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 $s = 1$.

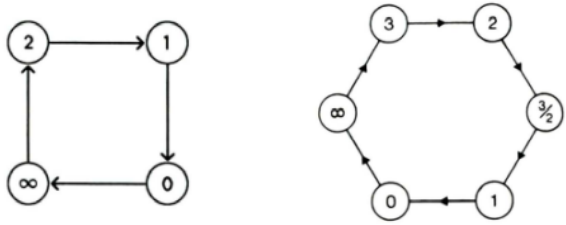
(4) For studying the iterates of rational functions, we adopt the definitions

$$\frac{\text{nonzero}}{0} = \infty, \quad \frac{\text{nonzero}}{\infty} = 0;$$

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

$$az = 1. \tag{6.7}$$

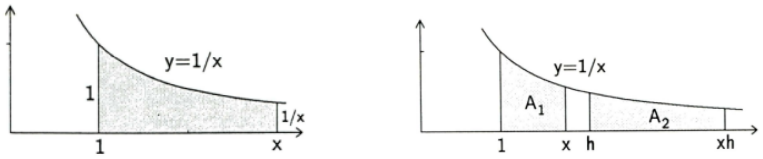
Here we replace the unknown constant z by a variable x. We try to drive the error y to 0 in the equation

$$ax = 1 - y \tag{6.8}$$

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

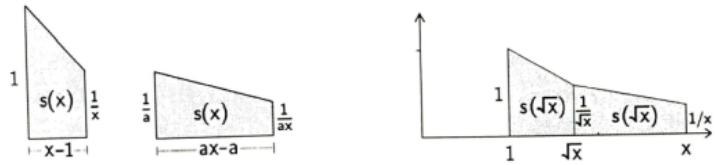
If we stretch the hyperbola $y = 1/x$ horizontally by a factor h and compress it vertically by the same factor, we get back to the curve $xy = 1$ since the product of the two coordinates of any point is unchanged. In Fig. 6.2b we see a region A_1 and the region A_2 resulting from it by stretching horizontally 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 $2u'$.

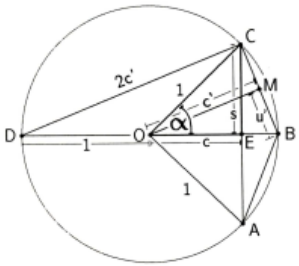


Figure 6.4. Computing α without trigonometry

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

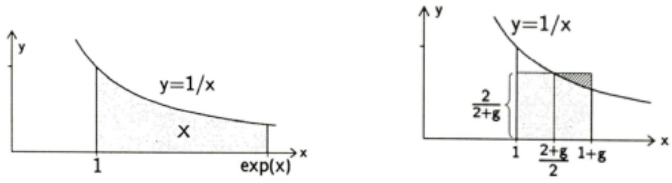
$$\exp 2x = (\exp x)^2. \tag{6.13}$$

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.

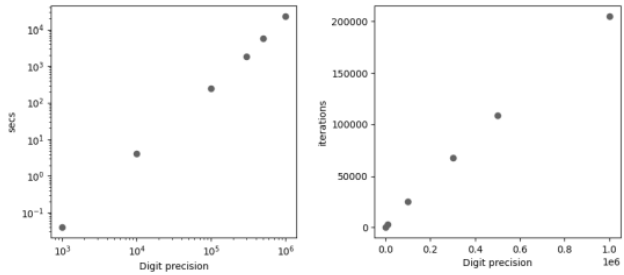


Figure 6.7. expD(1) running times and iterations

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.

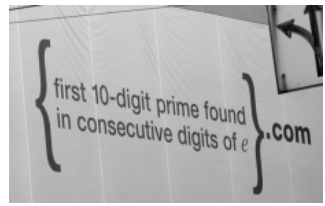


Figure 6.8. Google's billboard problem

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.

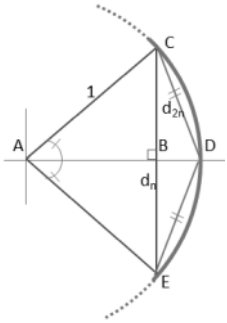


Figure 6.9. Archimedes' π approximation

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.

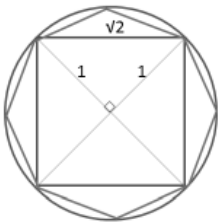


Figure 6.10. A square and octagon inside a unit circle

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

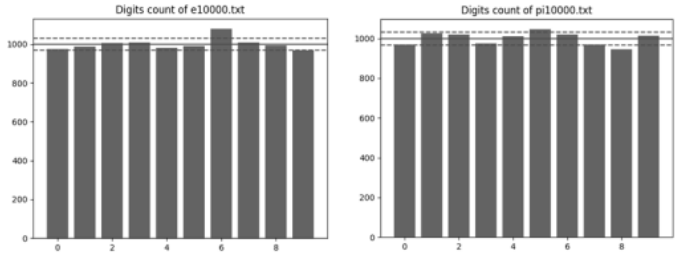


Figure 6.11. Digit counts for e and π

short programs (as we've seen), so according to this definition, their digits must be extremely non-random.

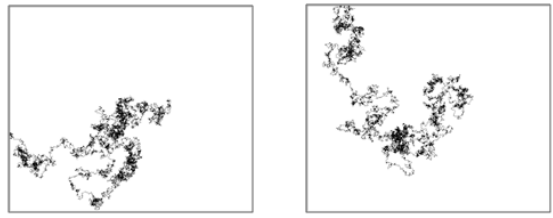
Chudnovsky believes that there must be hidden regularities in e and π, at least in some base. For instance, consider 9**(9**(9**9)) and 2**(2**(2**(2**(2**2)))). These huge numbers have very simple descriptions but their decimal representations look completely random. However, in base 9 the first number 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.

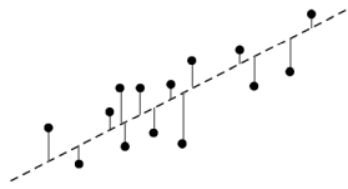


Figure 6.13. Using least-squares to fit a line to points

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.

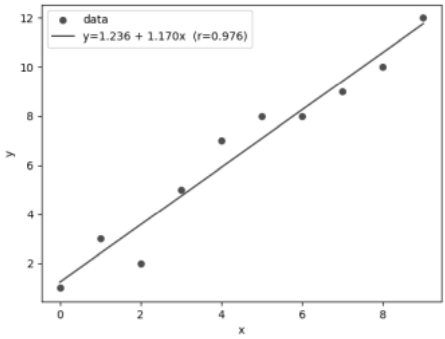


Figure 6.14. A best-fit line using least squares

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:

$$S = \sum (Y_i - y_i)^2 = \sum (k_0 + k_1 x_i + k_2 x_i^2 + \cdots + k_m x_i^m - y_i)^2$$

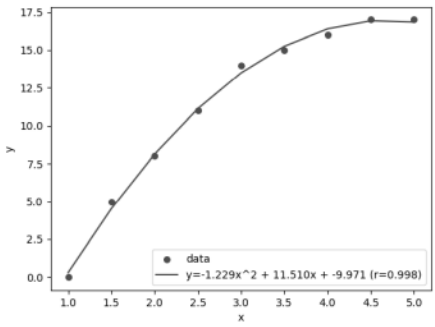


Figure 6.15. A best-fit parabola using least squares

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.

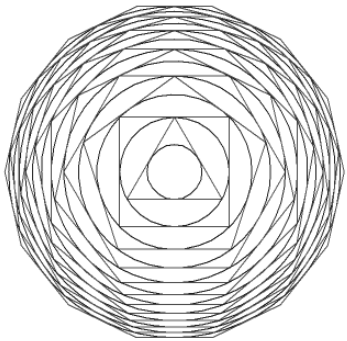
Fig. 6.16b simplifies this image to two circles and a (slightly rotated) equilateral triangle. The radii of the inner circle (the incircle) and the outer circle (the circumcircle) are related by the angle between the points where they touch the triangle – the inner circle touches half way along a side 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.

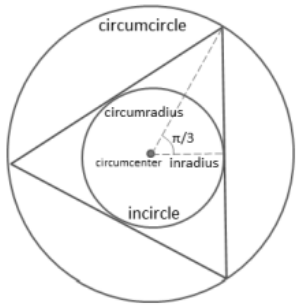
The first polygon, the triangle, requires $n = 3$, so we define $r(2) = 1$, which makes the inradius of the triangle's inner circle = 1.

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

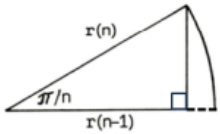


Figure 6.17. Generalizing the radius and angle

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.

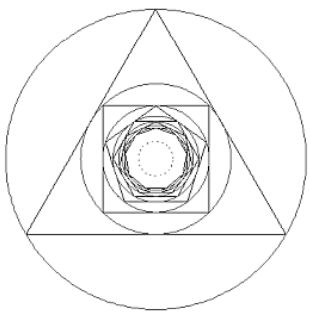


Figure 6.18. Inscribed polygons and circles

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.

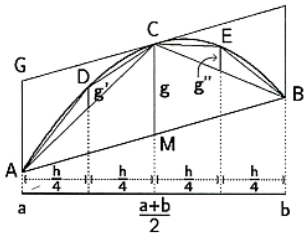


Figure 6.19. The area under a parabola

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).

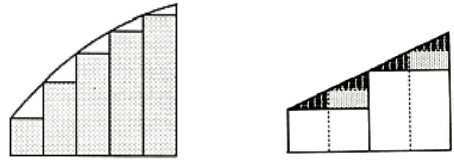
Archimedes also proved that the tangent at C is parallel to the chord AB, which has many consequences for the parabola's geometry.

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.

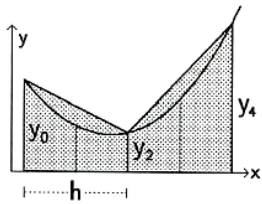


Figure 6.21. Trapezoidal area approximation

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.

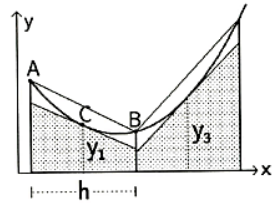


Figure 6.22. Midpoint trapezoidal area approximation

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:

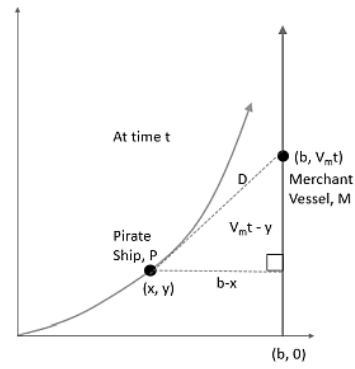


Figure 6.23. Pirates in pursuit

$$y = \frac{1}{2} b\left[\frac{1}{2}(1 - \frac{x}{b})^2 - \ln(1 - \frac{x}{b})\right] - \frac{1}{4} b \tag{6.43}$$

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
```

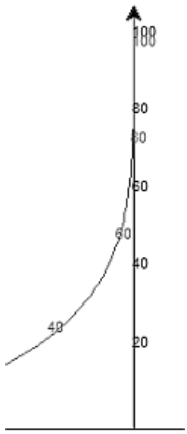


Figure 6.24. Pirates catch up

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
```

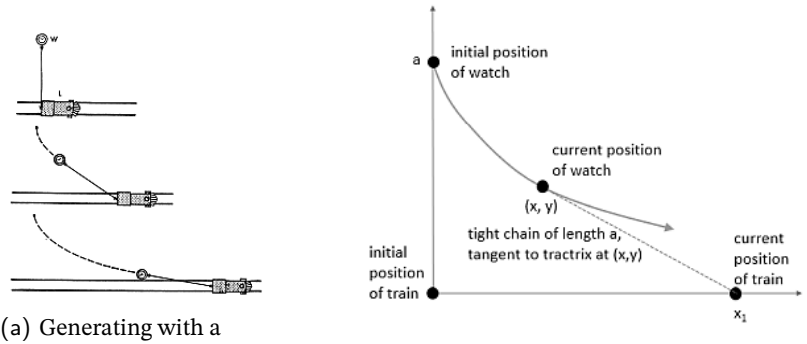
The distances between the step numbers printed in Fig. 6.25 are a rough indication that the separation of the two vessels is becoming constant. The final distance is 49.50, close to the theoretical answer $\frac{b}{2} = 50$.



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 $a = 2$.

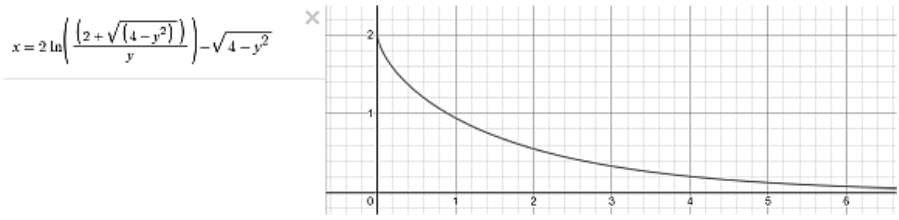


Figure 6.27. The Tractrix starting at $a = 2$

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.

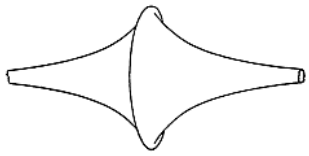


Figure 6.28. The pseudosphere

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 $\frac{\pi a^2}{2}$.

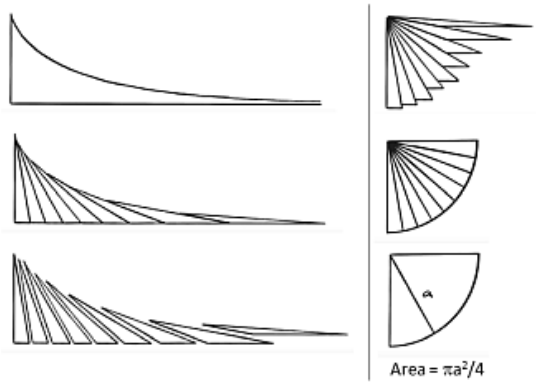


Figure 6.29. Using visual calculus for the tractrix area

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.

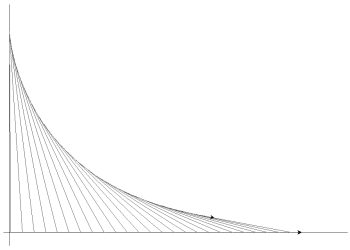


Figure 6.30. Turtles tractrix

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**].

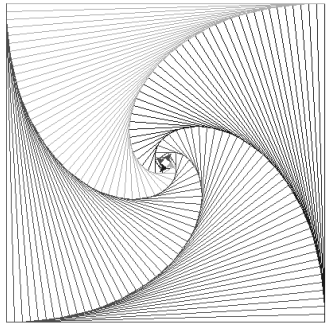


Figure 6.31. Four bugs meet

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.

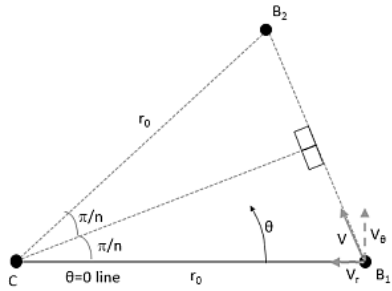


Figure 6.32. Two bugs in the n-bug problem

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 $b = 0.0001$.

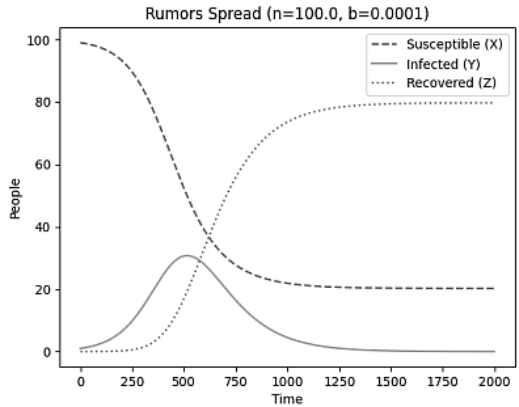


Figure 6.33. The rumor spreads

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.

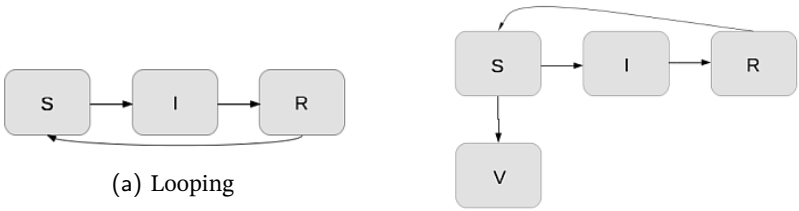
At another school where the flu has already spread, it was observed that at the start of a day there were 40 susceptibles and 8 infected, and the numbers were 30 and 18, respectively, 24 hours later. Using 1 hour as a time unit, $\beta = 10/(40 \cdot 8 \cdot 24)$. Why?

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:

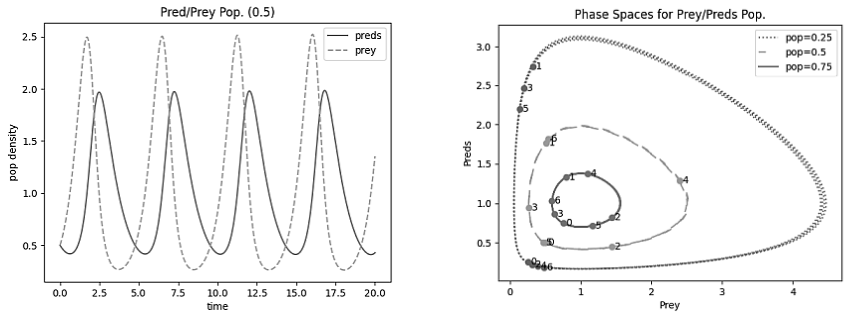
```
for i in range(N_t-1):
  prey[i+1] = prey[i] + dt*alpha*prey[i] -
                        dt*beta*prey[i]*preds[i]
  preds[i+1] = preds[i] + dt*delta*prey[i]*preds[i] -
                          dt*gamma*preds[i]
```

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 $f(3) = 50$.