Movable Type Scripts

Calculate distance, bearing and more between Latitude/Longitude points

This page presents a variety of calculations for latitude/longitude points, with the formulas and code fragments for implementing them.

All these formulas are for calculations on the basis of a spherical earth (ignoring ellipsoidal effects) – which is accurate enough[*] for most purposes… [In fact, the earth is very slightly ellipsoidal; using a spherical model gives errors typically up to 0.3%[1] – see notes for further details].

Great-circle distance between two points

Enter the co-ordinates into the text boxes to try out the calculations. A variety of formats are accepted, principally:

- deg-min-sec suffixed with N/S/E/W (e.g. 40°44′55″N, 73 59 11W), or
- signed decimal degrees without compass direction, where negative indicates west/south (e.g. 40.7486, -73.9864):

Point 1:	50 03 59N	005 42 53W
Point 2:	58 38 38N	003 04 12W

Distance:	**968.9** km (to 4 SF[*])
Initial bearing:	**009° 07′ 11″**
Final bearing:	**011° 16′ 31″**
Midpoint:	**54° 21′ 44″ N, 004° 31′ 50″ W**

And you can see it on a map

Distance

This uses the ‘**haversine**’ formula to calculate the great-circle distance between two points – that is, the shortest distance over the earth’s surface – giving an ‘as-the-crow-flies’ distance between the points (ignoring any hills they fly over, of course!).

Haversine formula:

$$a = \sin^2(\Delta\varphi/2) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2(\Delta\lambda/2)$$

$$c = 2 \cdot \operatorname{atan2}(\sqrt{a}, \sqrt{(1-a)})$$

$$d = R \cdot c$$

where φ *is latitude,* λ *is longitude,* R *is earth’s radius (mean radius = 6,371km);*
note that angles need to be in radians to pass to trig functions!

JavaScript:

```
const R = 6371e3; // metres
const φ1 = lat1 * Math.PI/180; // φ, λ in radians
const φ2 = lat2 * Math.PI/180;
const Δφ = (lat2-lat1) * Math.PI/180;
const Δλ = (lon2-lon1) * Math.PI/180;

const a = Math.sin(Δφ/2) * Math.sin(Δφ/2) +
          Math.cos(φ1) * Math.cos(φ2) *
          Math.sin(Δλ/2) * Math.sin(Δλ/2);
const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));

const d = R * c; // in metres
```

Note in these scripts, I generally use lat/lon for latitude/longitude in degrees, and φ/λ for latitude/longitude in radians – having found that mixing degrees & radians is often the easiest route to head-scratching bugs...

The haversine formula[1] ‘remains particularly well-conditioned for numerical computation even at small distances’ – unlike calculations based on the *spherical law of cosines*. The ‘(re)versed sine’ is $1-\cos\theta$, and the ‘half-versed-sine’ is $(1-\cos\theta)/2$ or $\sin^2(\theta/2)$ as used above. Once widely used by navigators, it was described by Roger Sinnott in *Sky & Telescope* magazine in 1984 (“Virtues of the Haversine”): Sinnott explained that the angular separation between Mizar and Alcor in Ursa Major – 0°11′49.69″ – could be accurately calculated in Basic on a TRS-80 using the haversine.

For the curious, *c* is the angular distance in radians, and *a* is the square of half the chord length between the points.

If `atan2` is not available, *c* could be calculated from $2 \cdot \operatorname{asin}(\ \min(1, \sqrt{a})\ )$ (including protection against rounding errors).

Historical aside: The height of technology for navigator’s calculations used to be log tables. As there is no (real) log of a negative number, the ‘versine’ enabled them to keep trig functions in positive numbers. Also, the $\sin^2(\theta/2)$ form of the haversine avoided addition (which entailed an anti-log lookup, the addition, and a log lookup). Printed tables for the haversine/inverse-haversine (and its logarithm, to aid multiplications) saved navigators from squaring sines, computing square roots, etc – arduous and error-prone activities.

Using Chrome on an aging Core i5 PC, a distance calculation takes around 2 – 5 microseconds (hence around 200,000 – 500,000 per second). Little to no benefit is obtained by factoring out common terms; probably the JIT compiler optimises them out.

Spherical Law of Cosines

In fact, JavaScript (and most modern computers & languages) use ‘IEEE 754’ 64-bit floating-point numbers, which provide 15 significant figures of precision. By my estimate, with this precision, the simple spherical law of cosines formula (cos *c* = cos *a* cos *b* + sin *a* sin *b* cos *C*) gives well-conditioned results down to distances as small as a few metres on the earth’s surface. (*Note that the geodetic form of the law of cosines is rearranged from the canonical one so that the latitude can be used directly, rather than the* colatitude).

This makes the simpler law of cosines a reasonable 1-line alternative to the haversine formula for many geodesy purposes (if not for astronomy). The choice may be driven by programming language, processor, coding context, available trig functions (in different languages), etc – and, for very small distances an equirectangular approximation may be more suitable.

Law of cosines:

$$d = \operatorname{acos}(\ \sin\varphi_1 \cdot \sin\varphi_2 + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \cos\Delta\lambda\ ) \cdot R$$

JavaScript:

```
const φ1 = lat1 * Math.PI/180, φ2 = lat2 * Math.PI/180, Δλ = (lon2-lon1) * Math.PI/180, R = 6371e3;
const d = Math.acos( Math.sin(φ1)*Math.sin(φ2) + Math.cos(φ1)*Math.cos(φ2) * Math.cos(Δλ) ) * R;
```

Excel:

```
=ACOS( SIN(lat1)*SIN(lat2) + COS(lat1)*COS(lat2)*COS(lon2-lon1) ) * 6371000
```

(or with lat/lon in degrees):

```
=ACOS( SIN(lat1*PI()/180)*SIN(lat2*PI()/180) + COS(lat1*PI()/180)*COS(lat2*PI()/180)*COS(lon2*PI()/180-lon1*PI()/180) ) * 6371000
```

While simpler, the law of cosines is slightly slower than the haversine, in my tests.

Equirectangular approximation

If performance is an issue and accuracy less important, for small distances Pythagoras’ theorem can be used on an equirectangular projection:*

Formula

$$x = \Delta\lambda \cdot \cos\varphi_m$$

$$y = \Delta\varphi$$

$$d = R \cdot \sqrt{x^2 + y^2}$$

JavaScript:

```
const x = (λ2-λ1) * Math.cos((φ1+φ2)/2);
const y = (φ2-φ1);
const d = Math.sqrt(x*x + y*y) * R;
```

This uses just one trig and one sqrt function – as against half-a-dozen trig functions for cos law, and 7 trigs + 2 sqrts for haversine. Accuracy is somewhat complex: along meridians there are no errors, otherwise they depend on distance, bearing, and latitude, but are small enough for many purposes* (and often trivial compared with the spherical approximation itself).

Alternatively, the *polar coordinate flat-earth formula* can be used: using the co-latitudes $\theta_1 = \pi/2-\varphi_1$ and $\theta_2 = \pi/2-\varphi_2$, then $d = R \cdot \sqrt{\theta_1^2 + \theta_2^2 - 2 \cdot \theta_1 \cdot \theta_2 \cdot \cos \Delta\lambda}$. I've not compared accuracy.

Bearing

In general, your current heading will vary as you follow a great circle path (orthodrome); the final heading will differ from the initial heading by varying degrees according to distance and latitude (if you were to go from say 35°N,45°E (≈ Baghdad) to 35°N, 135°E (≈ Osaka), you would start on a heading of 60° and end up on a heading of 120°!).

This formula is for the initial bearing (sometimes referred to as forward azimuth) which if followed in a straight line along a great-circle arc will take you from the start point to the end point:[1]

Baghdad to Osaka – not a constant bearing!

Formula: $\theta = \text{atan2}(\sin \Delta\lambda \cdot \cos \varphi_2 , \cos \varphi_1 \cdot \sin \varphi_2 - \sin \varphi_1 \cdot \cos \varphi_2 \cdot \cos \Delta\lambda)$

where φ_1,λ_1 *is the start point,* φ_2,λ_2 *the end point (*$\Delta\lambda$ *is the difference in longitude)*

JavaScript: *(all angles in radians)*

```
const y = Math.sin(λ2-λ1) * Math.cos(φ2);
const x = Math.cos(φ1)*Math.sin(φ2) -
          Math.sin(φ1)*Math.cos(φ2)*Math.cos(λ2-λ1);
const θ = Math.atan2(y, x);
const brng = (θ*180/Math.PI + 360) % 360; // in degrees
```

Excel: *(all angles in radians)*

```
=ATAN2(COS(lat1)*SIN(lat2)-SIN(lat1)*COS(lat2)*COS(lon2-lon1),
       SIN(lon2-lon1)*COS(lat2))
*note that Excel reverses the arguments to ATAN2 – see notes below
```

Since atan2 returns values in the range -π ... +π (that is, -180° ... +180°), to normalise the result to a compass bearing (in the range 0° ... 360°, with −ve values transformed into the range 180° ... 360°), convert to degrees and then use (θ+360) % 360, where % is (floating point) modulo.

For final bearing, simply take the *initial* bearing from the *end* point to the *start* point and reverse it (using θ = (θ+180) % 360).

Midpoint

This is the half-way point along a great circle path between the two points.[1]

Formula:

$$B_x = \cos \varphi_2 \cdot \cos \Delta\lambda$$

$$B_y = \cos \varphi_2 \cdot \sin \Delta\lambda$$

$$\varphi_m = \text{atan2}(\sin \varphi_1 + \sin \varphi_2, \sqrt{(\cos \varphi_1 + B_x)^2 + B_y^2})$$

$$\lambda_m = \lambda_1 + \text{atan2}(B_y, \cos(\varphi_1)+B_x)$$

JavaScript: *(all angles in radians)*

```
const Bx = Math.cos(φ2) * Math.cos(λ2-λ1);
const By = Math.cos(φ2) * Math.sin(λ2-λ1);
const φ3 = Math.atan2(Math.sin(φ1) + Math.sin(φ2),
                      Math.sqrt( (Math.cos(φ1)+Bx)*(Math.cos(φ1)+Bx) + By*By ) );
const λ3 = λ1 + Math.atan2(By, Math.cos(φ1) + Bx);
```

The longitude can be normalised to −180…+180 using `(lon+540)%360-180`

Just as the initial bearing may vary from the final bearing, the midpoint may not be located half-way between latitudes/longitudes; the midpoint between 35°N,45°E and 35°N,135°E is around 45°N,90°E.

Intermediate point

An intermediate point at any fraction along the great circle path between two points can also be calculated.[1]

Formula: $a = \sin((1-f)\cdot\delta) / \sin\delta$

$b = \sin(f\cdot\delta) / \sin\delta$

$x = a \cdot \cos\varphi_1 \cdot \cos\lambda_1 + b \cdot \cos\varphi_2 \cdot \cos\lambda_2$

$y = a \cdot \cos\varphi_1 \cdot \sin\lambda_1 + b \cdot \cos\varphi_2 \cdot \sin\lambda_2$

$z = a \cdot \sin\varphi_1 + b \cdot \sin\varphi_2$

$\varphi_i = \text{atan2}(z, \sqrt{x^2 + y^2})$

$\lambda_i = \text{atan2}(y, x)$

where f *is fraction along great circle route (f=0 is point 1, f=1 is point 2),* δ *is the angular distance* d/R *between the two points.*

Destination point given distance and bearing from start point

Given a start point, initial bearing, and distance, this will calculate the destination point and final bearing travelling along a (shortest distance) great circle arc.

Destination point along great-circle given distance and bearing from start point

Start point:	53°19′14″N, 001°43′47″W	Destination point:	**53° 11′ 18″ N, 000° 08′ 00″ E**
Bearing:	096°01′18″	Final bearing:	**097° 30′ 52″**
Distance:	124.8 km		view map

Formula: $\varphi_2 = \text{asin}(\sin\varphi_1 \cdot \cos\delta + \cos\varphi_1 \cdot \sin\delta \cdot \cos\theta)$

$\lambda_2 = \lambda_1 + \text{atan2}(\sin\theta \cdot \sin\delta \cdot \cos\varphi_1, \cos\delta - \sin\varphi_1 \cdot \sin\varphi_2)$

where φ *is latitude,* λ *is longitude,* θ *is the bearing (clockwise from north),* δ *is the angular distance* d/R*;* d *being the distance travelled,* R *the earth's radius*

JavaScript: *(all angles in radians)*

```
const φ2 = Math.asin( Math.sin(φ1)*Math.cos(d/R) +
                      Math.cos(φ1)*Math.sin(d/R)*Math.cos(brng) );
const λ2 = λ1 + Math.atan2(Math.sin(brng)*Math.sin(d/R)*Math.cos(φ1),
                           Math.cos(d/R)-Math.sin(φ1)*Math.sin(φ2));
```

The longitude can be normalised to −180…+180 using `(lon+540)%360-180`

Excel: *(all angles in radians)*

```
lat2: =ASIN(SIN(lat1)*COS(d/R) + COS(lat1)*SIN(d/R)*COS(brng))
lon2: =lon1 + ATAN2(COS(d/R)-SIN(lat1)*SIN(lat2), SIN(brng)*SIN(d/R)*COS(lat1))
* Remember that Excel reverses the arguments to ATAN2 – see notes below
```

For final bearing, simply take the *initial* bearing from the *end* point to the *start* point and reverse it with `(brng+180)%360`.

Intersection of two paths given start points and bearings

This is a rather more complex calculation than most others on this page, but I've been asked for it a number of times. This comes from