Putting $\theta = 0$, we get $A = 1$, so

$$\operatorname{cis}\theta = e^{i\theta}.$$

To express what we mean by a power a^z when z is complex, we need a definition that produces the standard value of a^x when z is real, but also gives $a^{z_1} \times a^{z_2} = a^{z_1+z_2}$ for all complex numbers z_1 and z_2. A suitable definition utilizes an infinite series of complex terms:

$$e^z = 1 + z + z^2/2! + z^3/3! + z^4/4! + \cdots + z^n/n! + \cdots$$

which can be shown to converge for all values of z.

It's also possible to prove that $e^{z_1} \cdot e^{z_2} = e^{z_1+z_2}$, so that $e^z = e^{x+iy} = e^x \cdot e^{iy}$, where x and y are real. By definition, e^{iy} is

$$\begin{aligned} e^{iy} &= 1 + iy + (iy)^2/2! + (iy)^3/3! + \cdots \\ &= 1 + iy - y^2/2! - iy^3/3! + y^4/4! + \cdots \\ &= (1 - y^2/2! + y^4/4! - \cdots) + i(y - y^3/3! + y^5/5! - \cdots) \end{aligned}$$

It can be shown by Maclaurin's expansion that for all y,

$$\cos y = 1 - y^2/2! + y^4/4! - \cdots \quad \text{and} \quad \sin y = y - y^3/3! + y^5/5! - \cdots$$

so we have

$$e^{iy} = \cos y + i \sin y$$

or

$$e^{i\theta} = \cos\theta + i \sin\theta,$$

a formula due to the great mathematician Euler (1707-1783). Note that when $\theta = \pi$, we have $e^{i\pi} + 1 = 0$, a remarkable equation connecting the mathematical Big Five – e, π, i, 1, and 0.

The formula

$$\operatorname{cis}\theta_1 \cdot \operatorname{cis}\theta_2 = \operatorname{cis}(\theta_1 + \theta_2)$$

can now be rephrased as:

$$e^{i\theta_1} \cdot e^{i\theta_2} = e^{i(\theta_1+\theta_2)}$$

It's trivial to derive De Moivre's formula using Euler's formula and the exponential law for integer powers

$$(e^{ix})^n = e^{inx},$$

since Euler's formula implies that the left side of this equality is equivalent to $(\cos x + i \sin x)^n$ while the right side is equal to $\cos nx + i \sin nx$.

Using the r, θ polar forms, three useful consequences are

$$\begin{aligned} r_1 e^{i\theta_1} \cdot r_2 e^{i\theta_2} &= r_1 r_2 e^{i(\theta_1+\theta_2)} \\ (re^{i\theta})^n &= r^n e^{in\theta} \\ (re^{i\theta})^{1/n} &= r^{1/n} e^{i\theta/n} \end{aligned}$$

The angle θ is periodic — a complete rotation of 360° or 2π radians leaves the trigonometric functions unchanged, so that:

$$\begin{aligned} e^{i\theta} &= e^{i(\theta+2\pi)} = e^{i(\theta+4\pi)} = e^{i(\theta+6\pi)} = \ldots \\ (re^{i\theta})^{1/n} &= r^{1/n} e^{i\theta/n} = r^{1/n} e^{i(\theta+2\pi)/n} = \ldots \end{aligned}$$

and so every complex number has n n-th roots:

$$\sqrt[n]{z} = z^{1/n} = (re^{i\theta})^{1/n} = r^{1/n} e^{i(\theta+2\pi k)/n}, \quad k = 0, 1, 2, \ldots, n-1$$

J.5 Multi-valued Exponentiation

e^z is not a one-to-one function for complex numbers, which is a problem because a function can only have an inverse if it maps distinct values to distinct values. We need to exclude enough complex numbers from z's domain so that any e^z result can only be generated by a single z. The problem is that a complex number's polar angle can be represented by $\theta + 2\pi k$, where k is any integer. The most common choice for restricting its range is to require $-\pi < \theta \leq \pi$, which is called a *principal branch*.

J.5.1 The Complex Logarithm, $\log z$. Once e^z is defined to be one-to-one, it makes sense to find its inverse, $\log z$. We can define it by considering the polar form $z = re^{i\theta}$. Taking logs of both sides results in

$$\log z = \log(re^{i\theta}) = \log r + i\theta = \log r + i(\theta + 2\pi k)$$

As usual, the angle is θ plus any multiple of 2π, which poses a familiar problem. The left side of Fig. J.5 shows the imaginary part of $\log z$ spiralling up the z-axis, highlighting how a single $(x + iy)$ point can generate an infinite number of log values depending on how its polar angle is interpreted.

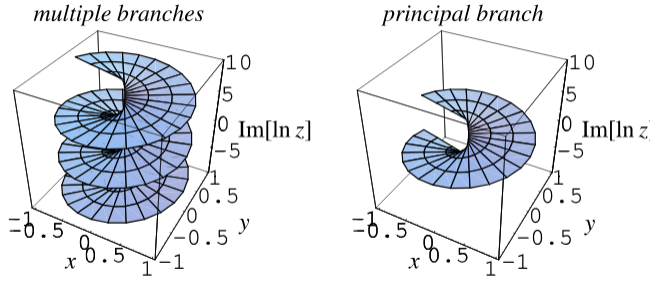


Figure J.5. 3D Surface Plot of $\log(z)$ Showing its Imaginary Part

This is avoided by applying angle-based branch cuts to the log function similar to those imposed on e^z. The angles are constrained so that

$$\log(z) = \log r + i\theta \quad \text{where } -\pi < \theta \leq \pi$$

The right-hand side of Fig. J.5 shows what this does to the imaginary part of $\log z$. When $\theta \approx \pi$, z is just above the negative real x-axis, and $\theta \approx -\pi$ is just below the axis.

Although this constrains $\log z$ to be single-valued, it introduces a discontinuity at the boundaries of the branch cuts along the negative x-axis – the gap visible in the figure.

J.5.2 Other Complex Math Functions. The exponential function is the foundation for most of the other complex math functions. For example,

$$\sin z = \frac{e^{iz} - e^{-iz}}{2i}, \qquad \cos z = \frac{e^{iz} + e^{-iz}}{2},$$

and

$$\sinh z = \frac{e^{z} - e^{-z}}{2}, \qquad \cosh z = \frac{e^{z} + e^{-z}}{2}.$$

Of course, this implies that

$$\sin(iz) = i\sinh z, \qquad \cos(iz) = \cosh z.$$

The inverse complex functions are defined using logarithms:

$$\begin{aligned} \arcsin z &= -i\log(iz + \sqrt{1-z^2}), & \arccos z &= -i\log(z + i\sqrt{1-z^2}), \\ \sinh^{-1} z &= \log(z + \sqrt{z^2+1}), & \cosh^{-1} z &= \log(z + \sqrt{z^2-1}). \end{aligned}$$

The multi-valued nature of the angles used in $z = re^{i\theta}$ mean that these functions must also utilize branch cuts to limit their range of inputs and outputs.

J.6 Complex Numbers in Python

The imaginary unit symbol is written as `j` in Python, so $3+4i$ becomes `3+4j`, but i on its own is written as `1j`, not `j`. Below is a sample session involving complex numbers and simple arithmetic:

```
>>> a = 3 + 4j
>>> b = 1 - 2j
>>> a + b
(4+2j)
>>> a - b
(2+6j)
>>> a * b
(11-2j)
>>> a / b
(-1+2j)
>>> a + 2
(5+4j)
>>> a * 2
(6+8j)
>>> a / 2
(1.5+2j)
>>> a**2
(-7+24j)
>>> a ** b      # complex powers
(-21.083139690689016-24.00021070941257j)
>>> a**(1/2)    # roots
(2+1j)
>>> a.conjugate()
(3-4j)
>>> abs(a)      # the modulus
5.0
>>> a.real
3.0
>>> a.imag
4.0
>>> import math
>>> math.degrees(
        math.atan2(a.imag, a.real))
53.13010235415598
```

We didn't need to import `cmath` for any of these basic operations, but `math` is necessary if we want to get a number's polar angle.

The following session shows some of the common errors a user may encounter when first using complex numbers in Python:

```
>>> c = 3 + 4i
            ^
SyntaxError: invalid syntax
>>> (1 + 2j) * (3 + 4j)
(-5+10j)
>>> 1 + 2j * (3 + 4j)     # parenthesis needed
```

```
(-7+6j)
>>> (3 + 4j) // 2
TypeError: can't take floor of complex number.
>>> math.sqrt(1 + 2j)
TypeError: can't convert complex to float
>>> math.sin((3 + 4j))
TypeError: can't convert complex to float
```

J.6.1 Using `cmath`. A common misunderstanding is that the functions in the `math` module will work with complex numbers, which isn't the case as the session above illustrates. For that capability, the `cmath` module (`https://docs.python.org/3/library/cmath.html`) adds an assortment of exponential, logarithmic, trigonometric, and hyperbolic functions, as listed in Table J.1.

Exponential and Logarithmic Functions	
`exp(z)`	Return e^z
`log(z[, base])`	Return the logarithm of z to the given base (default: natural log)
`log10(z)`	Return the base-10 logarithm of z
`sqrt(z)`	Return the square root of z
Trigonometric Functions	
sin(z), cos(z), tan(z), asin(z), acos(z), atan(z)	
Hyperbolic Functions	
sinh(z), cosh(z), tanh(z), asinh(z), acosh(z), atanh(z)	
Complex Number Conversion	
`phase(z)`	Return the phase (angle) of complex number z in radians
`polar(z)`	Convert complex number z to polar coordinates (r, ϕ)
`rect(r, phi)`	Convert polar coordinates (r, ϕ) to complex number
Classification Functions	
`isfinite(z)`	Return True if both real and imaginary parts are finite
`isinf(z)`	Return True if either real or imaginary part is infinity
`isnan(z)`	Return True if either real or imaginary part is NaN
`isclose(a, b)`	Return True if values a and b are close to each other
Constants	
cmath.pi, cmath.e, cmath.tau, cmath.inf, cmath.infj, cmath.nan, cmath.nanj	

Table J.1. Python `cmath` Module

The following session shows a few uses of `cmath`:

```
>>> a = 3 + 4j
>>> import cmath
>>> r, theta = cmath.polar(a)
>>> import math
>>> math.degrees(theta)
53.13010235415598
>>> r
5.0
>>> cmath.rect(r, theta)
(3.0000000000000004+3.9999999999999996j)
>>> cmath.sin(a)
(3.853738037919377-27.016813258003936j)
>>> cmath.exp(a)
(-13.128783081462158-15.200784463067954j)
>>> cmath.exp(cmath.log(a))
(3+3.999999999999999j)
>>> cmath.exp(1j*math.pi) + 1      # Euler's equation e^{i pi) + 1 = 0
1.2246467991473532e-16j
```

J.6.2 The *n*th Root using Python. Let's return to the nth root example from earlier, but this time coded in Python. The aim is to find the fifth roots of $(4 - 4i)$. The following code creates the variable:

```
>>> a = 4 - 4j
>>> cmath.polar(a)
(5.656854249492381, -0.7853981633974483)   # r, theta
>>> 4 * math.sqrt(2)      # this is r
5.656854249492381
>>> math.radians(315)     # this is not theta
5.497787143782138
>>> math.radians(315-360) # this is theta
-0.7853981633974483
```

The polar angle for $(4 - 4j)$ has been normalized to fall between $-\pi$ and π.

The next fragment shows the *single* value returned for $a^{1/5}$:

```
>>> root = a ** (1/5)
>>> root
(1.3968022466674208-0.22123174208247431j)
>>> cmath.polar(root)
(1.4142135623730954, -0.15707963267948966)  # r, theta
>>> cmath.phase(root)
-0.15707963267948966      # theta again
>>> math.degrees( cmath.phase(root))   # in degrees
-9.0
```

If we look back at Fig. J.4, it turns out that Python is giving us R_5, but the angle is $-9°$ rather than $351°$.

We could get the other roots by adding and subtracting multiples of 360/5° to this result, but a simpler approach is to use our getRoots() function, imported from complexUtils.py:

```
>>> import complexUtils
>>> rs = complexUtils.getRoots(a, 5)     # a^(1/5)
>>> for r in rs: print(r)
(1.3968022466674208-0.22123174208247431j)
(0.6420395219202063+1.2600735106701009j)
(-1+1.0000000000000002j)
(-1.260073510670101-0.6420395219202206j)
(0.22123174208247404-1.3968022466674208j)
```

Once again referring to Fig. J.4, we can see that the roots are listed in counterclockwise order starting with R_5. The code for getRoots() implements the formula:

$$z^{1/n} = (re^{i\theta})^{1/n} = r^{1/n} * e^{i(\theta+2\pi k)/n}, \quad \text{for } k = 0, 1, 2, \ldots, n-1$$

```
def getRoots(z, n):
  # Handle zero specially
  if z.real == 0 and z.imag == 0:
    return [complex(0, 0)]
  # Get principal root
  rRoot = abs(z)**(1/n)
  theta = cmath.phase(z)
  # Calculate all n roots
  roots = []
  for k in range(n):
    ang = (theta + 2*math.pi * k)/n
    roots.append( rRoot*cmath.exp(1j*ang))
  return roots
```

Going one step better, showRoots.py plots these roots in the style of Fig. J.4. In the following call, the user supplies three integers representing $(4-4i)^{1/5}$:

```
> python showRoots.py
Real part: 4
Imaginary part: -4
Root number (n): 5
```

The image in Fig. J.6 is generated.

J.6.3 Investigating Branch Cuts. The branch cut for `cmath.log()` restricts a complex number's angle to be in the range $-\pi < \theta \leq \pi$, which corresponds to either side of the negative real x-axis. This can be confirmed by testing some values which are just above and just below the branch cut:

```
>>> above = cmath.rect(1, math.pi-0.00001)  # e^(i pi - eps)
>>> above
```

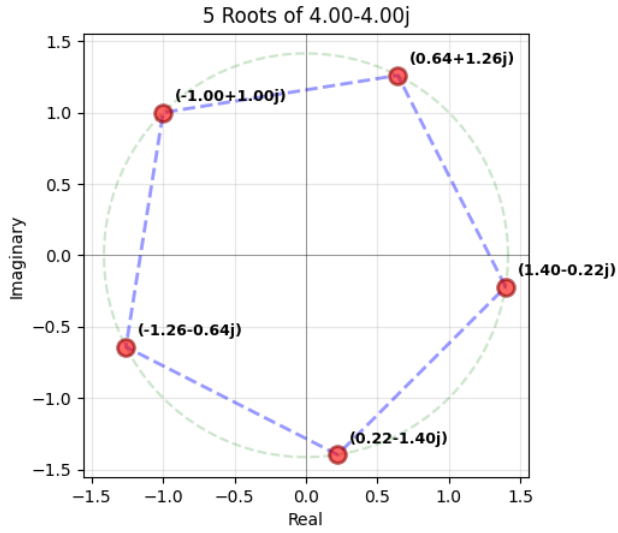


Figure J.6. The Fifth Roots of 4-4i using `showRoots.py`

```
(-0.99999999995+1.000000000002131e-05j)     # (-1 + <small>i)
>>> below = cmath.rect(1, -math.pi+0.00001)  # e^(i -pi + eps)
>>> below
(-0.99999999995-1.000000000002131e-05j)      # (-1 - <small>i)
>>> cmath.log(above)
(-4.1355554465750615e-18+3.141582653589793j)  # 0 + i pi
>>> cmath.log(below)
(-4.1355554465750615e-18-3.141582653589793j)  # 0 - i pi
```

`above` and `below` have different polar coordinates, but their rectangular forms are just above and below $x = -1$. Fig. J.7 shows these two points on the principal branch for `cmath.log()` from Fig. J.5.

The discontinuity in the log curve becomes apparent when the logarithms of `above` and `below` are compared – their imaginary parts differ by almost 2π. In other words, two complex numbers in close proximity have been mapped to very different places on the log curve.

This discontinuity issue exists for all of the complex math functions, so care has to be taken when using them. viewCMath.py offers a way to visualize these curves by creating four plots of the `cmath` function supplied by the user. For instance, in the following call the user has typed `log`:

```
> python viewCMath.py
Enter function name: log
Plotting log(z)
```

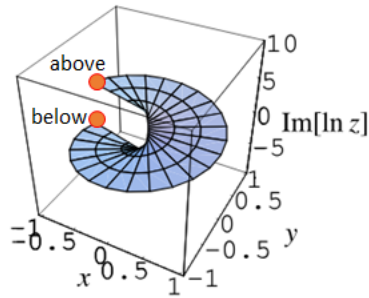


Figure J.7. The above and below points on the Principal Branch for cmath.log()

```
Real range: [-3.0, 3.0]
Imaginary range: [-3.0, 3.0]
```

The program reads the ranges for the real and imaginary axes from the text file cmFuncs.txt, and generates the four plots of cmath.log(z) shown in Fig. J.8.

The plots can be individually rotated, and the 'Imaginary Part' graph in Fig. J.8 has been turned so it corresponds to the image in Fig. J.5. The branch cut isn't rendered as a gap but rather as a very abrupt jump in the values plotted on the z-axis.

Each graph is based on a series of (x, y, z) points generated by calling the specified function on $(x + iy)$ to obtain z. Each of these z complex numbers is converted in a different way appropriate for a particular graph. For example, the imaginary plot utilizes z.imag values.

The drawback of 3D plots is that they only represent three dimensions of data. Across the base of the graph, we're plotting the real and imaginary parts of each complex number input to the function ($\log z$ in Fig. J.8), but only *part* of the complex number returned by the function along the z-axis.

For example, the $(x + iy)$ output by a call to $\log z$ is divided into the top-right plot for each x and the bottom-left plot for each y. Argueably this is useful because the branch cut related to the polar angle range becomes clearer when the imaginary output is separated out.

Another very common approach is to draw phase plots enhanced with domain coloring and contour lines. For example, Fig. J.9 is generated by phaseCMath.py when the user enters log:

```
> python phaseCMath.py
Enter function name: log
Plotting log(z)
```

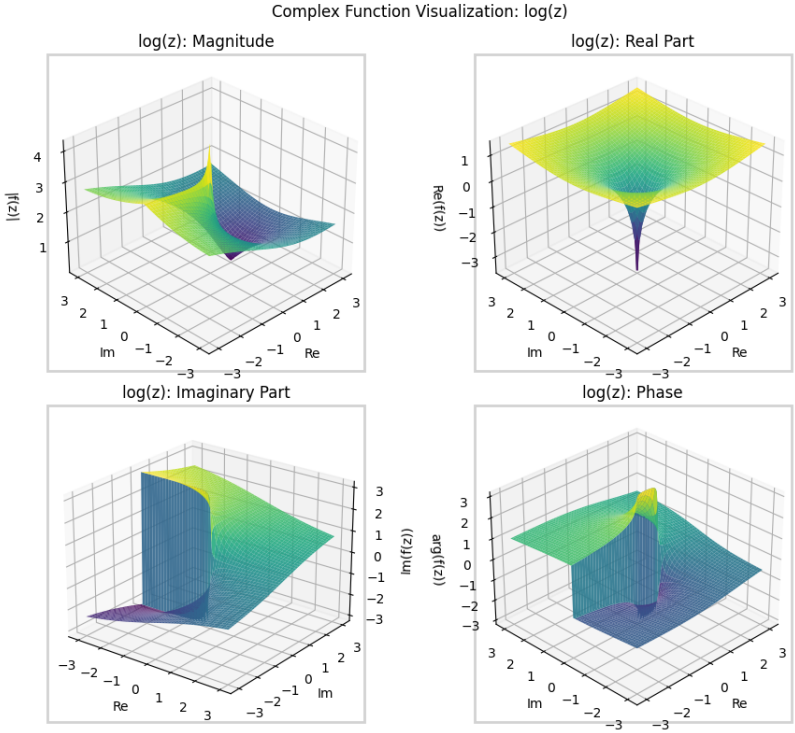


Figure J.8. Visualizing `cmath.log(z)`

```
Real range: [-3.0, 3.0]
Imaginary range: [-3.0, 3.0]
```

Fig. J.9 corresponds most closely to the bottom-right plot of the phase information in Fig. J.8, but the colors, and their ordering, have additional meaning.

Positive numbers are colored red, negative numbers are colored blue, and zeros and poles occur at the points where all the colors meet. A zero is distinguished by the change from red, yellow, green, to blue in a counter-clockwise order around the point. A pole, which is when the output extends off to infinity, has its colors change in a clockwise order. This means that a zero is present on the middle right of Fig. J.9.

A sudden jump from one color to another, like the one that occurs in the middle left of Fig. J.9 is an indication of a discontinuity caused by a branch cut.

There are two types of contour line drawn over the surface – the black lines are contours of equal phase, and are labelled in degrees. The white lines are

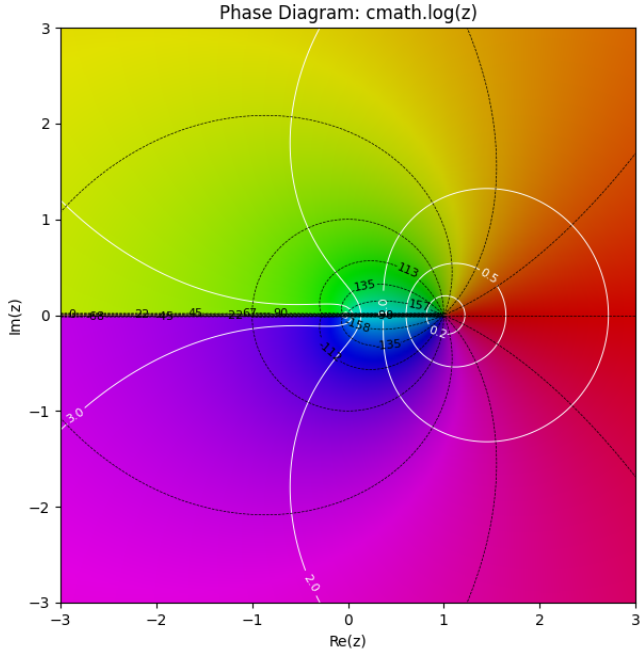


Figure J.9. Visualizing `cmath.log(z)` with a Phase Plot

contours of equal magnitude (modulus), and so are showing the data that is presented in the top-left graph of Fig. J.8. In Fig. J.9 these reinforce the fact that a zero occurs on the middle right, and that the magnitudes increase to the middle left.

Section 17.8 on solving cubic equations uses much of the same code (which is located in `complexUtils.py`). Alternatively, a very nice online tool that does much the same job can be found at `https://www.dynamicmath.xyz/complex/function-plotter/hsv.htm`. A good overview of the approach can be found in [**Weg16**].

Bibliography

[AD86] H. Abelson and A. Disessa, *Turtle geometry: The computer as a medium for exploring mathematics*, Artificial Intelligence Series, MIT Press, 1986.

[AH22] S. Andrilli and D. Hecker, *Elementary linear algebra*, Academic Press, 2022.

[Ahr01] W Ahrens, *Mathematische unterhaltungen und spiele*, Leipzig, B.G. Teubner, 1901.

[All78] Dennis Allison, *Quick and dirty functions for basic*, Dr. Dobb's Journal **3** (1978), no. 9.

[AM10] Anurag Agarwal and James Marengo, *The locus of the focus of a rolling parabola*, The College Mathematics Journal **41** (2010), no. 2, pp. 129–133.

[ASS96] H. Abelson, G.J. Sussman, and J. Sussman, *Structure and interpretation of computer programs, second edition*, MIT Electrical Engineering and Computer Science, MIT Press, 1996.

[Bar12] M. F. Barnsley, *Fractals everywhere*, 3rd ed ed., Dover Pub., 2012.

[BC87] W.W.R. Ball and H.S.M. Coxeter, *Mathematical recreations and essays*, Dover Recreational Math Series, Dover Pub., 1987.

[BCG04] E.R. Berlekamp, J.H. Conway, and R.K. Guy, *Winning ways for your mathematical plays, volumes 1-4*, AK Peters/CRC Recreational Mathematics Series, CRC Press, 2004.

[Bea06] J.D. Beasley, *The mathematics of games*, Dover books on mathematics, Dover Pub., 2006.

[Ben88] J.L. Bentley, *More programming pearls: Confessions of a coder*, Addison-Wesley, 1988.

[Ben00] ———, *Programming pearls*, 2nd ed ed., Addison-Wesley, 2000.

[BFB16] Richard L. Burden, J. Douglas. Faires, and Annette M. Burden, *Numerical analysis*, 10th ed ed., Cengage Learning, 2016.

[BH19] J.K. Blitzstein and J. Hwang, *Introduction to probability*, 2nd ed ed., Chapman & Hall/CRC Texts in Statistical Science, CRC Press, 2019.

[BHS93] G. Blom, L. Holst, and D. Sandell, *Problems and snapshots from the world of probability*, Springer New York, 1993.

[Bla16] Viktor Blasjö, *How to find the logarithm of any number using nothing but a piece of string*, The College Mathematics Journal **47** (2016), no. 2, 95–100.

[Bol12] B. Bold, *Famous problems of geometry and how to solve them*, Dover Books on Mathematics, Dover Publications, 2012.

[Bri89] K. M Briggs, *How to calculate the Feigenbaum constants on your pc*, Aust. Math Soc. Gazette **16** (1989), 89—92.

[Bro88] R. Bronson, *Schaum's outline of theory and problems of matrix operations*, McGraw-Hill, 1988.

[BS09] J. Bell and B. Stevens, *A survey of known results and research areas for n-queens*, Discrete Math. **309** (2009), no. 1, 1–31.

[BS24] J. Bryant and C. Sangwin, *How round is your circle?: Where engineering and mathematics meet*, Princeton University Press, 2024.

[BT08] D. Bertsekas and J.N. Tsitsiklis, *Introduction to probability*, Athena Scientific optimization and computation series, Athena Scientific, 2008.

[CLRS22] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to algorithms, fourth edition*, MIT Press, 2022.

[Dar04] D. Darling, *The universal book of mathematics: From abracadabra to zeno's paradoxes*, Wiley, 2004.

[dBOCvKO08] M. de Berg, O. Otfried Cheong, M. van Kreveld, and M Overmars, *Computational geometry: Algorithms and applications*, 3rd ed ed., Springer, 2008.

[Dij76] E.W. Dijkstra, *A discipline of programming*, Prentice-Hall series in Automatic Computation, Prentice-Hall, 1976.

[DKL98] Erik Bjørnager Dam, Martin Koch, and Martin Lillholm, *Quaternions, interpolation and animation*, Technical report diku-tr—98/5, University of Copenhagen Universitetsparken, July 1998.

[Dom63] A.P. Domoryad, *Mathematical games and pastimes: Popular lectures in mathematics*, Macmillan, London, 1963.

[Dox10] A. Doxiadis, *Uncle Petros and Goldbach's conjecture: A novel of mathematical obsession*, Bloomsbury Pub., 2010.

[DS76] L.E. Dubins and L.J. Savage, *How to gamble if you must: Inequalities for stochastic processes*, Dover Books on Mathematics, Dover Pub., 1976.

[Dun90] W. Dunham, *Journey through genius: Great theorems of mathematics*, A Penguin book, Wiley, 1990.

[Ebe10] David Eberly, *Quaternion algebra and calculus*, `https://www.geometrictools.com/Documentation/Quaternions.pdf`, 2010, [Accessed 14-08-2025].

[Eng76] A. Engel, *Wahrscheinlichkeitsrechnung und statistik, band.2*, Klett Studienbücher, Stuttgart, 1976 (de).

[Eng84] ———, *Elementary mathematics from an algorithmic standpoint*, Keele Mathematical Education Pub., 1984.

[Eng87] ———, *Stochastik*, Klett, Stuttgart, 1987 (de).

[Eps13] Richard A. Epstein, *The theory of gambling and statistical logic*, 2nd ed ed., Elsevier/Academic Press, 2013.

[Fal13] K. Falconer, *Fractal geometry: Mathematical foundations and applications*, John Wiley, 2013.

[Fee00] T.G. Feeman, *Portraits of the earth: A mathematician looks at maps*, Mathematical world, American Mathematical Soc., 2000.

[Fel68] W. Feller, *An introduction to probability theory and its applications, volume 1*, 3rd ed ed., John Wiley, 1968.

[Fel71] ———, *An introduction to probability theory and its application, volume ii*, 2nd ed ed., John Wiley, 1971.

[FU16] M. Frame and A. Urry, *Fractal worlds: Grown, built, and imagined*, Yale Univ. Press, 2016.

[Gar61] M. Gardner, *The second Scientific American book of mathematical puzzles and diversions*, An Essandess paperback, Touchstone, 1961.

[Gar71] ———, *Sixth book of mathematical games from Scientific American*, Scribner, New York, NY, USA, 1971.

[Gar75] ———, *Mathematical carnival*, Random House, 1975.

[Gar88] ———, *Riddles of the sphinx: And other mathematical puzzle tales*, New mathematical library, Mathematical Assoc. of America, 1988.

[Gar97] ———, *Penrose tiles to trapdoor ciphers: And the return of Dr. Matrix*, MAA spectrum, Mathematical Assoc. of America, 1997.

[Gar20] ———, *Knotted doughnuts and other mathematical entertainments*, American Mathematical Soc., 2020.

[GH97] J. Gullberg and P. Hilton, *Mathematics from the birth of numbers*, WW Norton, 1997.
[GKP94] R.L. Graham, D.E. Knuth, and O. Patashnik, *Concrete mathematics: A foundation for computer science*, 2nd ed ed., Addison-Wesley, 1994.
[Gol81] Solomon W. Golomb, *Shift register sequences*, Aegean Park Press, 1981.
[Gol91] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys **23** (1991), no. 1, 5–48.
[Gri81] D. Gries, *The science of programming*, Springer Heidelberg, 1981.
[GS97] C.M. Grinstead and J.L. Snell, *Introduction to probability*, Open Textbook Library, American Mathematical Society, 1997.
[Guy04] R.K. Guy, *Unsolved problems in number theory*, Problem Books in Mathematics, Springer New York, 2004.
[Har68] J.F. Hart, *Computer approximations*, SIAM Series, Wiley, 1968.
[Har01] Suzanne Harper, *Exploring infinite series using Baravelle spirals*, Virginia Mathematics Teacher **27** (2001), 46–54.
[Hav19] Julian Havil, *Curves for the mathematically curious: An anthology of the unpredictable, historical, beautiful, and romantic*, Princeton University Press, 2019.
[HCV99] D. Hilbert and S. Cohn-Vossen, *Geometry and the imagination*, 2nd ed ed., AMS Chelsea Pub., 1999.
[Hea26] P. J. Heawood, *Comparison between rhumb-line and great-circle courses*, The Mathematical Gazette **13** (1926), no. 183, 137–145.
[Hig02] P. Higgins, *Mathematics for the imagination*, Popular science, OUP Oxford, 2002.
[HK78] I. N. Herstein and I. Kaplansky, *Matters mathematical*, 2nd ed ed., Chelsea Pub., 1978.
[Hof99] D.R. Hofstadter, *Gödel, Escher, Bach: An eternal golden braid*, 20th anniversary ed ed., Basic Books, 1999.
[Hog93] L. Hogben, *Mathematics for the million: How to master the magic of numbers*, W.W. Norton, 1993.
[Hon70] R. Honsberger, *Ingenuity in mathematics*, New Mathematical Library, no. 23, Mathematical Assoc. of America, 1970.
[HS79] P. Holgate and H. Solomon, *Geometric probability*, Journal of the Royal Statistical Society. Series A **142** (1979), no. 3, 386.
[HUA83] J.E. Hopcroft, J.D. Ullman, and A.V. Aho, *Data structures and algorithms*, Addison-Wesley, 1983.
[Hug14] J.F. Hughes, *Computer graphics: Principles and practice*, The systems programming series, Addison-Wesley, 2014.
[HW92] Leon Hall and Stan Wagon, *Roads and wheels*, Mathematics Magazine **65** (1992), no. 5, 283–301.
[HWHBS08] G.H. Hardy, E.M. Wright, D.R. Heath-Brown, and J. Silverman, *An introduction to the theory of numbers*, Oxford mathematics, OUP Oxford, 2008.
[IR90] K. Ireland and M.I. Rosen, *A classical introduction to modern number theory*, 2nd ed ed., Graduate Texts in Mathematics, Springer, 1990.
[Ise92] C. Isenberg, *The science of soap films and soap bubbles*, Dover books explaining science, Dover Publications, 1992.
[Jac69] K. Jacobs, *Selecta mathematica i*, Springer-Verlag, 1969.
[KG81] D. A. Klarner and M. Gardner (eds.), *The mathematical Gardner*, Prindle, Weber and Schmidt, 1981.
[Kiu13] J. Kiusalaas, *Numerical methods in engineering with python 3*, Cambridge University Press, 2013.
[Kle12] F. Klein, *Elementary mathematics from an advanced standpoint: Geometry*, Dover Books on Mathematics, Dover Pub., 2012.
[KN01] E. Kasner and J.R. Newman, *Mathematics and the imagination*, Dover Books on Mathematics Series, Dover Pub., 2001.

[Knu96] D.E. Knuth, *Selected papers on computer science*, CSLI Lecture Notes, no. No. 59, Cambridge Univ. Press, 1996.

[Knu97] ———, *The art of computer programming*, 3rd ed ed., vol. 1, 2, 3, Addison-Wesley, 1997.

[Kre10] E. Kreyszig, *Advanced engineering mathematics*, John Wiley & Sons, 2010.

[KS83] J.G. Kemeny and J.L. Snell, *Finite markov chains: With a new appendix "generalization of a fundamental matrix"*, Undergraduate Texts in Mathematics, Springer New York, 1983.

[KST74] J.G. Kemeny, J.L. Snell, and G.L. Thompson, *Introduction to finite mathematics*, Prentice-Hall mathematics series, Prentice-Hall, 1974.

[Kuc94] M.E. Kuczma, *Problems: 144 problems of the Austrian-Polish mathematics competition, 1978-1993*, Academic Distribution Center, 1994.

[Len11] E. Lengyel, *Mathematics for 3D game programming and computer graphics*, third ed., Cengage Learning, 2011.

[LG59] S. Loyd and M. Gardner, *Mathematical puzzles*, Dover Brain Games: Math Puzzles Series, no. Vol. 1, Dover Pub., 1959.

[LG60] ———, *More mathematical puzzles of Sam Loyd*, Dover Recreational Math Series, no. Vol. 2, Dover Pub., 1960.

[LH89] Cliff Long and Thomas Hern, *Graphing the complex zeros of polynomials using modulus surfaces*, The College Mathematics Journal **20** (1989), no. 2, 98–105.

[LLM21] D.C. Lay, S.R. Lay, and J.J. McDonals, *Linear algebra and its applications*, 6th ed ed., Pearson Education, 2021.

[Loc67] E.H. Lockwood, *A book of curves*, University Press, 1967.

[Lüt22] J. Lützen, *A history of mathematical impossibility*, OUP Oxford, 2022.

[Man67] B.B Mandelbrot, *How long is the coast of Britain*, Science **156** (1967), no. 3775, 636–638.

[Man83] ———, *The fractal geometry of nature*, vol. 173, Macmillan, 1983.

[Mao20] E. Maor, *Trigonometric delights*, Princeton University Press, 2020.

[Mar05] D. Marsh, *Applied geometry for computer graphics and CAD*, Springer Undergraduate Mathematics Series, Springer London, 2005.

[Mor99] M.E. Mortenson, *Mathematics for computer graphics applications*, Industrial Press, 1999.

[Nah07] P.J. Nahin, *Chases and escapes: The mathematics of pursuit and evasion*, Princeton Puzzlers, Princeton Univ. Press, 2007.

[Nah13] ———, *Digital dice: Computational solutions to practical probability problems*, Paradoxes, perplexities, and mathematical conundrums for the serious head scratcher, Princeton Univ. Press, 2013.

[Niv65] I. Niven, *Mathematics of choice: or, how to count without counting*, Anneli Lax New Mathematical Library, Random House, 1965.

[NZM91] I. Niven, H.S. Zuckerman, and H.L. Montgomery, *An introduction to the theory of numbers*, Wiley, 1991.

[Ogi68] C. Stanley Ogilvy, *A calculus notebook*, Prindle, Weber & Schmidt, Boston, 1968.

[Ogi90] C.S. Ogilvy, *Excursions in geometry*, Dover Books on Mathematics Series, Dover Publications, 1990.

[Old63] C.D. Olds, *Continued fractions*, Anneli Lax New Mathematical Library, vol. 9, Random House, 1963.

[Oli93] D. Olivastro, *Ancient puzzles: Classic brainteasers and other timeless mathematical games of the last ten centuries*, Random House Pub. Group, 1993.

[O'R98] J. O'Rourke, *Computational geometry in C*, Cambridge Tracts in Theoretical Computer Science, Cambridge Univ. Press, 1998.

[Ore17] O. Ore, *Invitation to number theory*, 2nd ed; revised and updated by john j. watkins and robin wilson ed., Anneli Lax New Mathematical Library, American Mathematical Society, 2017.

[Pap20] S.A. Papert, *Mindstorms: Children, computers, and powerful ideas*, Basic Books, 2020.

[Ped83] D. Pedoe, *Geometry and the visual arts*, Dover Books on Mathematics Series, Dover Publications, 1983.

[Per17] D. Perkins, *ϕ, π, e and i*, MAA spectrum, MAA, The Mathematical Assoc. of America, 2017.

[Pic95] C.A. Pickover, *Keys to infinity*, W. H. Freeman, 1995.

[Pic01] _____, *Wonders of numbers: Adventures in mathematics, mind, and meaning*, Oxford Univ., 2001.

[Pol20] G. Polya, *Mathematics and plausible reasoning, volume 1: Induction and analogy in mathematics*, Princeton Univ. Press, 2020.

[PR09] W.B. Poucher and M.A. Revilla, *From Baylor to Baylor: 1991-2006 acm-icpc world finals*, Lulu.com, 2009.

[PS89] I. Pressman and D. Singmaster, *The jealous husbands and the missionaries and cannibals*, The Mathematical Gazette **73** (1989), 73 – 81.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical recipes in C: The art of scientific computing*, 2 ed., Cambridge University Press, 1992.

[Rén87] A. Rényi, *A diary on information theory*, Wiley Series in Probability and Mathematical Statistics, Wiley, 1987.

[RND77] E.M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial algorithms: Theory and practice*, Prentice Hall, 1977.

[RND10] S.J. Russell, P. Norvig, and E. Davis, *Artificial intelligence: A modern approach*, Prentice Hall series in AI, Prentice Hall, 2010.

[Rob60] Gerson B. Robison, *Rockers and rollers*, Mathematics Magazine **33** (1960), no. 3, 139–144.

[San04] L.A. Santaló, *Integral geometry and geometric probability*, 2nd ed ed., Cambridge Mathematical Library, Cambridge Univ. Press, 2004.

[Sha78] D. Shanks, *Solved and unsolved problems in number theory*, 2nd ed ed., Chelsea Pub., 1978.

[Sho85] Ken Shoemake, *Animating rotation with quaternion curves*, ACM SIGGRAPH Computer Graphics **19** (1985), no. 3, 245–254.

[Sie64] W. Sierpinski, *Elementary theory of numbers*, PWN- Polish Scientific Pub., 1964.

[Sil12] J.H. Silverman, *A friendly introduction to number theory*, 4th ed ed., Pearson, 2012.

[Sim92] G.F. Simmons, *Calculus gems: Brief lives and memorable mathematics*, McGraw-Hill, 1992.

[Sol78] H. Solomon, *Geometric probability*, CBMS-NSF Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathematics, 1978.

[SR12] Javier Sánchez-Reyes, *The catenary as roulette*, The College Mathematics Journal **43** (2012), no. 3, pp. 216–219.

[Sta12] R.P. Stanley, *Enumerative combinatorics*, Cambridge Studies in Advanced Mathematics, vol. 1, Cambridge Univ. Press, 2012.

[Ste99] H. Steinhaus, *Mathematical snapshots*, Dover Recreational Math Series, Dover Publications, 1999.

[Sti10] J. Stillwell, *Mathematics and its history*, 3rd ed ed., Undergraduate Texts in Mathematics, Springer New York, 2010.

[Sto18] Paul K. Stockmeyer, *4. the tower of hanoi for humans*, Research in Games, Graphs, Counting, and Complexity, Volume 2, pp. 52–70, Princeton University Press, 2018.

[Str91] G. Strang, *Calculus*, 3rd ed ed., Open Textbook Library, Wellesley-Cambridge Press, 1991.

[Str15] S.H. Strogatz, *Nonlinear dynamics and chaos: with applications to physics, biology, chemistry and engineering*, 2nd ed ed., Westview Press, 2015.

[SW11] R. Sedgewick and K. Wayne, *Algorithms*, Addison-Wesley, 2011.

[TFW96] G.B. Thomas, R.L. Finney, and M.D. Weir, *Calculus and analytical geometry*, 9th ed., Addison-Wesley, 1996.

[Tho64] G. (Director) Thomas, *Carry on cleo*, Anglo Amalgamated Film, 1964.

[VB13] G. Van Brummelen, *Heavenly mathematics: the forgotten art of spherical trigonometry*, Princeton University Press, 2013.

[Vor11] N.N Vorobév, *Fibonacci numbers*, Dover Books on Mathematics, Dover Pub, 2011.

[Wag99] Stan Wagon, *The ultimate flat tire*, Math Horizons **6** (1999), no. 3, 14–17.

[Wag10] S. Wagon, *Mathematica in action*, 3rd ed ed., Springer, 2010.

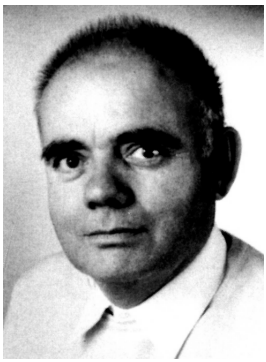
[Weg16] Elias Wegert, *Visual exploration of complex functions*, Mathematical Analysis, Probability and Applications – Plenary Lectures (Tao Qian and Luigi G. Rodino, eds.), Springer International Publishing, 2016, pp. 253–279.

[Yat47] R.C. Yates, *A handbook on curves and their properties*, J.W. Edwards, 1947.

[YG88] D.M. Young and R.T. Gregory, *A survey of numerical mathematics*, A Survey of Numerical Mathematics, no. v. 1, Dover Pub., 1988.

[Zwi96] Daniel Zwillinger, *CRC standard mathematical tables and formulae*, 30 ed., CRC Press, 1996.

Authors Bio



Arthur Engel (1928 – 2022) was a German mathematics teacher, educationalist, and prolific author. He was very active in the German National Mathematics Competition, and quite a few of the problems in this book were inspired by those competitions.

In 1990 Engel was awarded the Federal Cross of Merit by the Minister of Education of West Germany. In 1991 he was one of the first three recipients of the David Hilbert Award of the World Federation of National Mathematics Competitions.

With his 1984 'Elementary Mathematics from an Algorithmic Standpoint', Engel became one of the first educators to propose that mathematics in schools should focus on the concept of algorithm. This was followed in 1993 by the original version of this text, 'Exploring Mathematics with Your Computer', which excitingly illustrated the value of programming for teaching mathematics, and math as a source of informative programming problems.

Andrew Davison (1962 -) is a British computer science teacher and author, with a long career in academia. For many happy years, he has taught numerous topics in the Department of Computer Engineering, Faculty of Engineering, Prince of Songkla University (PSU) in Hat Yai, Thailand (`http://coe.psu.ac.th/~ad/`).

I first encountered Engel's work in the late 1980s through his wonderful 'Elementary Mathematics from an Algorithmic Standpoint', and later 'Exploring Mathematics with Your Computer', which offered many fascinating problems suitable for students of discrete math and algorithms. The math content of these books is timeless, but the programming examples, rooted in BASIC and Pascal, became a major hindrance to their use. I offered to revise 'Exploring Mathematics with Your Computer' making it relevant for modern programming students.