13

Navigating the Earth

Why Study Spherical Trigonometry?

- **It is the natural geometry of the Earth**. Distances, bearings, angles, and areas on a sphere follow spherical, not planar, relations. Any problem involving global position, long distances, or directions across Earth must use spherical trigonometry for accuracy. Modern GPS and mapping still rely on spherical or ellipsoidal formulae.

- **It is central to astronomy**. The positions of stars, planets, and satellites are mapped onto a celestial sphere. Converting between coordinate systems (equatorial, ecliptic, galactic), computing angular separations, and predicting rising/setting all use spherical trigonometry.

13.1 Introduction

Spherical trigonometry is the branch of geometry dealing with triangles drawn on the surface of a sphere, a topic of great importance in astronomy and navigation, but often ignored in modern textbooks on trigonometry. This is a shame since it lends itself to practical examples especially if you're an amateur star-gazer or sailor, and offers a fresh perspective that reveals hidden commonalities with Euclidean geometry.

We'll be focusing on the use of spherical trigonometry for navigating over the Earth, but if you'd like a wider discussion that includes its astronomical origins, we strongly recommend *Heavenly Mathematics* by Glen Van Brummelen [**VB13**].

If you'd prefer something a bit shorter, try chapter 7 of the excellent *Mathematics for the Million* by Lancelot Hogben [**Hog93**] and/or chapter 13 of Eli Maor's *Trigonometric Delights* [**Mao20**].

13.2 Properties of the Sphere

The location of a point P on a sphere is specified in terms of *latitude* ϕ and *longitude* λ (see Fig. 13.1). These are defined with respect to a reference point M on the equatorial *great circle*. A great circle is any circular intersection of a sphere and a plane passing through the sphere's center O.

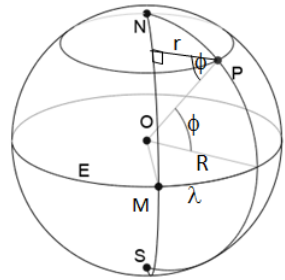


Figure 13.1. Locating a Point P

Given any point P, different from the poles (N and S), there's a unique great circle through P and the poles, and the half containing P is termed its *meridian*. The angle ϕ at the sphere's center between P and the equator is its latitude, and considered to be positive in the northern hemisphere and negative in the south. The meridian through M is the *prime meridian*.

A set of points with the same latitude is a *parallel*, and forms a small spherical circle over the surface centered on one of the poles. If the radius of the latitude circle is r and the equator's radius is R then $r = R\cos\phi$.

The angle λ around the equator from M to P's meridian is its longitude, which is positive to the east of M and negative to the west.

On the Earth, the prime meridian passes through Greenwich, England (see Fig. 13.2), placing New Orleans at 29° 57' 16.74" N and -90° 4' 30.252"' E, or (29.95465, -90.07507) as a decimal tuple. Of course, the Earth isn't a sphere, it's an ellipsoid or oblate spheroid, but we'll be ignoring that distinction since it simplifies the maths (at the expense of some accuracy).

The meridian located at 180°E is the same as that at 180°W, and runs almost entirely through the Pacific Ocean, coinciding for the most part with the international date line.

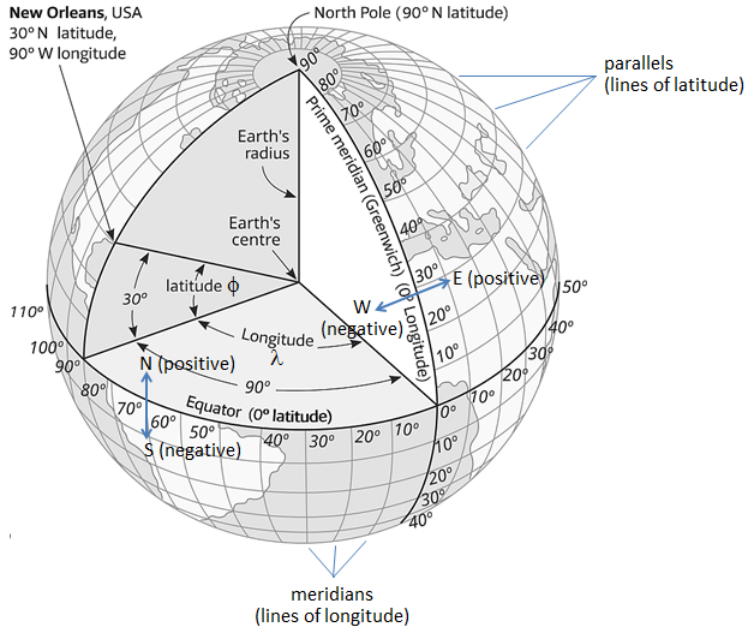


Figure 13.2. Locating a Point on Earth

13.3 Curves on a Sphere

There are two notable types of curves running over a sphere. We've already mentioned the great circle (Fig. 13.3(a)) which forms the shortest arc between any two points on its circle. This is obviously useful in navigation when we need to get from one point to another in the shortest time.

The second curve is the *rhumb line*, or loxodrome, which intersects all the meridians at the same angle (see Fig. 13.3(b)). Compass navigation is particularly simple along a rhumb line since the bearing never changes (e.g. it's θ at A, B, C, and Z). By contrast, the bearing for a great circle usually keeps changing as a journey progresses.

13.4 Mapping a Sphere

Euler proved in 1775 that it wasn't possible to map even a part of a sphere onto the plane without introducing some distortion. Nevertheless, cartographers had to try, and one way to classify their (many) attempts is based on the spherical properties that are preserved. Some of the more common ones are:

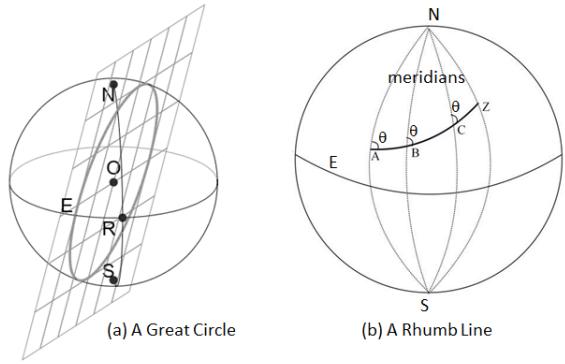


Figure 13.3. Curves Over a Sphere

- Preserving direction. This is only possible with respect to one or two chosen points going to every other point on the sphere, and the resulting map will vary significantly depending on the choice. One familiar example is the orthographic projection in Fig. 13.4.

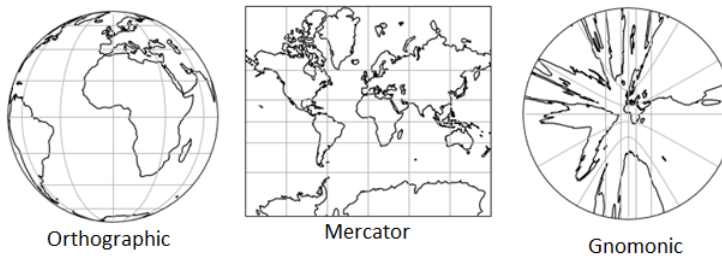


Figure 13.4. Orthographic, Mercator, and Gnomonic Projections

- Preserving Angle. *Conformal* projections preserve angles but not necessarily lengths, even though over small distances the shapes are fairly similar to those on the sphere. The Mercator map (Fig. 13.4) is the most popular, and has the significant benefit of allowing rhumb lines to be drawn as straight lines.
- Preserving area. An example is the Lambert cylindrical equal-area projection (see Fig. 13.5).

  Each non-polar point on the sphere is mapped to a point on a cylinder wrapped around its equator. As depicted in Fig. 13.6, a line is drawn horizontally through a given point on the sphere until it intersects the cylinder.

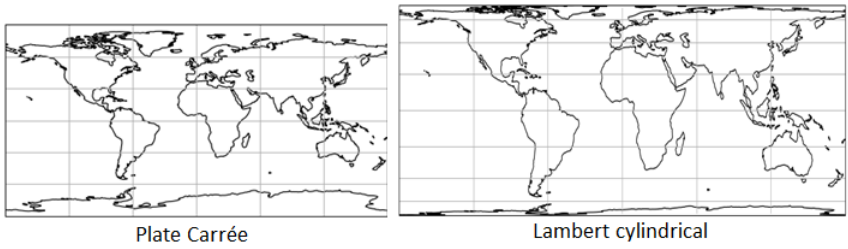


Figure 13.5. Plate Carrée and Lambert Cylindrical Projections

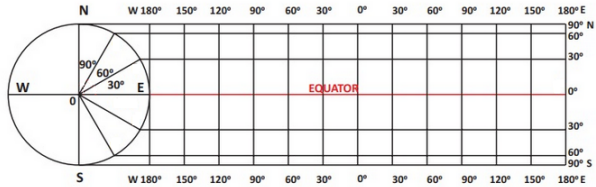


Figure 13.6. Lambert Cylindrical Equal Area Projection

- Preserving distance (equidistant). As with direction preserving projections, the mapping only works for a chosen point (or two) going to every other point. The most common map of this type is the Plate Carrée projection (see Fig. 13.5).
- Preserving the shortest route. The gnomonic projection (Fig. 13.4) allows great circles to be drawn as straight lines, but significantly distorts the surrounding map.

The examples in Figs. 13.4 and 13.5 were drawn with Cartopy (`https://scitools.org.uk/cartopy/docs/latest/`), a Python module that extends Matplotlib. We'll be using it throughout this section to illustrate the maths behind great circles and rhumb lines.

13.5 Spherical Triangles

Fig. 13.7 shows a spherical triangle ABC formed by the intersection of three great circles with O at the sphere's center. (We've borrowed this figure and the associated maths from Hogben [**Hog93**].)

The angles at A, B, and C are between the triangles's edges on the surface of the sphere, while the lengths (a, b, and c) can be viewed as the angles between the radial lines OA, OB, and OC. If the sphere has unit radius (as assumed in the right hand image in Fig. 13.7), then the side lengths are equal to the angles at O.

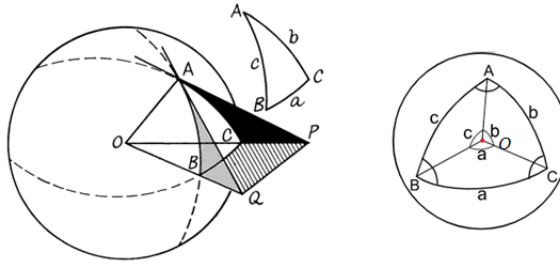


Figure 13.7. A Spherical Triangle ABC

In the left-hand image, we've extended OB and OC to OBQ and OCP such that AQ and AP are tangents to the great circles with arcs AB and AC. This means that OAQ and OAP are right angles, although it's hard to draw them as such.

The result is a triangular pyramid $OAPQ$, which is unfolded in Fig. 13.8 around the point A. Fig. 13.8 is a view of the resulting four triangles from above with OQP as the base of the pyramid.

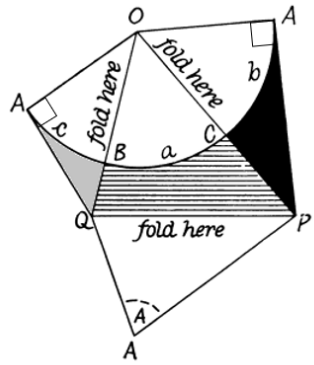


Figure 13.8. The Unfolded Pyramid $OAPQ$

The triangles can be related using Euclidean trigonometry, specifically its sine law:

$$PQ^2 = PO^2 + QO^2 - 2PO \cdot QO \cos a$$
$$PQ^2 = PA^2 + QA^2 - 2PA \cdot QA \cos A$$

Subtracting these equations:

$$(PO^2 - PA^2) + (QO^2 - QA^2) - 2PO \cdot QO \cos a + 2PA \cdot QA \cos A = 0$$

Utilizing Pythagoras, and some rearranging:

$$2PO \cdot QO \cos a = 2AO^2 + 2PA \cdot QA \cos A$$

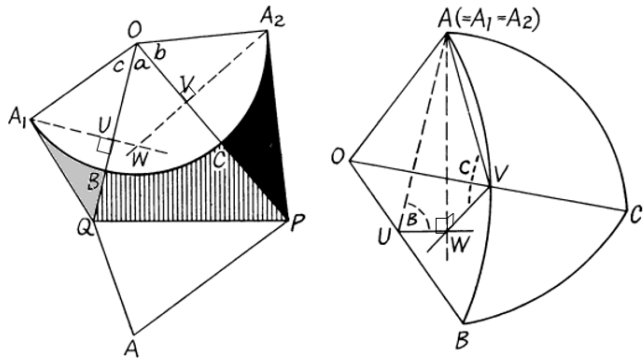
Divide through by $2PO \cdot QO$:

$$\begin{aligned} \cos a &= \frac{AO}{PO} \cdot \frac{AO}{QO} + \frac{PA}{PO} \cdot \frac{QA}{QO} \cos A \\ &= \cos POA \cos QOA + \sin POA \sin QOA \cos A \\ &= \cos b \cos c + \sin b \sin c \cos A \end{aligned}$$

This formula (the *spherical* cosine law) allows us to calculate a third side a, when we know the other two (b and c) and the included angle A. Similar reasoning can be used to provide expressions for calculating the other two sides.

13.6 The Spherical Sine Law

We can employ Fig. 13.8 to demonstrate another important spherical trigonometric formula. The four triangles are drawn in Fig. 13.9(a) with the versions of A named separately, and with a few new coordinates.



(a) Unfolded Pyramid OAPQ (b) Refolded Pyramid

Figure 13.9. The Unfolded and Refolded Pyramid

As the pyramid model is unfolded, A_1 rotates around OQ, and A_2 describes a similar arc about OP. Upon reaching the plane,

$$A_1UO = 90° = A_2VO$$

When the triangles are folded back into a pyramid in Fig. 13.9(b), A will be directly above W, and from Fig. 13.9(a) we see that:

$$\frac{A_1U}{A_1O} = \sin c, \quad \text{and} \quad \frac{A_2V}{A_2O} = \sin b$$

and since $A_1O = A_2O = AO$:

$$A_1U = AO \sin c \quad \text{and} \quad A_2V = AO \sin b$$

In Fig. 13.9(b), AUW is the angle between the planes that include the AB and BC arcs, which means that AUW equals the angle B of the spherical triangle. And so:

$$\frac{AW}{A_1U} = \sin B, \quad \text{and} \quad \frac{AW}{A_2V} = \sin C$$
$$AW = A_1U \sin B \quad \text{and} \quad AW = A_2V \sin C$$

A combination of these equalities produces:

$$AO \sin c \sin B = AW = AO \sin b \sin C$$

and so:

$$\frac{\sin c}{\sin C} = \frac{\sin b}{\sin B}$$

It's not difficult to show that these ratios are also equal to $\frac{\sin a}{\sin A}$.

13.7 Triangles made from Lunes

The lune in Fig. 13.10 is a slice of the surface of a sphere of radius R cut by two planes through meridians separated by an angle θ. The lune's surface area, $2R^2\theta$, is the total area of the sphere ($4\pi R^2$) multiplied by the angular fraction of the circumference $\theta/2\pi$. The volume of the wedge is $\frac{2}{3}R^3\theta$, which is obtained by multiplying the sphere's volume ($\frac{4}{3}\pi R^3$) by the same fraction.

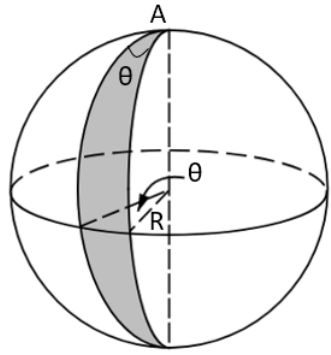


Figure 13.10. A Lune

The two planes that cut the meridians can be extended through the sphere, resulting in the double lune shown in Fig. 13.11. A and A' are located on opposite

sides of the sphere (A' is the *antipodal* point of A), and the shaded area is double that of the original lune, so equals $4R^2\theta$.

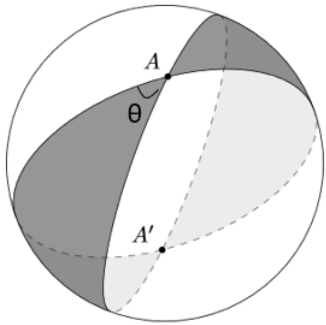


Figure 13.11. A Double Lune

The spherical triangle ABC in Fig. 13.12 is constructed by overlapping three double lunes (D_A, D_B, and D_C) located at A, B, and C.

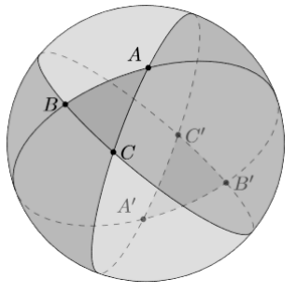


Figure 13.12. A Spherical Triangle made from three Double Lunes

The sum of the areas of the double lunes, denoted by area(D_A) + area(D_B) + area(D_C), is the area of the entire sphere and, due to overlapping, two extra copies of the areas of ABC and $A'B'C'$. The areas of these two triangles are the same, and so:

$$\text{area}(D_A) + \text{area}(D_B) + \text{area}(D_C) = 4\pi R^2 + 4T$$

where T is the area of the spherical triangle ABC. So:

$$4R^2A + 4R^2B + 4R^2C = 4\pi R^2 + 4T$$

Therefore, the area of the spherical triangle is

$$T = R^2(A + B + C - \pi)$$

It is a small step from this to determine that the the sum of the angles $A+B+C$ of *any* spherical triangle is always greater than 180° (or π radians). Specifically: any spherical triangle must occupy some portion of the sphere's surface, and so its area must be positive: $T > 0$. This implies that $A + B + C - \pi > 0$, and so $A + B + C > \pi$.

This property can be utilized to argue for Euclid's theorem – that it's impossible to map any portion of a sphere onto the plane without introducing some distortion. If we decompose the shape into one of more spherical triangles then it's impossible for their angular sum to equal those of the mapped flat triangles.

13.8 Are Great Circle Arcs the Shortest Distance?

Our intuition tells us that a great circle arc between two points is the shortest path, but let's prove it. We'll choose two arbitrary points and show that the great circle connecting them is shorter than every possible other path. This approach is based on one presented in Feeman's *Portraits of the Earth* [**Fee00**].

We'll assume the sphere (see Fig. 13.13) has unit radius and employs an (x, y, z) coordinate system centered around O, and the polar coordinates are (ρ, λ, ψ). ρ is the length of the point's vector, λ corresponds to its longitude, while ψ is its inclination, which is equivalent to $\pi/2 - \phi$ (ϕ is the point's latitude).

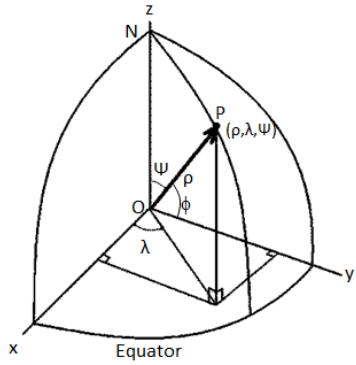


Figure 13.13. 3D Polar and Cartesian Coordinates

The Cartesian coordinates can be expressed as:

$$x = \rho \sin\psi \cos\lambda, \quad y = \rho \sin\psi \sin\lambda \quad z = \rho \cos\psi$$

It's more convenient for our needs to replace ψ by the latitude ϕ:

$$\begin{array}{rcl} x & = & \rho\sin(\pi/2-\phi)\cos\lambda \quad = \quad \rho\cos\phi\cos\lambda \\ y & = & \rho\sin(\pi/2-\phi)\sin\lambda \quad = \quad \rho\cos\phi\sin\lambda \\ z & = & \rho\cos(\pi/2-\phi) \qquad\quad = \quad \rho\sin\phi \end{array}$$

We'll use the north pole as the starting point, which simplifies the calculations. P will have latitude λ and longitude ϕ, and the latitude at the north pole is $\pi/2$ and its longitude is undefined.

The great circle arc from the north pole to P will follow P's meridian at longitude λ. The arc will travel an angle of $\pi/2-\phi$, which is also the meridian arc's length since the sphere has unit radius.

Now its time to consider some other path connecting these same points and show that its length is greater than (or equal to) $\pi/2-\phi$. During the trip over the sphere from the north pole to P, the latitude and longitude keep changing. This can be expressed by two functions, $\phi(t)$ and $\lambda(t)$, that return the current latitude and longitude at a given time t.

The total path length will be computed using the integral of a speed function with respect to the trip duration. This function can be obtained via the derivative of the distance vector.

At time t the latitude is $\phi(t)$ and longitude is $\lambda(t)$. In 3D rectangular coordinates, this means that the position at time t is

$$\begin{array}{rcl} p(t) & = & (x,y,z) \\ & = & (\cos\phi(t)\cos\lambda(t),\ \cos\phi(t)\sin\lambda(t),\ \sin\phi(t)). \end{array}$$

The velocity vector at time t is the coordinate-by-coordinate derivative of the position vector:

$$\begin{array}{rcl} p'(t) & = & (-\sin\phi(t)\dfrac{d\phi}{dt}\cos\lambda(t) - \cos\phi(t)\sin\lambda(t)\dfrac{d\lambda}{dt}, \\ & & -\sin\phi(t)\dfrac{d\phi}{dt}\sin\lambda(t) + \cos\phi(t)\cos\lambda(t)\dfrac{d\lambda}{dt}, \\ & & \cos\phi(t)\dfrac{d\phi}{dt}). \end{array}$$

The speed at time t is the magnitude of the velocity vector, which is the square root of the sum of the squares of the coordinates of the velocity vector. After some refactoring, this simplifies to

$$\|p'(t)\| = \sqrt{\left(\frac{d\phi}{dt}\right)^2 + \cos^2\phi(t)\left(\frac{d\lambda}{dt}\right)^2}$$

We'll assume that the trip from the north pole began at time $t=0$ and concluded at time t_1. Thus, $\phi(t_1)=\phi$, $\lambda(t_1)=\lambda$, and $\phi(0)=\pi/2$. (The value of $\lambda(0)$ isn't

defined, but it doesn't matter.) The total distance travelled is:

$$L = \int_{t=0}^{t_1} \|p'(t)\|\, dt.$$

But:

$$\|p'(t)\| \geq \sqrt{\left(\frac{d\phi}{dt}\right)^2} = \left|\frac{d\phi}{dt}\right| \geq -\frac{d\phi}{dt}.$$

It follows that:

$$L \geq \int_{t=0}^{t_1} -\frac{d\phi}{dt} dt = \phi(0) - \phi(t_1) = \pi/2 - \phi$$

The right-hand side of this inequality is the length of the great circle connecting the north pole and P. Thus, we can conclude that the great circle route is indeed the shortest possible.

13.9 Spherical Triangles Summarized

Table 13.1 summarizes the three main results for spherical triangles, and compares them to their equivalents for Euclidean triangles. The triangles are shown in Fig. 13.14.

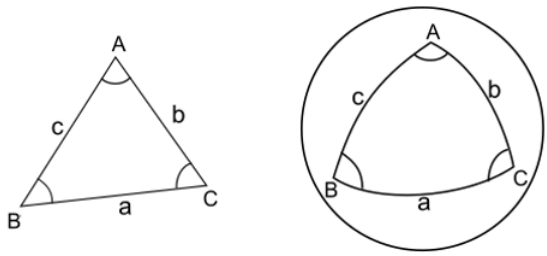


Figure 13.14. Euclidean and Spherical Triangle

13.10 Calculating with Great Circles

Calculations involving great circles typically fall into three areas: finding the great circle distance between two points, determining the initial bearing to travel from one point to another, and obtaining the destination coordinate if only the start point, bearing, and travel distance are supplied.

The best source for code for these calculations is the Movable Type Scripts web page "Calculate distance, bearing and more between Latitude/Longitude

Property	Euclidean Triangle	Spherical Triangle
Law of Cosines	$c^2 = a^2 + b^2 - 2ab\cos C$	$\cos c = \cos a \cos b + \sin a \sin b \cos C$
Law of Sines	$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$	$\frac{\sin a}{\sin A} = \frac{\sin b}{\sin B} = \frac{\sin c}{\sin C}$
Sum of Triangle Angles	$A + B + C = \pi$	$A + B + C > \pi$

Table 13.1. Comparison of Triangle Laws in Euclidean and Spherical Geometries

points" (`https://www.movable-type.co.uk/scripts/latlong.html`), which offers both formulae and JavaScript functions. Although the page assumes the Earth is spherical (as we've done), formulae for calculating distances on ellipsoids is available separately at `https://www.movable-type.co.uk/scripts/latlong-vincenty.html`.

Our Python functions in `nav.py` are closely based on Movable Type Scripts' JavaScript, so the emphasis here will be on explaining how the spherical cosine and sine laws are manipulated to get the formulae used by those functions.

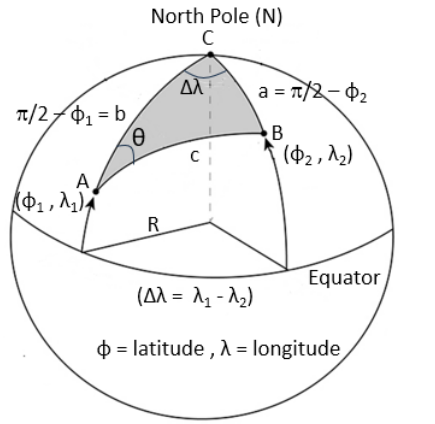


Figure 13.15. A Spherical Triangle for Calculations

Most great circle equations can be derived from the spherical triangle in Fig. 13.15. A is the trip's starting point at latitude ϕ_1, longitude λ_1, with destination B at latitude ϕ_2, longitude λ_2. The third point, C, is located at the North pole

(latitude $\pi/2$) making AC and BC meridian arcs, and the side angles b and a *co-latitudes*, $\pi/2-\phi_1$ and $\pi/2-\phi_2$. We'll treat the sphere as having a unit radius, but when it comes time to convert sides to distances, we'll assume the Earth's radius is $R = 6371$ km.

Since C is positioned at the North pole, its angle is the difference between A and B's longitudes ($\Delta\lambda = \lambda_1 - \lambda_2$). We'll occasionally refer to the bearing of someone leaving A to travel to B as θ, which is equivalent to the angle A, at least initially. As mentioned before, a drawback of great circle navigation is that this bearing will change as the traveler moves towards B.

The required distance is c, which suggest the use of the cosine law:

$$\cos c = \cos a \cdot \cos b + \sin a \cdot \sin b \cdot \cos C$$

It can be rephrased to use latitudes:

$$\begin{aligned} \cos c &= \cos(\pi/2 - \phi_2) \cdot \cos(\pi/2 - \phi_1) + \sin(\pi/2 - \phi_2) \cdot \sin(\pi/2 - \phi_1) \cdot \cos C \\ &= \sin\phi_2 \cdot \sin\phi_1 + \cos\phi_2 \cdot \cos\phi_1 \cdot \cos C \end{aligned}$$

13.10.1 Distance. Given the coordinates of two points, the great circle distance between them is obtained using nav.distTo() from nav.py.

```
def distTo(lat1, lon1, lat2, lon2):
  phi1 = math.radians(lat1)
  phi2 = math.radians(lat2)
  dLambda = math.radians(lon2 - lon1)

  # Spherical law of cosines
  angle = math.acos(
    math.sin(phi1) * math.sin(phi2) + \
    math.cos(phi1) * math.cos(phi2) * math.cos(dLambda)
  )
  return EARTH_RADIUS * angle
```

The latitude and longitudes are supplied as decimal degrees, and the distance is returned in kilometers. In the code, `angle` corresponds to the spherical triangle side c.

13.10.2 The Haversine Distance. The *haversine* formula was a once-popular alternative for finding the distance since it's less susceptible to rounding errors when the distance is small. Our nav.havDistTo() is:

```
def havDistTo(lat1, lon1, lat2, lon2):
  phi1 = math.radians(lat1)
  phi2 = math.radians(lat2)
  deltaPhi = math.radians(lat2 - lat1)
  dLambda = math.radians(lon2 - lon1)

  h = math.sin(deltaPhi/2)**2 + \
```

```
        math.cos(phi1) * math.cos(phi2) * \
        math.sin(dLambda/2)**2
    c = 2 * math.atan2(math.sqrt(h), math.sqrt(1 - h))
    return EARTH_RADIUS * c
```

In distTo(), math.acos() returns the angular distance c (see Fig. 13.15), but if the distance is small then accuracy may be lost, especially if we're relying on cosine tables rather than a computer.

In havDistTo(), the corresponding operation is the job of math.atan2(), and the inverse tangent of a number can be nicely approximated by the number itself, resulting in much less loss of precision.

That being said, modern programming languages (including Python) offer roughly 15 significant figures of precision for floats, which means that distTo() and havDistTo() produce equivalent distances down to a few meters. We checked this with `havDiff.py` which calculates distances using both approaches in the range 1 to 1000 meters in steps of 5. A graph showing the absolute differences between their values is shown in Fig. 13.16.

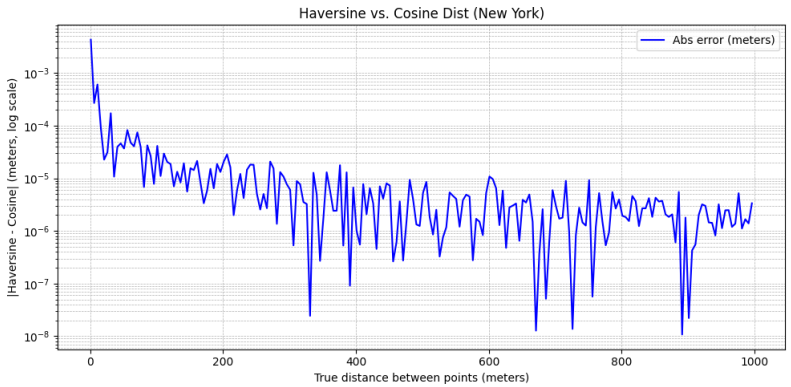


Figure 13.16. Haversine vs. Cosine Distance

The small errors are acceptable especially when our assumption that the Earth is a sphere introduces much greater errors. For example, the Earth's radius varies from 6,336 km (equatorial meridian) to 6,399 km (at the poles), whereas we're fixing it at 6,371 km. This means that calculated distance can be in error by up to 0.3%, or roughly 22 km, depending on the points in question.

The basic haversine formula is:

$$\text{haver}(x) = \sin^2(x/2) = \frac{1 - \cos x}{2}$$

The '(re)versed sine' is $1 - \cos x$ and the 'half-versed-sine' is $(1 - \cos x)/2$. After a little manipulation based around a spherical triangle such as the one in Fig.

13.14, the *law of haversine* is:

$$\text{haver}(c) = \text{haver}(a-b) + \sin a \sin b \, \text{haver}(C)$$

Utilizing the co-latitudes for a and b (see Fig. 13.15), this becomes:

$$\text{haver}(c) = \text{haver}(\phi_1 - \phi_2) + \cos\phi_2 \cos\phi_1 \, \text{haver}(\Delta\lambda)$$

The two uses of `haver(x)` on the right-hand side can be replaced by the $\sin^2(x/2)$ definition:

$$\text{haver}(c) = \sin^2 \Delta\phi + \cos\phi_2 \cos\phi_1 \sin^2 \Delta\lambda$$

where $\Delta\phi = \phi_1 - \phi_2$.

Let $h = \sin^2 \Delta\phi + \cos\phi_2 \cos\phi_1 \sin^2 \Delta\lambda$, and we want $c = \text{haver}^{-1}(h)$. Instead of directly inverting the haversine, we use the numerically stable identity involving the arc-tangent:

$$\begin{aligned} c &= \text{haver}^{-1}(h) \\ &= 2 \cdot \arcsin(\sqrt{h}) \\ &= 2 \cdot \arctan(\tfrac{\sqrt{h}}{\sqrt{1-h}}) \\ &= 2 \cdot \operatorname{arctan2}(\sqrt{h}, \sqrt{1-h}) \end{aligned}$$

13.10.3 Bearing. Given the coordinates of two points, A and B, the initial bearing from A to B is obtained by calling nav.bearingTo() from `nav.py`:

```
def bearingTo(lat1, lon1, lat2, lon2):
  # return degree bearing from first to second coord
  phi1 = math.radians(lat1)
  phi2 = math.radians(lat2)
  dLambda = math.radians(lon2 - lon1)

  y = math.sin(dLambda) * math.cos(phi2)
  x = math.cos(phi1) * math.sin(phi2) - \
      math.sin(phi1) * math.cos(phi2) * math.cos(dLambda)
  theta = math.atan2(y, x)
  return (math.degrees(theta) + 360) % 360
```

The bearing is returned in degrees in the range 0°...360°, with negative values mapped to the range 180° – 360°.

The function returns the angle at A (see Fig. 13.15), by taking the arc-tangent of y/x ($\tan A$). This division is best thought of as $\sin A/\cos A$ so y can be equated with $\sin A$ and x with $\cos A$.

y ($\sin A$) is obtained via the sine law:

$$\frac{\sin a}{\sin A} = \frac{\sin b}{\sin B} = \frac{\sin c}{\sin C}$$

a is rephrased as a co-latitude, and C as the longitude delta:

$$\frac{\sin(\pi/2 - \phi_2)}{\sin A} = \frac{\sin c}{\sin \Delta\lambda}$$

so:

$$\sin A = \frac{\sin \Delta\lambda \cdot \cos \phi_2}{\sin c}$$

For the determination of x ($\cos A$), start with the cosine law for a:

$$\cos a = \cos b \cdot \cos c + \sin b \cdot \sin c \cdot \cos A$$

Rearranging:

$$\begin{aligned}
\sin b \cdot \sin c \cdot \cos A &= \cos a - \cos b \cdot \cos c \\
&= \cos a - \cos b \cdot (\cos a \cdot \cos b + \sin a \cdot \sin b \cdot \cos C) \\
&= \cos a - \cos^2 b \cdot \cos a - \sin a \cdot \sin b \cdot \cos b \cdot \cos C \\
&= \cos a - (1 - \sin^2 b) \cdot \cos a - \sin a \cdot \sin b \cdot \cos b \cdot \cos C \\
&= \sin^2 b \cdot \cos a - \sin a \cdot \sin b \cdot \cos b \cdot \cos C
\end{aligned}$$

Divide through by $\sin b \cdot \sin c$:

$$\cos A = \frac{\sin b \cdot \cos a - \sin a \cdot \cos b \cdot \cos C}{\sin c}$$

Rewrite a and b as co-latitudes:

$$\begin{aligned}
\cos A &= \frac{\sin(\pi/2 - \phi_1) \cdot \cos(\pi/2 - \phi_2) - \sin(\pi/2 - \phi_2) \cdot \cos(\pi/2 - \phi_1) \cdot \cos C}{\sin c} \\
&= \frac{\cos \phi_1 \cdot \sin \phi_2 - \cos \phi_2 \cdot \sin \phi_1 \cdot \cos \Delta\lambda}{\sin c}
\end{aligned}$$

Combine these $\sin A$ and $\cos A$ equations:

$$\frac{y}{x} = \tan A = \frac{\sin A}{\cos A} = \frac{\sin \Delta\lambda \cdot \cos \phi_2}{\cos \phi_1 \cdot \sin \phi_2 - \cos \phi_2 \cdot \sin \phi_1 \cdot \cos \Delta\lambda}$$

This numerator and denominator are implemented in the code for `bearingTo()`.

13.10.4 Calculating a Destination. nav.destCoord() is supplied with a starting point, an initial bearing, a distance to travel, and returns the destination.

```
def destCoord(lat1, lon1, dist, bearing):
  theta = math.radians(bearing)
  phi1 = math.radians(lat1)
  lambda1 = math.radians(lon1)
  delta = dist/EARTH_RADIUS

  phi2 = math.asin(math.sin(phi1) * math.cos(delta) + \
             math.cos(phi1) * math.sin(delta) * \
             math.cos(theta)  )

  lambda2 = lambda1 + \
       math.atan2(math.sin(theta) * math.sin(delta) * \
```

```
                math.cos(phi1),
                math.cos(delta) - math.sin(phi1) *  \
                math.sin(phi2)  )

    lambda2 = (lambda2 + math.pi) % (2 * math.pi) - math.pi
    return math.degrees(phi2), math.degrees(lambda2)  # lat, lon
```

With reference to Fig. 13.15, the code is calculating ϕ_2 and λ_2 of point B. The latitude, ϕ_2, is obtained via the cosine law for a:

$$\cos a = \cos b \cdot \cos c + \sin b \cdot \sin c \cdot \cos A$$

Substitute the co-latitudes for a and b:

$$\begin{aligned} \cos(\pi/2 - \phi_2) &= \cos(\pi/2 - \phi_1) \cdot \cos c + \sin(\pi/2 - \phi_1) \cdot \sin c \cdot \cos A \\ \sin \phi_2 &= \sin \phi_1 \cdot \cos c + \cos \phi_1 \cdot \sin c \cdot \cos A \end{aligned}$$

The longitude is:

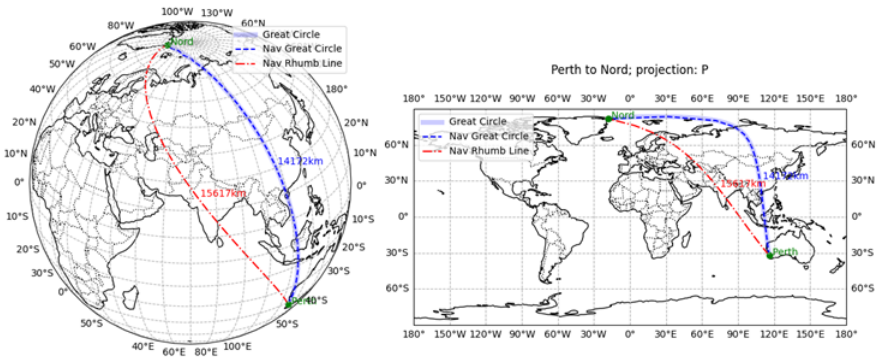
$$\lambda_2 = \lambda_1 + \arctan(\tan \Delta\lambda)$$

or

$$\lambda_2 = \lambda_1 + \arctan(\tan C)$$

The tangent can be written as $\sin C / \cos C$, which splits the problem into two parts. The next steps are quite similar to those used in the bearing formula above, so won't be explained again.

13.10.5 Drawing the Great Circle. showMap.py utilizes the great circle functions in nav.py to offer the user a choice of four map projections for displaying two great circles and a rhumb line connecting Perth, Australia to Nord, Greenland. Fig. 13.17 shows the Orthographic and Platte Carrée options.



(a) Orthographic Projection (b) Platte Carree Projection

Figure 13.17. Orthographic and Platte Carrée Trips from Perth to Nord

Although the curves appear differently on the two maps, a study of the countries they pass over reveals them to be the same. The distances – 14172 km for the great circles, and 15617 km for the rhumb line – are also the same. We'll discuss rhumb lines in the next section, and focus here on how the great circles are drawn.

The solid curve, labeled 'Great Circle' on the map, is drawn using the Cartopy plot() function:

```
plt.plot([lon1, lon2], [lat1, lat2],
     color='blue', lw=5, alpha=0.2,
     label="Great Circle", transform=ccrs.Geodetic())
```

It only requires the coordinates of the two end points, and uses a Geodetic() transform to interpolate great circle coordinates between those points for the plotting. We've included this code mainly as a check that the coordinates generated by our nav.greatCircleCoords() are correct. That function is called like so:

```
coords = nav.greatCircleCoords(lat1, lon1, lat2, lon2, GCStep)
print("\nGreat Circle (lat,lon)s and deg Bearing:")
for coord in coords:
  cLat, cLon, cBear = coord
  print(f"  ({cLat:.2f}, {cLon:.2f}); {cBear:.2f}")
```

The returned list contains triplets of the form (latitude, longitude, bearing); the most interesting aspect of the data is that the bearing changes at each point along the line (as you'd expect with great circles):

```
  :
(43.94, 105.62); 351.10
(50.23, 104.09); 349.97
(56.49, 102.08); 348.36
(62.71, 99.28); 345.94
(68.83, 95.00); 342.02
(74.77, 87.50); 334.89
(80.17, 71.48); 319.23
  :
```

The data is passed to Cartopy's plot():

```
lats, lons, _ = zip(*coords)
plt.plot(lons, lats, color='blue', linestyle='--',
         label="Nav Great Circle",
         transform=ccrs.PlateCarree())
```

The key difference here is that the PlateCarree() transform connects adjacent points in the list with straight lines without generating intermediate points.

nav.greatCircleCoords() produces its list by first calculating the total distance to the destination with nav.distTo(), from which it obtains a step distance between points. A loop calls nav.destCoord() to move forward by that step using

the current bearing, then nav.bearingTo() updates the bearing for the new location, and the loop repeats:

```
def greatCircleCoords(lat1, lon1, lat2, lon2, num):
  lat1 = parseCoord(lat1)
  lon1 = parseCoord(lon1)
  lat2 = parseCoord(lat2)
  lon2 = parseCoord(lon2)

  dist = distTo(lat1, lon1, lat2, lon2)
  distStep = dist/num

  bearing = bearingTo(lat1, lon1, lat2, lon2)
  nSteps = 1
  coords = [(lat1, lon1, bearing)]
  currLat, currLon = lat1, lon1
  while (nSteps < num) or \
        (not areClose(currLat, currLon, lat2, lon2)):
    destLat, destLon =  \
       destCoord(currLat, currLon, distStep, bearing)
    bearing = bearingTo(destLat, destLon, lat2, lon2)
    nSteps += 1
    coords.append((destLat, destLon, bearing))
    currLat, currLon = destLat, destLon
  return coords
```

The call to areClose() adds a second test to end coordinate generation if the current position is sufficient close to the destination.

13.11 The Mercator Projection

Sailors have plotted courses using rhumb lines and compass headings since the thirteenth century. Mathematicians call these lines loxodromes, from the Latin *loxos* - slant and *drome* - running, because they cut meridians at the same angle. One consequence is that as a rhumb line heads north (or south), and the meridians get closer together, it turns steadily toward the pole, spiraling around it as in Fig. 13.18.

The curve in Fig. 13.18 was drawn using nav.rhumbDestCoord() which is described in section 13.12.3 below. The complete code is available in NYCSpiral.py.

Sailors initially tried to plot rhumb lines on planar charts (i.e. maps where the lines of latitude are equally spaced) but the position errors gradually increase with the line's length, a problem finally solved by Gerhardus Mercator (1512–1594). His map was still laid out on an easy-to-understand rectangular grid: all the circles of latitude were represented by horizontal lines parallel to the equator

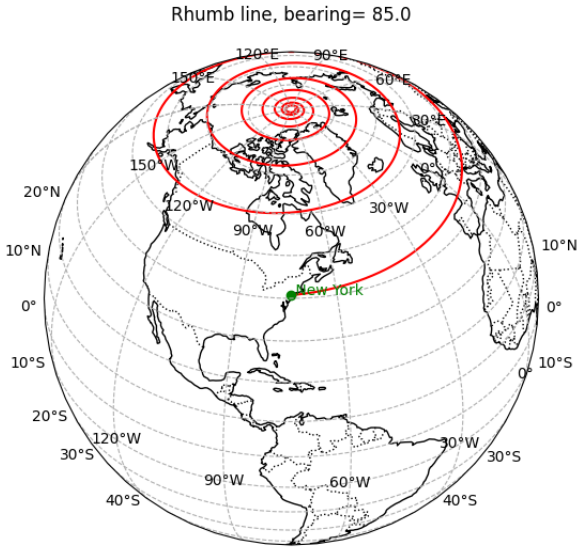


Figure 13.18. A Rhumb Line Bearing of 85° from New York City

and equal to it in length, and all the meridians were drawn as vertical lines perpendicular to the equator. But the important change was that the projection was *conformal* – the map preserved the angles between all the points on the globe.

On a globe, the parallels decrease in size at higher latitudes, until they shrink to a point at either pole. Mercator's map draws these circles as horizontal lines of equal length, which means that each parallel has to be stretched horizontally by a factor depending on its latitude.

Fig. 13.19 shows a parallel at latitude ϕ, which has a radius of $R\cos\phi$ and circumference $2\pi R\cos\phi$ on the globe.

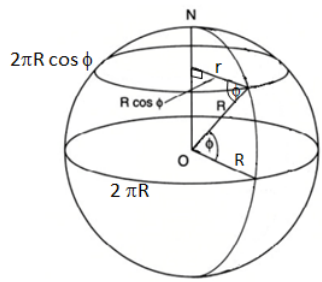


Figure 13.19. Lengths for Latitude ϕ

On Mercator's map this parallel must be stretched to be the same length as the equator, $2\pi R$, which requires it to be multiplied by the factor $\frac{2\pi R}{2\pi R \cos\phi} = \sec\phi$. Note that this stretching is a function of ϕ: the higher the latitude, the greater the ratio.

Mercator's big idea was that in order for the map to be angle preserving, the east-west stretch of the parallels had to be accompanied by an *equal* north-south stretch of the spacings between the parallels, implying that they had to progressively increase at higher latitudes. In other words, equally spaced latitudes along each meridian on a globe will gradually increase on the map (see Fig. 13.20).

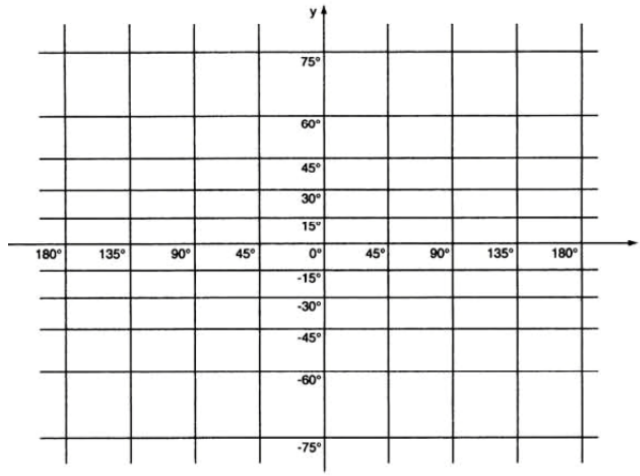


Figure 13.20. The Mercator Grid

Fig. 13.20 also highlights the map's main shortcoming: distances are hard to measure because regions far to the north and south are grossly exaggerated in size (e.g. compare Greenland to Africa on a globe and on a Mercator map).

Mercator first published his map in 1569 with the succinct title *New and Improved Description of the Lands of the World, Amended and Intended for the Use of Navigators*. It was massive, printed across 18 sheets, each measuring 33 by 40 cm (`https://en.wikipedia.org/wiki/Mercator_1569_world_map`).

Mercator never gave a full account of how he had progressively increased the distances between the parallels. It was left to Edward Wright in 1599 to present the mathematical details in *Certaine errors in navigation, arising either of the ordinarie erroneous making or vsing of the sea chart, compasse, crosse stafie, and tables of declination of the sunne, and fixed starres detected and corrected*. He's also devised the 'bladder' explanation of the projection, responsible for much mirth in school classrooms ever since.

Wright asks us to imagine a spherical bladder representing the Earth, with meridians and parallels drawn on its surface. It's placed inside a vertical cylinder representing the map projection, preferably made of glass so you can see the projection develop (see Fig. 13.21).

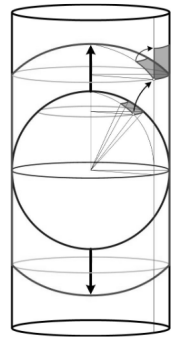


Figure 13.21. Wright's Expanding Bladder in a Cylinder

Inflate the bladder slowly so that its vertical axis aligns with the axis of the cylinder until its equator line touches the cylinder walls. This sets the equator on the bladder and cylinder to be the same. Then continue slowly inflating the bladder so that it stretches by the same amount in all directions – this enforces Mercator's angle-preserving condition. As a consequence, the parallels on the bladder grow farther and farther apart, but because the cylinder is open ended, the bladder's poles never touch the cylinder's sides.

A mathematical analysis of the process is outlined by Jim Stevenson in *Mercator Projection Balloon*, online at `https://josmfs.net/wordpress/wp-content/uploads/2018/12/Mercator-Balloon-170306.pdf`.

Fig. 13.22 shows the projection problem: the spherical rectangle on the globe, with width $(R\cos\phi)\Delta\lambda$ and height $R\Delta\phi$, must be mapped to a planar rectangle of width Δx and height Δy.

Fig. 13.23 depicts the two rectangles (spherical and planar) in more detail. The spherical rectangle is defined by meridians of longitudes λ_1 and $\lambda_1 + \Delta\lambda$, and parallels of latitudes ϕ_1 and $\phi_1 + \Delta\phi$. The sides of this rectangle have length $(R\cos\phi_1)\Delta\lambda$ and $R\Delta\phi$ respectively. The planar rectangle is delimited by the lines x_1, $x_1 + \Delta x$, y_1, and $y_1 + \Delta y$, with sides $R\Delta x$ and $R\Delta y$.

The compass bearing from P on the globe is at the angle ψ, and since the projection is angle preserving this is also true for the bearing from P' in the projection.

Since the meridians in the Mercator projection are vertical straight lines, this implies that $\Delta x = \Delta\lambda$, the angular distance (or delta) between the two longitudes at the equator.

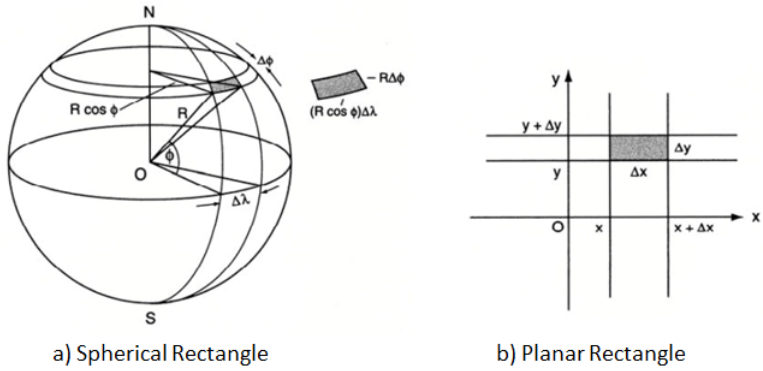


Figure 13.22. The Mercator Projection of a Rectangle

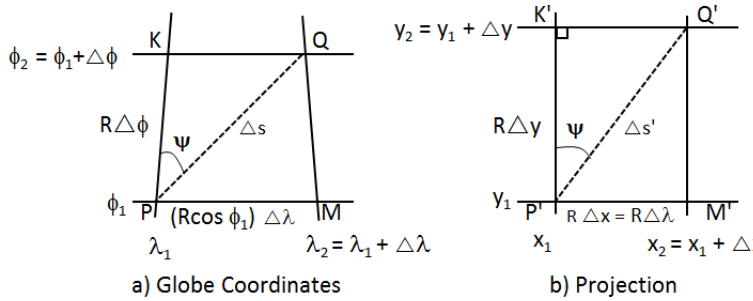


Figure 13.23. The Spherical and Planar Rectangles

The scaling in the x-direction is therefore:

$$\frac{P'M'}{PM} = \frac{R\Delta\lambda}{(R\cos\phi_1)\Delta\lambda} = \frac{1}{\cos\phi_1} = \sec\phi_1$$

The scaling in the y-direction must be the same, and so:

$$\begin{aligned} \frac{K'P'}{KP} &= \frac{R\Delta y}{R\Delta\phi} = \sec\phi_1 \\ \Delta y &= (\sec\phi_1)\Delta\phi \end{aligned}$$

This is a finite difference equation, which Wright solved with numerical integration, a time-consuming procedure before computers. An alternative available to us is to write it as a differential equation: let Δy and $\Delta\phi$ become infinitely small, and in the limit we get

$$\frac{dy}{d\phi} = \sec\phi$$

whose solution is

$$y = \int_0^{\phi} \sec\phi \, d\phi,$$

which must be multiplied by R if the sphere doesn't have a unit radius.

Geographers of the time didn't have calculus yet, so searched for a formula via other techniques. In 1620, Edmund Gunter published a table of $\log(\tan\theta)$ values, and in 1645 Henry Bond discovered that the latitude lines on the Mercator map were analogous to a scaling of those values. Essentially, he realized that

$$\int_0^{\phi} \sec\phi \, d\phi = -\ln\tan\left(\frac{1}{2}(\frac{\pi}{2} - \phi)\right)$$

Newton and Leibnitz developed the calculus during the 1660s, and in 1668, via a complicated geometric argument, James Gregory proved the equation. During the next decade or two, simpler proofs of $\int \sec\phi \, d\phi$ appeared, and a modern notation version of Isaac Barrow's solution from 1670 is given below. It was also the first to employ partial fractions decomposition.

$$\begin{aligned}
\sec\phi &= \frac{1}{\cos\phi} = \frac{\cos\phi}{\cos^2\phi} \\
&= \frac{\cos\phi}{1-\sin^2\phi} = \frac{\cos\phi}{(1+\sin\phi)(1-\sin\phi)} \\
&= \frac{1}{2}\left(\frac{\cos\phi}{1+\sin\phi} + \frac{\cos\phi}{1-\sin\phi}\right)
\end{aligned}$$

So the task becomes,

$$\int \sec\phi \, d\phi = \frac{1}{2}\int \left(\frac{\cos\phi}{1+\sin\phi} + \frac{\cos\phi}{1-\sin\phi}\right) d\phi$$

The first term inside the right-hand integral is of the form $\frac{u'}{u}$, where u is a function of ϕ, and the second term is of the form $-\frac{u'}{u}$. So, using

$$\int \frac{u'}{u} \, d\phi = \ln|u(\phi)| + C,$$

we get

$$\begin{aligned}
\int \sec\phi \, d\phi &= \frac{1}{2}\left[\ln|1+\sin\phi| - \ln|1-\sin\phi|\right] + C \\
&= \frac{1}{2}\ln\left|\frac{1+\sin\phi}{1-\sin\phi}\right| + C.
\end{aligned}$$

Inside the logarithm, multiply top and bottom by $(1+\sin\phi)$. The denominator becomes

$$(1-\sin\phi)(1+\sin\phi) = 1 - \sin^2\phi = \cos^2\phi,$$

so the expression inside the logarithm is now:

$$\int \sec\phi \, d\phi = \frac{1}{2}\ln\left|\frac{(1+\sin\phi)^2}{\cos^2\phi}\right| + C$$

or, using the power property of logarithms:

$$= \ln \left| \frac{1 + \sin \phi}{\cos \phi} \right| + C.$$

Now, write $\phi = 2(\phi/2)$, and use the double-angle formulas:

$$\sin \phi = 2 \sin \phi/2 \cos \phi/2, \quad \cos \phi = \cos^2 \phi/2 - \sin^2 \phi/2$$

and

$$1 = (\cos \pi/2)^2 + (\sin \phi/2)^2$$

to rewrite the expression inside the logarithm:

$$\begin{aligned} \frac{1+\sin \phi}{\cos \phi} &= \frac{1+2 \sin \phi/2 \cos \phi/2}{\cos^2 \phi/2 - \sin^2 \phi/2} \\ &= \frac{(\cos \pi/2)^2 + (\sin \phi/2)^2 + 2 \sin \phi/2 \cos \phi/2}{\cos^2 \phi/2 - \sin^2 \phi/2} \\ &= \frac{(\cos \phi/2 + \sin \phi/2)^2}{(\cos \phi/2 + \sin \phi/2)(\cos \phi/2 - \sin \phi/2)} \\ &= \frac{\cos \phi/2 + \sin \phi/2}{\cos \phi/2 - \sin \phi/2} \end{aligned}$$

Divide the numerator and denominator by $\cos \phi/2$:

$$\frac{1 + \tan \phi/2}{1 - \tan \phi/2}$$

This can be simplified using:

$$\tan(A + B) = \frac{\tan A + \tan B}{1 - \tan A \cdot \tan B}$$

with $A = \pi/4$ and $B = \phi/2$. So:

$$\frac{1 + \tan \phi/2}{1 - \tan \phi/2} = \tan (\pi/4 + \phi/2)$$

Substitute this into the logarithmic expression:

$$\int \sec \phi \, d\phi = \ln \left| \tan \left(\frac{\pi}{4} + \frac{\phi}{2} \right) \right| + C.$$

Turning now to the definite integral, we have

$$\int_0^{\phi} \sec t \, dt = \ln \left| \tan \left(\frac{\pi}{4} + \frac{\phi}{2} \right) \right| - \ln \tan \frac{\pi}{4}$$

But $\ln(\tan \frac{\pi}{4}) = \ln 1 = 0$, so

$$\int_0^{\phi} \sec t \, dt = \ln \tan \left(\frac{\pi}{4} + \frac{\phi}{2} \right)$$

We've dropped the absolute value sign because in the relevant range for ϕ, — $\pi/2 < \phi < \pi/2$, $\tan(\pi/4 + \phi/2)$ is always positive.

We began with:

$$y = \int_0^{\phi} \sec t \, dt$$

It can now be expressed as:

$$y = \ln \tan\left(\frac{\pi}{4} + \frac{\phi}{2}\right),$$

where $R \cdot y$ is the height of a parallel at latitude ϕ on a Mercator map.

This form of the equation quickly became popular because tables of $\log(\tan\theta)$ values had already been created. However, John Wallis published a series form for the integral in 1685, offering an alternative way of computing the height. From the partial fractions approach,

$$\int_0^{\phi} \sec t \, dt = \int_0^{\phi} \frac{\cos t}{1 - \sin^2 t} \, dt,$$

The fraction can be expanded as:

$$= \int_0^{\theta} \cos t \left[1 + \sin^2 t + \sin^4 t + \sin^6 t + \ldots\right] dt.$$

This uses the geometric series:

$$\frac{1}{1-a} = 1 + a + a^2 + a^3 + a^4 + \ldots \quad \text{if} \quad |a| < 1,$$

with $a = \sin^2 t$, which satisfies $|a| < 1$ for $t \in (-\pi/2, \pi/2)$.

The next step is to convert this integral-of-an-infinite-sum into an infinite-sum-of-integrals, which can be calculated term by term. In general, such a transformation may *not* be valid, but is acceptable here because this series is convergent for all the values in the interval, and the functions are continuous.

$$\begin{aligned} \int_0^{\phi} \sec t \, dt &= \int_0^{\phi} \cos t \cdot 1 \, dt + \int_0^{\phi} \cos t \cdot \sin^2 t \, dt + \int_0^{\phi} \cos t \cdot \sin^4 t \, dt + \ldots \\ &= \sin\phi + \frac{\sin^3\phi}{3} + \frac{\sin^5\phi}{5} + \frac{\sin^7\phi}{7} + \ldots \\ &= y + \frac{y^3}{3} + \frac{y^5}{5} + \frac{y^7}{7} + \ldots \quad \text{with } y = \sin\phi. \end{aligned}$$

13.12 Calculating with Rhumb Lines

Calculations in nav.py involving rhumb lines follow the same pattern as those for great circles: finding the distance between two points, determining the bearing to travel from one point to another (which will stay the same throughout the trip), and obtaining the destination coordinate if only the start point, bearing, and travel distance are supplied.

13.12.1 Distance. Given the coordinates of two points, the rhumb line distance between them is obtained using nav.rhumbDistTo() in nav.py.

```
def rhumbDistTo(lat1, lon1, lat2, lon2):
  phi1 = math.radians(lat1)
  phi2 = math.radians(lat2)
  deltaPhi = phi2 - phi1
  dLambda = math.radians(lon2 - lon1)

  deltaY = math.log(math.tan(math.pi/4 + phi2/2) / \
                    math.tan(math.pi/4 + phi1/2))
  q = deltaPhi/deltaY if abs(deltaY) > 1e-12 \
                      else math.cos(phi1)
  ''' If on a constant latitude course (traveling east-west),
      use cos(phi)
  '''
  if abs(dLambda) > math.pi:
    # use shorter rhumb line across the date line
    dLambda = dLambda - 2*math.pi if dLambda > 0 \
                                  else dLambda + 2*math.pi
  return math.sqrt(deltaPhi**2 + (q * dLambda)**2) * EARTH_RADIUS
```

The latitude and longitude are supplied as decimal degrees, and the distance is returned in kilometers.

`deltaY` is Δy in the projection in Fig. 13.23(b), which is obtained by subtracting the two Mercator values (y_2 and y_1), albeit by utilizing division inside the logarithmic expression.

`q` is the ratio $\frac{\Delta\phi}{\Delta y}$ which defines the scaling from lengths in the projection to their globe values.

From Fig. 13.23(b):

$$\Delta s' = R\sqrt{\Delta y^2 + \Delta x^2}$$

Let

$$q = \frac{\Delta\phi}{\Delta y} \text{ if } |\Delta\phi| > 0, \text{ otherwise } \cos\phi_1$$

and so

$$\Delta y = \frac{\Delta\phi}{q}$$

The conditional associated with q deals with a trip that heads directly east or west from the starting point, P.

Replace the Δy term in the square root:

$$\begin{aligned} \Delta s' &= R\sqrt{\Delta y^2 + \Delta\lambda^2} \\ &= R\sqrt{\left(\frac{\Delta\phi}{q}\right)^2 + \Delta\lambda^2} \\ &= \frac{R}{q}\sqrt{\Delta\phi^2 + (q\Delta\lambda)^2} \end{aligned}$$

The q ratio links other lengths in the spherical and planar rectangles; in particular, it can be used to relate the travel distances

$$\begin{aligned} q &= \frac{\Delta s}{\Delta s'} \\ \Delta s &= q\Delta s' \end{aligned}$$

$\Delta s'$ can be replaced by the square root from above, producing:

$$\Delta s = q\Delta s' = R\sqrt{\Delta\phi^2 + (q\Delta\lambda)^2}$$

This is the expression calculated by nav.rhumbDistTo().

13.12.2 Bearing. Given the coordinates of two points, the bearing along the rhumb line is obtained using nav.rhumbBearingTo():

```
def rhumbBearingTo(lat1, lon1, lat2, lon2):
  # return degree bearing from first to second coord
  phi1 = math.radians(lat1)
  phi2 = math.radians(lat2)
  dLambda = math.radians(lon2 - lon1)

  deltaY = math.log(math.tan(math.pi/4 + phi2/2)/ \
                    math.tan(math.pi/4 + phi1/2))

  if abs(dLambda) > math.pi:
    # use shorter rhumb line across the date line
    dLambda = dLambda - 2*math.pi if dLambda > 0  \
                                  else dLambda + 2*math.pi

  psi = math.atan2(dLambda, deltaY)
  return (math.degrees(psi) + 360) % 360
```

`psi` in the code is ψ in Fig. 13.23(b), which is obtained from the arc-tangent of $\frac{\Delta x}{\Delta y}$.

The bearing is returned in degrees in the range 0°..360°, with negative values mapped to the range 180°–360°.

13.12.3 Calculating a Destination. nav.rhumbDestCoord() is supplied with a starting point, the bearing, a distance to travel, and returns the destination coordinate.

```
def rhumbDestCoord(lat1, lon1, dist, bearing):
  phi1 = math.radians(lat1)
  lambda1 = math.radians(lon1)
  psi = math.radians(bearing)
  delta = dist/EARTH_RADIUS

  deltaPhi = delta * math.cos(psi)
```

```
  phi2 = phi1 + deltaPhi

  # normalize latitude if going past the pole
  if phi2 > math.pi/2:
    phi2 = math.pi - phi2
  elif phi2 < -math.pi/2:
    phi2 = -math.pi - phi2

  deltaY = math.log(math.tan(math.pi/4 + phi2/2)/ \
                    math.tan(math.pi/4 + phi1/2))
  q = deltaPhi/deltaY if abs(deltaY) > 1e-12 \
                      else math.cos(phi1)

  dLambda = delta * math.sin(psi)/q
  lambda2 = lambda1 + dLambda

  deglambda = ((math.degrees(lambda2) + 540) % 360) - 180
  return math.degrees(phi2), deglambda  # lat, lon
```

In order to calculate the latitude and longitude of the destination (point Q in Fig. 13.23(a)), we must obtain the two deltas $\Delta\phi$ and $\Delta\lambda$.

For $\Delta\phi$, note that the `bearing` in the code is ψ in Fig. 13.23(a), `dist` is Δs, and `delta` is $\frac{\Delta s}{R}$. Therefore:

$$\cos\psi = \frac{R\Delta\phi}{\Delta s} = \frac{\Delta\phi}{\text{delta}}$$

and so

$$\Delta\phi = \text{ delta } \cdot \cos\psi$$

For $\Delta\lambda$, we again make use of the q ratio (first seen in the distance calculation):

$$q = \frac{\Delta\phi}{\Delta y} = \frac{\Delta s}{\Delta s'}$$

or

$$\Delta s' = \frac{\Delta s}{q}$$

With reference to Fig. 13.23(b):

$$\sin\psi = \frac{R\Delta\lambda}{\Delta s'}$$

$$R\Delta\lambda = \sin\psi \cdot \Delta s' = \sin\psi \cdot \frac{\Delta s}{q}$$

Divide through by R, and replace $\frac{\Delta s}{R}$ by `delta`:

$$\Delta\lambda = \sin\psi \cdot \frac{\Delta s}{qR} = \sin\psi \frac{\text{delta}}{q}$$

nav.rhumbDestCoord() adds `deltaPhi` ($\Delta\phi$) to the latitude and `dLambda` ($\Delta\lambda$) to the longitude, normalizes the results, and returns.

13.12.4 Drawing a Rhumb Line. showMap.py utilizes the rhumb line functions in nav.py to offer the user a choice of four map projections for displaying a rhumb line connecting Perth, Australia to Nord, Greenland. Fig. 13.17 shows the Orthographic and Platte Carrée options, while Fig. 13.24 draws on Mercator and Gnomonic maps.

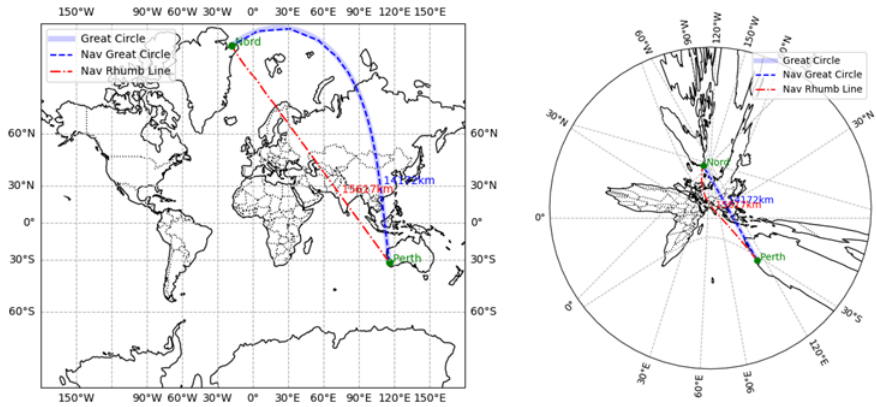


Figure 13.24. Mercator and Gnomonic Perth to Nord Trips

The obvious benefit of the Mercator projection is that the rhumb line is a straight line. The Gnomonic projection is included here since it has an analogous effect on a great circle, but with the side-effect of heavily distorting the map.

The rhumb line coordinates are generated by nav.rhumbCoords():

```
coords = nav.rhumbCoords(lat1, lon1, lat2, lon2, 20)
print("\nRhumb Line (lat,lon)s and deg Bearing:")
for coord in coords:
  cLat, cLon, cBear = coord
  print(f"  ({cLat:.2f}, {cLon:.2f}); {cBear:.2f}")
```

The returned list contains (latitude, longitude, and bearing) triplets, with the most interesting aspect being that the bearings do *not* change:

```
  :
(24.88, 72.70); 324.04
(30.56, 68.04); 324.04
(36.25, 63.09); 324.04
```

```
(41.93, 57.77); 324.04
(47.62, 51.96); 324.04
(53.30, 45.47); 324.04
(58.98, 38.05); 324.04
(64.67, 29.29); 324.04
  :
```

The data is passed to Cartopy's plot():

```
lats, lons, _ = zip(*coords)
plt.plot(lons, lats, color='red', ls='-.',
    label="Nav Rhumb Line", transform=ccrs.PlateCarree())
```

nav.rhumbCoords() generates its points in a similar way to nav.greatCircleCoords(), but by calling the rhumb line versions of the distance, bearing, and destination functions.

```
def rhumbCoords(lat1, lon1, lat2, lon2, num):
  lat1 = parseCoord(lat1)
  lon1 = parseCoord(lon1)
  lat2 = parseCoord(lat2)
  lon2 = parseCoord(lon2)

  dist = rhumbDistTo(lat1, lon1, lat2, lon2)
  distStep = dist/num

  bearing = rhumbBearingTo(lat1, lon1, lat2, lon2)
  nSteps = 1
  coords = [(lat1, lon1, bearing)]
  currLat, currLon = lat1, lon1
  while (nSteps < num) or \
        (not areClose(currLat, currLon, lat2, lon2)):
    destLat, destLon =  \
       rhumbDestCoord(currLat, currLon, distStep, bearing)
    bearing = rhumbBearingTo(destLat, destLon, lat2, lon2)
    nSteps += 1
    coords.append((destLat, destLon, bearing))
    currLat, currLon = destLat, destLon
  return coords
```

13.12.5 More Mercator Trips. mercatorTrip.py doesn't use any new navigation functions, but restricts itself to drawing a single great circle and rhumb line between any two cities on a Mercator map. The user supplies the city names and nav.readCoord() attempts to find their latitudes and longitudes in the world_cities.csv file. Fig. 13.25 shows two examples of its output.

The main visual change is that the map boundaries are limited by calls to nav.limitExtent() which calculates a rectangular region that is passed to Cartopy.

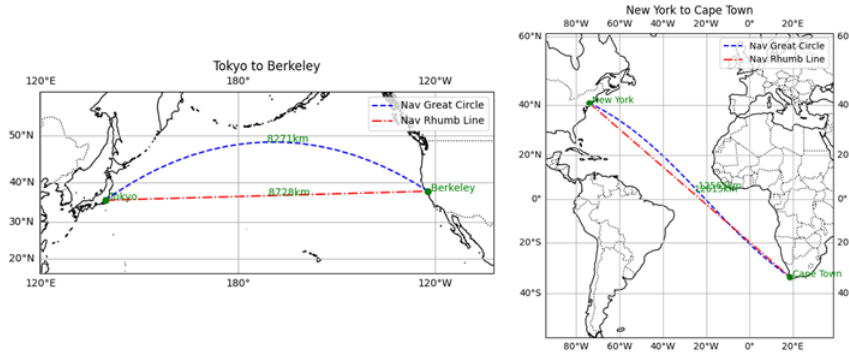


Figure 13.25. Two Mercator Map Trips

Exercises

(1) mercatorTrip.py can be repurposed to investigate a surprisingly tricky question – how much longer can a rhumb line be compared to the great circle between the same two points? The only detailed analysis that we know of was carried out by P.J. Heawood in "Comparison between Rhumb-Line and Great-Circle Courses" [**Hea26**].

(2) A common error when describing the Mercator map is to explain it not by visualizing Wright's gradually expanding bladder but as the result of wrapping a cylinder around the Earth, tangent to the equator, as in Fig. 13.26. Locations on the Earth's surface are projected onto the cylinder by a point-size light emitted from the Earth's core.

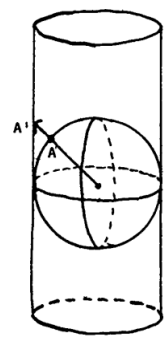


Figure 13.26. Cylindrical Projection: Point A is mapped to A'.

(a) What spacing of the parallels does this projection produce instead of the $R\int_0^{\phi}\sec\phi\,d\phi$ placement at latitude ϕ on the Mercator map?

(b) Are the parallels more spread out? This is difficult to decide as the cylindrical projection also stretches out the map at high latitudes (Fig. 13.27).

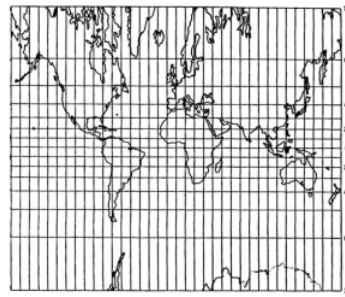


Figure 13.27. Cylindrical Projection World Map

(3) The rhumb line spiral in Fig. 13.18 is drawn by NYCSpiral.py. Experiment with different initial bearings to see how the spiral changes.

The code calls Cartopy plot() like so

```
plt.plot(lons, lats, color='red',
                     transform=ccrs.Geodetic())
      # to deal with curvature between points
```

which uses a `Geodetic` transform rather than the `PlateCarree` argument used elsewhere. Recall that `Geodetic` interpolates extra points between the supplied coordinates while `PlateCarree` merely connects them with a straight line.

If `transform=ccrs.Geodetic()` is replaced by `transform=ccrs.PlateCarree()` in the above code, what happens to the spiral and why?

(4) The Lambert cylindrical equal area projection in Figs. 13.5 and 13.6 of section 13.4 is quite different from the cylindrical mapping in Fig. 13.27. Why are they different?

Based on Fig. 13.28, show that the Lambert projection does indeed preserve areas.

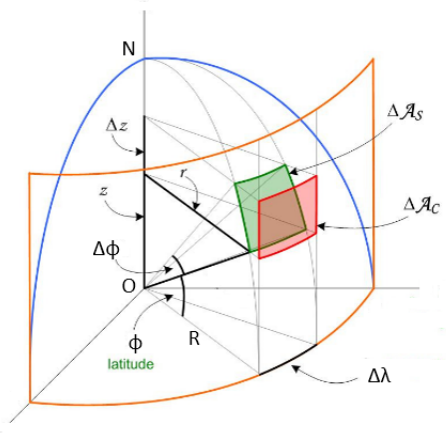


Figure 13.28. Lambert projection of area ΔA_s onto ΔA_c

(5) In 1685, Wallis devised a series form for the integral of $\tan x$ which begins with the following rewrites:

$$\begin{aligned}\int \tan x\, dx &= \int \frac{\sin x}{\cos x}\, dx \\ &= \int \frac{\sin x \cdot \cos x}{\cos^2 x}\, dx \\ &= \int \sin x \cdot \cos x \left(\frac{1}{1-\sin^2 x}\right) dx\end{aligned}$$

Continue the working, which will result in:

$$\int_0^x \tan x\, dx = \frac{1}{2}(s^2 + \frac{s^4}{2} + \frac{s^6}{3} + \frac{s^8}{4} + \cdots),$$

where $s = \sin x$.

(a) For what range of x values does this converge, and why?

(b) What goes wrong when we try to do the same for $\int_0^x \cot x\, dx$?

(6) Use Wallis' series for $\sec x$ to implement a partial sum approximations of the integral in Python. How do you decide when you have enough terms and can stop?