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 formulæ and code fragments for implementing them.

All these formulæ 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 (aren't those Google guys wonderful!)

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:

$$a = \sin^2(\Delta\varphi/2) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2(\Delta\lambda/2)$$
$$c = 2 \cdot \text{atan2}(\sqrt{a}, \sqrt{(1-a)})$$
$$d = R \cdot c$$

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:

```
var R = 6371e3; // metres
var φ1 = lat1.toRadians();
var φ2 = lat2.toRadians();
var Δφ = (lat2-lat1).toRadians();
var Δλ = (lon2-lon1).toRadians();

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

var d = R * c;
```

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 &*

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

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 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 $2 \cdot \text{asin}(\min(1, \sqrt{a}))$ (including protection against rounding errors).

positive numbers. Also, the sin²(θ/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 a middling 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:

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

JavaScript:

```
var φ1 = lat1.toRadians(), φ2 = lat2.toRadians(), Δλ = (lon2-lon1).toRadians(), R = 6371e3; // gives d in metres
var 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:

```
var x = (λ2-λ1) * Math.cos((φ1+φ2)/2);
var y = (φ2-φ1);
var 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]

Formula:

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

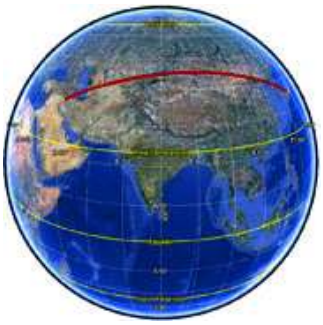
where φ_1,λ_1 *is the start point,* φ_2,λ_2 *the end point (*$\Delta\lambda$ *is the difference in longitude)*

JavaScript: *(all angles in radians)*

```
var y = Math.sin(λ2-λ1) * Math.cos(φ2);
var x = Math.cos(φ1)*Math.sin(φ2) -
        Math.sin(φ1)*Math.cos(φ2)*Math.cos(λ2-λ1);
var brng = Math.atan2(y, x).toDegrees();
```

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
```



Baghdad to Osaka – not a constant bearing!

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)*

```
var Bx = Math.cos(φ2) * Math.cos(λ2-λ1);
var By = Math.cos(φ2) * Math.sin(λ2-λ1);
var φ3 = Math.atan2(Math.sin(φ1) + Math.sin(φ2),
                    Math.sqrt( (Math.cos(φ1)+Bx)*(Math.cos(φ1)+Bx) + By*By ) );
var 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
Bearing: 096°01′18″
Distance: 124,8 km

Destination point: **53° 11′ 18″ N, 000° 08′ 00″ E**
Final bearing: **097° 30′ 52″**

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)*

```
var φ2 = Math.asin( Math.sin(φ1)*Math.cos(d/R) +
                    Math.cos(φ1)*Math.sin(d/R)*Math.cos(brng) );
var 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 Ed William's aviation formulary. See below for the JavaScript.

Intersection of two great-circle paths

Point 1: 51.8853 N, 0.2545 E Brng 1: 108.55°

Point 2: 49.0034 N, 2.5735 E Brng 2: 32.44°

Intersection point: **50° 54′ 27″ N, 004° 30′ 31″ E**

Formula:

$\delta_{12} = 2 \cdot \text{asin}(\sqrt{(\sin^2(\Delta\varphi/2) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2(\Delta\lambda/2))})$ — angular dist. p1–p2

$\theta_a = \text{acos}((\sin\varphi_2 - \sin\varphi_1 \cdot \cos\delta_{12}) / (\sin\delta_{12} \cdot \cos\varphi_1))$ — initial / final bearings between points 1 & 2

$\theta_b = \text{acos}((\sin\varphi_1 - \sin\varphi_2 \cdot \cos\delta_{12}) / (\sin\delta_{12} \cdot \cos\varphi_2))$

if $\sin(\lambda_2 - \lambda_1) > 0$
  $\theta_{12} = \theta_a$
  $\theta_{21} = 2\pi - \theta_b$
else
  $\theta_{12} = 2\pi - \theta_a$
  $\theta_{21} = \theta_b$

$\alpha_1 = \theta_{13} - \theta_{12}$ — angle p2–p1–p3

$\alpha_2 = \theta_{21} - \theta_{23}$ — angle p1–p2–p3

$\alpha_3 = \text{acos}(-\cos\alpha_1 \cdot \cos\alpha_2 + \sin\alpha_1 \cdot \sin\alpha_2 \cdot \cos\delta_{12})$ — angle p1–p2–p3

$\delta_{13} = \text{atan2}(\sin\delta_{12} \cdot \sin\alpha_1 \cdot \sin\alpha_2 , \cos\alpha_2 + \cos\alpha_1 \cdot \cos\alpha_3)$ — angular dist. p1–p3

$\varphi_3 = \text{asin}(\sin\varphi_1 \cdot \cos\delta_{13} + \cos\varphi_1 \cdot \sin\delta_{13} \cdot \cos\theta_{13})$ — p3 lat

$\Delta\lambda_{13} = \text{atan2}(\sin\theta_{13} \cdot \sin\delta_{13} \cdot \cos\varphi_1 , \cos\delta_{13} - \sin\varphi_1 \cdot \sin\varphi_3)$ — long p1–p3

$\lambda_3 = \lambda_1 + \Delta\lambda_{13}$ — p3 long

where φ_1, λ_1, θ_{13} *: 1st start point & (initial) bearing from 1st point towards intersection point*
φ_2, λ_2, θ_{23} *: 2nd start point & (initial) bearing from 2nd point towards intersection point*
φ_3, λ_3 *: intersection point*

% = (floating point) modulo

note – if sin α_1 = 0 and sin α_2 = 0: infinite solutions
if sin α_1 · sin α_2 < 0: ambiguous solution
this formulation is not always well-conditioned for meridional or equatorial lines

This is a lot simpler using vectors rather than spherical trigonometry: see latlong-vectors.html.

Cross-track distance

Here's a new one: I've sometimes been asked about distance of a point from a great-circle path (sometimes called cross track error).

Formula: $d_{xt} = \text{asin}(\sin(\delta_{13}) \cdot \sin(\theta_{13}-\theta_{12})) \cdot R$

where δ_{13} *is (angular) distance from start point to third point*
θ_{13} *is (initial) bearing from start point to third point*
θ_{12} *is (initial) bearing from start point to end point*
R is the earth's radius

JavaScript:
```
var dXt = Math.asin(Math.sin(d13/R)*Math.sin(θ13-θ12)) * R;
```

Here, the great-circle path is identified by a start point and an end point – depending on what initial data you're working from, you can use the formulæ above to obtain the relevant distance and bearings. The sign of d_{xt} tells you which side of the path the third point is on.

The along-track distance, from the start point to the closest point on the path to the third point, is

Formula: $d_{at} = \text{acos}(\cos(\delta_{13}) / \cos(\delta_{xt})) \cdot R$

where δ_{13} *is (angular) distance from start point to third point*
δ_{xt} *is (angular) cross-track distance*
R is the earth's radius

JavaScript:
```
var dAt = Math.acos(Math.cos(d13/R)/Math.cos(dXt/R)) * R;
```

Closest point to the poles

And: 'Clairaut's formula' will give you the maximum latitude of a great circle path, given a bearing θ and latitude φ on the great circle:

Formula: $\varphi_{max} = \text{acos}(| \sin \theta \cdot \cos \varphi |)$

JavaScript:
```
var φMax = Math.acos(Math.abs(Math.sin(θ)*Math.cos(φ)));
```

Rhumb lines

A 'rhumb line' (or loxodrome) is a path of constant bearing, which crosses all meridians at the same angle.

Sailors used to (and sometimes still) navigate along rhumb lines since it is easier to follow a constant compass bearing than to be continually adjusting the bearing, as is needed to follow a great circle. Rhumb lines are straight lines on a Mercator Projection map (also helpful for navigation).

Rhumb lines are generally longer than great-circle (orthodrome) routes. For instance, London to New York is 4% longer along a rhumb line than along a great circle – important for aviation fuel, but not particularly to sailing vessels. New York to Beijing – close to the most extreme example possible (though not sailable!) – is 30% longer along a rhumb line.

Rhumb-line distance between two points

Point 1: 50 21 59N, 004 08 02W
Point 2: 42 21 04N, 071 02 27W

Distance: **5198** km
Bearing: **260° 07′ 38″**
Midpoint: **46° 21′ 32″ N, 038° 49′ 00″ W**

view map

Destination point along rhumb line given distance and bearing from start point

Start point: 51 07 32N, 001 20 17E
Bearing: 116°38′10
Distance: 40,23 km

Destination point: **50° 57′ 48″ N, 001° 51′ 09″ E**

view map

Key to calculations of rhumb lines is the *inverse Gudermannian function*[1], which gives the height on a Mercator projection map of a given latitude: $\ln(\tan\varphi + \sec\varphi)$ or $\ln(\tan(\pi/4+\varphi/2))$. This of course tends to infinity at the poles (in keeping with the Mercator projection). For obsessives, there is even an ellipsoidal version, the 'isometric latitude': $\psi = \ln(\tan(\pi/4+\varphi/2) / [(1-e\cdot\sin\varphi) / (1+e\cdot\sin\varphi)]^{e/2})$, or its better-conditioned equivalent $\psi = \operatorname{atanh}(\sin\varphi) - e\cdot\operatorname{atanh}(e\cdot\sin\varphi)$.

The formulæ to derive Mercator projection easting and northing coordinates from spherical latitude and longitude are then[1]

$$E = R \cdot \lambda$$
$$N = R \cdot \ln(\tan(\pi/4 + \varphi/2))$$

The following formulæ are from Ed Williams' aviation formulary.[1]

Distance

Since a rhumb line is a straight line on a Mercator projection, the distance between two points along a rhumb line is the length of that line (by Pythagoras); but the distortion of the projection needs to be compensated for.

On a constant latitude course (travelling east-west), this compensation is simply $\cos\varphi$; in the general case, it is $\Delta\varphi/\Delta\psi$ where $\Delta\psi = \ln(\tan(\pi/4 + \varphi_2/2) / \tan(\pi/4 + \varphi_1/2))$ (the 'projected' latitude difference)

Formula:

$$\Delta\psi = \ln(\tan(\pi/4 + \varphi_2/2) / \tan(\pi/4 + \varphi_1/2)) \quad \text{('projected' latitude difference)}$$
$$q = \Delta\varphi/\Delta\psi \text{ (or } \cos\varphi \text{ for E-W line)}$$
$$d = \sqrt{(\Delta\varphi^2 + q^2\cdot\Delta\lambda^2)} \cdot R \quad \text{(Pythagoras)}$$

where φ *is latitude*, λ *is longitude*, $\Delta\lambda$ *is taking shortest route (<180°),* R *is the earth's radius*, ln *is natural log*

JavaScript: *(all angles in radians)*

```
var Δψ = Math.log(Math.tan(Math.PI/4+φ2/2)/Math.tan(Math.PI/4+φ1/2));
var q = Math.abs(Δψ) > 10e-12 ? Δφ/Δψ : Math.cos(φ1); // E-W course becomes ill-conditioned with 0/0

// if dLon over 180° take shorter rhumb line across the anti-meridian:
if (Math.abs(Δλ) > Math.PI) Δλ = Δλ>0 ? -(2*Math.PI-Δλ) : (2*Math.PI+Δλ);

var dist = Math.sqrt(Δφ*Δφ + q*q*Δλ*Δλ) * R;
```

Bearing

A rhumb line is a straight line on a Mercator projection, with an angle on the projection equal to the compass bearing.

Formula:

$$\Delta\psi = \ln(\tan(\pi/4 + \varphi_2/2) / \tan(\pi/4 + \varphi_1/2)) \quad \text{('projected' latitude difference)}$$
$$\theta = \operatorname{atan2}(\Delta\lambda, \Delta\psi)$$

where φ *is latitude*, λ *is longitude*, $\Delta\lambda$ *is taking shortest route (<180°),* R *is the earth's radius*, ln *is natural log*

JavaScript: *(all angles in radians)*

```
var Δψ = Math.log(Math.tan(Math.PI/4+φ2/2)/Math.tan(Math.PI/4+φ1/2));

// if dLon over 180° take shorter rhumb line across the anti-meridian:
if (Math.abs(Δλ) > Math.PI) Δλ = Δλ>0 ? -(2*Math.PI-Δλ) : (2*Math.PI+Δλ);

var brng = Math.atan2(Δλ, Δψ).toDegrees();
```

Destination

Given a start point and a distance *d* along constant bearing *θ*, this will calculate the destination point. If you maintain a constant bearing along a rhumb line, you will gradually spiral in towards one of the poles.

Formula:

$$\delta = d/R \quad \text{(angular distance)}$$
$$\Delta\psi = \ln(\tan(\pi/4 + \varphi_2/2) / \tan(\pi/4 + \varphi_1/2)) \quad \text{('projected' latitude difference)}$$
$$q = \Delta\varphi/\Delta\psi \text{ (or } \cos\varphi \text{ for E-W line)}$$
$$\Delta\lambda = \delta \cdot \sin\theta / q$$
$$\varphi_2 = \varphi_1 + \delta \cdot \cos\theta$$
$$\lambda_2 = \lambda_1 + \Delta\lambda$$

where φ *is latitude*, λ *is longitude*, $\Delta\lambda$ *is taking shortest route (<180°),* ln *is natural log,* R *is the earth's radius*

JavaScript: *(all angles in radians)*

```
var Δφ = δ*Math.cos(θ);
var φ2 = φ1 + Δλ;

var Δψ = Math.log(Math.tan(φ2/2+Math.PI/4)/Math.tan(φ1/2+Math.PI/4));
var q = Math.abs(Δψ) > 10e-12 ? Δφ / Δψ : Math.cos(φ1); // E-W course becomes ill-conditioned with 0/0
```

```
var Δλ = δ*Math.sin(θ)/q;
var λ2 = λ1 + Δλ;

// check for some daft bugger going past the pole, normalise latitude if so
if (Math.abs(φ2) > Math.PI/2) φ2 = φ2>0 ? Math.PI-φ2 : -Math.PI-φ2;
```

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

Mid-point

This formula for calculating the 'loxodromic midpoint', the point half-way along a rhumb line between two points, is due to Robert Hill and Clive Tooth[1] (thx Axel!).

Formula:

$$\varphi_m = (\varphi_1+\varphi_2) / 2$$
$$f_1 = \tan(\pi/4 + \varphi_1/2)$$
$$f_2 = \tan(\pi/4 + \varphi_2/2)$$
$$f_m = \tan(\pi/4+\varphi_m/2)$$
$$\lambda_m = [(\lambda_2-\lambda_1) \cdot \ln(f_m) + \lambda_1 \cdot \ln(f_2) - \lambda_2 \cdot \ln(f_1)] / \ln(f_2/f_1)$$

where φ *is latitude*, λ *is longitude*, ln *is natural log*

JavaScript: *(all angles in radians)*

```
if (Math.abs(λ2-λ1) > Math.PI) λ1 += 2*Math.PI; // crossing anti-meridian

var φ3 = (φ1+φ2)/2;
var f1 = Math.tan(Math.PI/4 + φ1/2);
var f2 = Math.tan(Math.PI/4 + φ2/2);
var f3 = Math.tan(Math.PI/4 + φ3/2);
var λ3 = ( (λ2-λ1)*Math.log(f3) + λ1*Math.log(f2) - λ2*Math.log(f1) ) / Math.log(f2/f1);

if (!isFinite(λ3)) λ3 = (λ1+λ2)/2; // parallel of latitude
```

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

Using the scripts in web pages

Using these scripts in web pages would be something like the following:

```
<!doctype html>
<html lang="en">
<head>
    <title>Using the scripts in web pages</title>
    <meta charset="utf-8">
    <script defer src="//cdn.rawgit.com/chrisveness/geodesy/v1.1.1/latlon-spherical.js"></script>
    <script defer src="//cdn.rawgit.com/chrisveness/geodesy/v1.1.1/dms.js"></script>
    <script>
        document.addEventListener('DOMContentLoaded', function() {
            document.querySelector('#calc-dist').onclick = function() {
                const lat1 = document.querySelector('#lat1').value;
                const lon1 = document.querySelector('#lon1').value;
                const lat2 = document.querySelector('#lat2').value;
                const lon2 = document.querySelector('#lon2').value;
                const p1 = new LatLon(Dms.parseDMS(lat1), Dms.parseDMS(lon1));
                const p2 = new LatLon(Dms.parseDMS(lat2), Dms.parseDMS(lon2));
                const dist = parseFloat(p1.distanceTo(p2).toPrecision(4));
                document.querySelector('#result-distance').textContent = dist;
            }
        });
    </script>
</head>
<body>
    <form>
        Lat 1: <input type="text" name="lat1" id="lat1">
        Lon 1: <input type="text" name="lon1" id="lon1">
        Lat 2: <input type="text" name="lat2" id="lat2">
        Lon 2: <input type="text" name="lon2" id="lon2">
        <button type="button" id="calc-dist">Calculate distance</button>
        <output id="result-distance"></output> metres
    </form>
</body>
</html>
```

Convert between degrees-minutes-seconds & decimal degrees

Latitude	Longitude	
52.20472°N	000.14056°E	1° ≈ 111 km (110.57 eq'l — 111.70 polar)

52°12.283′N	000°08.433′E	1′ ≈ 1.85 km (= 1 nm)	0.01° ≈ 1.11 km
52°12′17.0″N	000°08′26.0″E	1″ ≈ 30.9 m	0.0001° ≈ 11.1 m

Display calculation results as: ○ degrees ○ deg/min ◉ deg/min/sec

Notes:

- Accuracy: since the earth is not quite a sphere, there are small errors in using spherical geometry; the earth is actually roughly **ellipsoidal** (or more precisely, oblate spheroidal) with a radius varying between about 6,378km (equatorial) and 6,357km (polar), and local radius of curvature varying from 6,336km (equatorial meridian) to 6,399km (polar). 6,371 km is the generally accepted value for the earth's mean radius. This means that errors from assuming spherical geometry might be up to 0.55% crossing the equator, though generally below 0.3%, depending on latitude and direction of travel (*whuber* explores this in excellent detail on stackexchange). An accuracy of better than 3m in 1km is mostly good enough for me, but if you want greater accuracy, you could use the Vincenty formula for calculating geodesic distances on ellipsoids, which gives results accurate to within 1mm. (Out of sheer perversity – I've never needed such accuracy – I looked up this formula and discovered the JavaScript implementation was simpler than I expected).
- Trig functions take arguments in **radians**, so latitude, longitude, and bearings in **degrees** (either decimal or degrees/minutes/seconds) need to be converted to radians, rad = π.deg/180. When converting radians back to degrees (deg = 180.rad/π), West is negative if using signed decimal degrees. For bearings, values in the range -π to +π [-180° to +180°] need to be converted to 0 to +2π [0°–360°]; this can be done by (brng+2.π)%2.π [or brng+360)%360] where % is the (floating point) modulo operator (note that different languages implement the modulo operation in different ways).
- All bearings are with respect to **true north**, 0°=N, 90°=E, etc; if you are working from a compass, magnetic north varies from true north in a complex way around the earth, and the difference has to be compensated for by variances indicated on local maps.
- The **atan2**() function widely used here takes two arguments, atan2(y, x), and computes the arc tangent of the ratio y/x. It is more flexible than atan(y/x), since it handles x=0, and it also returns values in all 4 quadrants -π to +π (the atan function returns values in the range -π/2 to +π/2).
- If you implement any formula involving atan2 in a spreadsheet (Microsoft **Excel**, LibreOffice Calc, Google Sheets, Apple Numbers), you will need to reverse the arguments, as Excel etc have them the opposite way around from JavaScript – conventional order is atan2(y, x), but Excel uses atan2(x, y). To use atan2 in a (VBA) macro, you can use WorksheetFunction.Atan2().
- If you are using **Google Maps**, several of these functions are now provided in the Google Maps API V3 'spherical' library (computeDistanceBetween(), computeHeading(), computeOffset(), interpolate(), etc; note they use a default Earth radius of 6,378,137 meters).
- If you use UK Ordnance Survey Grid References, I have implemented a script for converting between Lat/Long & OS Grid References.
- If you use UTM coordinates or MGRS grid references, I have implemented scripts for converting between Lat/Long, UTM, & MGRS.
- I learned a lot from the US Census Bureau GIS FAQ which is no longer available, so I've made a copy.
- Thanks to Ed Williams' Aviation Formulary for many of the formulæ.
- For **miles**, divide km by 1.609344
- For **nautical miles**, divide km by 1.852

See below for the JavaScript source code, also available on GitHub. Note I use Greek letters in variables representing maths symbols conventionally presented as Greek letters: I value the great benefit in legibility over the minor inconvenience in typing (if you encounter any problems, ensure your `<head>` includes `<meta charset="utf-8">`).

With its untyped C-style syntax, JavaScript reads remarkably close to pseudo-code: exposing the algorithms with a minimum of syntactic distractions. These functions should be simple to translate into other languages if required, though can also be used as-is in browsers and Node.js.

I have extended the base JavaScript Number object with toRadians() and toDegrees() methods: I don't see great likelihood of conflicts, as these are ubiquitous operations.

I also have a page illustrating the use of the spherical law of cosines for selecting points from a database within a specified bounding circle – the example is based on MySQL+PDO, but should be extensible to other DBMS platforms.

Several people have asked about example **Excel** spreadsheets, so I have implemented the distance & bearing and the destination point formulæ as spreadsheets, in a form which breaks down the all stages involved to illustrate the operation.

January 2010: I have revised the scripts to be structured as methods of a LatLon object. Of course, while JavaScript is object-oriented, it is a prototype-based rather than class-based language, so this is not actually a class, but isolating code into a separate namespace is good JavaScript practice. If you're not familiar with JavaScript syntax, `LatLon.prototype.distanceTo = function(point) { ... }`, for instance, defines a '`distanceTo`' method of the `LatLon` object (/class) which takes a `LatLon` object as a parameter (and returns a number). The *Dms* namespace acts as a static class for geodesy formatting / parsing / conversion functions.

January 2015: I have refactored the scripts to inter-operate better, and rationalised certain aspects: the JavaScript file is now latlon-spherical.js instead of simply latlon.js; distances are now always in metres; the earth's radius is now a parameter to distance calculation methods rather than to the constructor; the previous `Geo` object is now `Dms`, to better indicate its purpose; the `destinationPoint` function has the distance parameter before the bearing.

Performance: as noted above, the haversine distance calculation takes around 2 – 5 microseconds (hence around 200,000 – 500,000 per second). I have yet to complete timing tests on other calculations.

Other languages: I cannot support translations into other languages, but if you have ported the code to another language, I am happy to provide links here.

- Brian Lambert has made an Objective-C version.
- Jean Brouwers has made a Python version.

I offer these scripts for free use and adaptation to balance my debt to the open-source info-verse. You are welcome to re-use these scripts [under an MIT licence, without any warranty express or implied] provided solely that you retain my copyright notice and a link to this page.

If you would like to show your appreciation and support continued development of these scripts, I would most gratefully accept donations.

If you need any advice or development work done, I am available for consultancy.

If you have any queries or find any problems, contact me at scripts-geo@movable-type.co.uk.


```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/* Latitude/longitude spherical geodesy tools                  (c) Chris Veness 2002-2016  */
/*                                                                           MIT Licence  */
/* www.movable-type.co.uk/scripts/latlong.html                                            */
/* www.movable-type.co.uk/scripts/geodesy/docs/module-latlon-spherical.html               */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

'use strict';
if (typeof module!='undefined' && module.exports) var Dms = require('./dms'); // ≡ import Dms from 'dms.js'


/**
 * Library of geodesy functions for operations on a spherical earth model.
 *
 * @module   latlon-spherical
 * @requires dms
 */


/**
 * Creates a LatLon point on the earth's surface at the specified latitude / longitude.
 *
 * @constructor
 * @param {number} lat - Latitude in degrees.
 * @param {number} lon - Longitude in degrees.
 *
 * @example
 *     var p1 = new LatLon(52.205, 0.119);
 */
function LatLon(lat, lon) {
    // allow instantiation without 'new'
    if (!(this instanceof LatLon)) return new LatLon(lat, lon);

    this.lat = Number(lat);
    this.lon = Number(lon);
}


/**
 * Returns the distance from 'this' point to destination point (using haversine formula).
 *
 * @param   {LatLon} point - Latitude/longitude of destination point.
 * @param   {number} [radius=6371e3] - (Mean) radius of earth (defaults to radius in metres).
 * @returns {number} Distance between this point and destination point, in same units as radius.
 *
 * @example
 *     var p1 = new LatLon(52.205, 0.119);
 *     var p2 = new LatLon(48.857, 2.351);
 *     var d = p1.distanceTo(p2); // 404.3 km
 */
LatLon.prototype.distanceTo = function(point, radius) {
    if (!(point instanceof LatLon)) throw new TypeError('point is not LatLon object');
    radius = (radius === undefined) ? 6371e3 : Number(radius);

    var R = radius;
    var φ1 = this.lat.toRadians(),  λ1 = this.lon.toRadians();
```

```
    var φ2 = point.lat.toRadians(), λ2 = point.lon.toRadians();
    var Δφ = φ2 - φ1;
    var Δλ = λ2 - λ1;

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

    return d;
};


/**
 * Returns the (initial) bearing from ‘this’ point to destination point.
 *
 * @param   {LatLon} point - Latitude/longitude of destination point.
 * @returns {number} Initial bearing in degrees from north.
 *
 * @example
 *     var p1 = new LatLon(52.205, 0.119);
 *     var p2 = new LatLon(48.857, 2.351);
 *     var b1 = p1.bearingTo(p2); // 156.2°
 */
LatLon.prototype.bearingTo = function(point) {
    if (!(point instanceof LatLon)) throw new TypeError('point is not LatLon object');

    var φ1 = this.lat.toRadians(), φ2 = point.lat.toRadians();
    var Δλ = (point.lon-this.lon).toRadians();

    // see http://mathforum.org/library/drmath/view/55417.html
    var y = Math.sin(Δλ) * Math.cos(φ2);
    var x = Math.cos(φ1)*Math.sin(φ2) -
            Math.sin(φ1)*Math.cos(φ2)*Math.cos(Δλ);
    var θ = Math.atan2(y, x);

    return (θ.toDegrees()+360) % 360;
};


/**
 * Returns final bearing arriving at destination destination point from ‘this’ point; the final bearing
 * will differ from the initial bearing by varying degrees according to distance and latitude.
 *
 * @param   {LatLon} point - Latitude/longitude of destination point.
 * @returns {number} Final bearing in degrees from north.
 *
 * @example
 *     var p1 = new LatLon(52.205, 0.119);
 *     var p2 = new LatLon(48.857, 2.351);
 *     var b2 = p1.finalBearingTo(p2); // 157.9°
 */
LatLon.prototype.finalBearingTo = function(point) {
    if (!(point instanceof LatLon)) throw new TypeError('point is not LatLon object');

    // get initial bearing from destination point to this point & reverse it by adding 180°
    return ( point.bearingTo(this)+180 ) % 360;
};


/**
 * Returns the midpoint between ‘this’ point and the supplied point.
 *
 * @param   {LatLon} point - Latitude/longitude of destination point.
 * @returns {LatLon} Midpoint between this point and the supplied point.
 *
 * @example
 *     var p1 = new LatLon(52.205, 0.119);
 *     var p2 = new LatLon(48.857, 2.351);
 *     var pMid = p1.midpointTo(p2); // 50.5363°N, 001.2746°E
 */
LatLon.prototype.midpointTo = function(point) {
    if (!(point instanceof LatLon)) throw new TypeError('point is not LatLon object');

    // φm = atan2( sinφ1 + sinφ2, √( (cosφ1 + cosφ2·cosΔλ) · (cosφ1 + cosφ2·cosΔλ) ) + cos²φ2·sin²Δλ )
    // λm = λ1 + atan2(cosφ2·sinΔλ, cosφ1 + cosφ2·cosΔλ)
    // see http://mathforum.org/library/drmath/view/51822.html for derivation

    var φ1 = this.lat.toRadians(), λ1 = this.lon.toRadians();
    var φ2 = point.lat.toRadians();
    var Δλ = (point.lon-this.lon).toRadians();

    var Bx = Math.cos(φ2) * Math.cos(Δλ);
    var By = Math.cos(φ2) * Math.sin(Δλ);

    var x = Math.sqrt((Math.cos(φ1) + Bx) * (Math.cos(φ1) + Bx) + By * By);
    var y = Math.sin(φ1) + Math.sin(φ2);
    var φ3 = Math.atan2(y, x);

    var λ3 = λ1 + Math.atan2(By, Math.cos(φ1) + Bx);
```

```
    return new LatLon(ϕ3.toDegrees(), (λ3.toDegrees()+540)%360-180); // normalise to −180..+180°
};


/**
 * Returns the point at given fraction between 'this' point and specified point.
 *
 * @param   {LatLon} point - Latitude/longitude of destination point.
 * @param   {number} fraction - Fraction between the two points (0 = this point, 1 = specified point).
 * @returns {LatLon} Intermediate point between this point and destination point.
 *
 * @example
 *   let p1 = new LatLon(52.205, 0.119);
 *   let p2 = new LatLon(48.857, 2.351);
 *   let pMid = p1.intermediatePointTo(p2, 0.25); // 51.3721°N, 000.7073°E
 */
LatLon.prototype.intermediatePointTo = function(point, fraction) {
    if (!(point instanceof LatLon)) throw new TypeError('point is not LatLon object');

    var ϕ1 = this.lat.toRadians(), λ1 = this.lon.toRadians();
    var ϕ2 = point.lat.toRadians(), λ2 = point.lon.toRadians();
    var sinϕ1 = Math.sin(ϕ1), cosϕ1 = Math.cos(ϕ1), sinλ1 = Math.sin(λ1), cosλ1 = Math.cos(λ1);
    var sinϕ2 = Math.sin(ϕ2), cosϕ2 = Math.cos(ϕ2), sinλ2 = Math.sin(λ2), cosλ2 = Math.cos(λ2);

    // distance between points
    var Δϕ = ϕ2 - ϕ1;
    var Δλ = λ2 - λ1;
    var a = Math.sin(Δϕ/2) * Math.sin(Δϕ/2)
        + Math.cos(ϕ1) * Math.cos(ϕ2) * Math.sin(Δλ/2) * Math.sin(Δλ/2);
    var δ = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));

    var A = Math.sin((1-fraction)*δ) / Math.sin(δ);
    var B = Math.sin(fraction*δ) / Math.sin(δ);

    var x = A * cosϕ1 * cosλ1 + B * cosϕ2 * cosλ2;
    var y = A * cosϕ1 * sinλ1 + B * cosϕ2 * sinλ2;
    var z = A * sinϕ1 + B * sinϕ2;

    var ϕ3 = Math.atan2(z, Math.sqrt(x*x + y*y));
    var λ3 = Math.atan2(y, x);

    return new LatLon(ϕ3.toDegrees(), (λ3.toDegrees()+540)%360-180); // normalise lon to −180..+180°
};


/**
 * Returns the destination point from 'this' point having travelled the given distance on the
 * given initial bearing (bearing normally varies around path followed).
 *
 * @param   {number} distance - Distance travelled, in same units as earth radius (default: metres).
 * @param   {number} bearing - Initial bearing in degrees from north.
 * @param   {number} [radius=6371e3] - (Mean) radius of earth (defaults to radius in metres).
 * @returns {LatLon} Destination point.
 *
 * @example
 *     var p1 = new LatLon(51.4778, -0.0015);
 *     var p2 = p1.destinationPoint(7794, 300.7); // 51.5135°N, 000.0983°W
 */
LatLon.prototype.destinationPoint = function(distance, bearing, radius) {
    radius = (radius === undefined) ? 6371e3 : Number(radius);

    // sinϕ2 = sinϕ1·cosδ + cosϕ1·sinδ·cosθ
    // tanΔλ = sinθ·sinδ·cosϕ1 / cosδ−sinϕ1·sinϕ2
    // see http://williams.best.vwh.net/avform.htm#LL

    var δ = Number(distance) / radius; // angular distance in radians
    var θ = Number(bearing).toRadians();

    var ϕ1 = this.lat.toRadians();
    var λ1 = this.lon.toRadians();

    var sinϕ1 = Math.sin(ϕ1), cosϕ1 = Math.cos(ϕ1);
    var sinδ = Math.sin(δ), cosδ = Math.cos(δ);
    var sinθ = Math.sin(θ), cosθ = Math.cos(θ);

    var sinϕ2 = sinϕ1*cosδ + cosϕ1*sinδ*cosθ;
    var ϕ2 = Math.asin(sinϕ2);
    var y = sinθ * sinδ * cosϕ1;
    var x = cosδ - sinϕ1 * sinϕ2;
    var λ2 = λ1 + Math.atan2(y, x);

    return new LatLon(ϕ2.toDegrees(), (λ2.toDegrees()+540)%360-180); // normalise to −180..+180°
};


/**
 * Returns the point of intersection of two paths defined by point and bearing.
 *
 * @param   {LatLon} p1 - First point.
 * @param   {number} brng1 - Initial bearing from first point.
```

```
 * @param   {LatLon} p2 - Second point.
 * @param   {number} brng2 - Initial bearing from second point.
 * @returns {LatLon|null} Destination point (null if no unique intersection defined).
 *
 * @example
 *     var p1 = LatLon(51.8853, 0.2545), brng1 = 108.547;
 *     var p2 = LatLon(49.0034, 2.5735), brng2 =  32.435;
 *     var pInt = LatLon.intersection(p1, brng1, p2, brng2); // 50.9078°N, 004.5084°E
 */
LatLon.intersection = function(p1, brng1, p2, brng2) {
    if (!(p1 instanceof LatLon)) throw new TypeError('p1 is not LatLon object');
    if (!(p2 instanceof LatLon)) throw new TypeError('p2 is not LatLon object');

    // see http://williams.best.vwh.net/avform.htm#Intersection

    var φ1 = p1.lat.toRadians(), λ1 = p1.lon.toRadians();
    var φ2 = p2.lat.toRadians(), λ2 = p2.lon.toRadians();
    var θ13 = Number(brng1).toRadians(), θ23 = Number(brng2).toRadians();
    var Δφ = φ2-φ1, Δλ = λ2-λ1;

    var δ12 = 2*Math.asin( Math.sqrt( Math.sin(Δφ/2)*Math.sin(Δφ/2)
        + Math.cos(φ1)*Math.cos(φ2)*Math.sin(Δλ/2)*Math.sin(Δλ/2) ) );
    if (δ12 == 0) return null;

    // initial/final bearings between points
    var θa = Math.acos( ( Math.sin(φ2) - Math.sin(φ1)*Math.cos(δ12) ) / ( Math.sin(δ12)*Math.cos(φ1) ) );
    if (isNaN(θa)) θa = 0; // protect against rounding
    var θb = Math.acos( ( Math.sin(φ1) - Math.sin(φ2)*Math.cos(δ12) ) / ( Math.sin(δ12)*Math.cos(φ2) ) );

    var θ12 = Math.sin(λ2-λ1)>0 ? θa : 2*Math.PI-θa;
    var θ21 = Math.sin(λ2-λ1)>0 ? 2*Math.PI-θb : θb;

    var α1 = (θ13 - θ12 + Math.PI) % (2*Math.PI) - Math.PI; // angle 2-1-3
    var α2 = (θ21 - θ23 + Math.PI) % (2*Math.PI) - Math.PI; // angle 1-2-3

    if (Math.sin(α1)==0 && Math.sin(α2)==0) return null; // infinite intersections
    if (Math.sin(α1)*Math.sin(α2) < 0) return null;      // ambiguous intersection

    //α1 = Math.abs(α1);
    //α2 = Math.abs(α2);
    // ... Ed Williams takes abs of α1/α2, but seems to break calculation?

    var α3 = Math.acos( -Math.cos(α1)*Math.cos(α2) + Math.sin(α1)*Math.sin(α2)*Math.cos(δ12) );
    var δ13 = Math.atan2( Math.sin(δ12)*Math.sin(α1)*Math.sin(α2), Math.cos(α2)+Math.cos(α1)*Math.cos(α3) );
    var φ3 = Math.asin( Math.sin(φ1)*Math.cos(δ13) + Math.cos(φ1)*Math.sin(δ13)*Math.cos(θ13) );
    var Δλ13 = Math.atan2( Math.sin(θ13)*Math.sin(δ13)*Math.cos(φ1), Math.cos(δ13)-Math.sin(φ1)*Math.sin(φ3) );
    var λ3 = λ1 + Δλ13;

    return new LatLon(φ3.toDegrees(), (λ3.toDegrees()+540)%360-180); // normalise to −180..+180°
};


/**
 * Returns (signed) distance from 'this' point to great circle defined by start-point and end-point.
 *
 * @param   {LatLon} pathStart - Start point of great circle path.
 * @param   {LatLon} pathEnd - End point of great circle path.
 * @param   {number} [radius=6371e3] - (Mean) radius of earth (defaults to radius in metres).
 * @returns {number} Distance to great circle (-ve if to left, +ve if to right of path).
 *
 * @example
 *   var pCurrent = new LatLon(53.2611, -0.7972);
 *   var p1 = new LatLon(53.3206, -1.7297);
 *   var p2 = new LatLon(53.1887,  0.1334);
 *   var d = pCurrent.crossTrackDistanceTo(p1, p2);  // -307.5 m
 */
LatLon.prototype.crossTrackDistanceTo = function(pathStart, pathEnd, radius) {
    if (!(pathStart instanceof LatLon)) throw new TypeError('pathStart is not LatLon object');
    if (!(pathEnd instanceof LatLon)) throw new TypeError('pathEnd is not LatLon object');
    radius = (radius === undefined) ? 6371e3 : Number(radius);

    var δ13 = pathStart.distanceTo(this, radius)/radius;
    var θ13 = pathStart.bearingTo(this).toRadians();
    var θ12 = pathStart.bearingTo(pathEnd).toRadians();

    var dxt = Math.asin( Math.sin(δ13) * Math.sin(θ13-θ12) ) * radius;

    return dxt;
};


/**
 * Returns maximum latitude reached when travelling on a great circle on given bearing from this
 * point ('Clairaut's formula'). Negate the result for the minimum latitude (in the Southern
 * hemisphere).
 *
 * The maximum latitude is independent of longitude; it will be the same for all points on a given
 * latitude.
 *
 * @param {number} bearing - Initial bearing.
 * @param {number} latitude - Starting latitude.
```

```
 */
LatLon.prototype.maxLatitude = function(bearing) {
    var θ = Number(bearing).toRadians();

    var φ = this.lat.toRadians();

    var φMax = Math.acos(Math.abs(Math.sin(θ)*Math.cos(φ)));

    return φMax.toDegrees();
};


/**
 * Returns the pair of meridians at which a great circle defined by two points crosses the given
 * latitude. If the great circle doesn't reach the given latitude, null is returned.
 *
 * @param {LatLon} point1 - First point defining great circle.
 * @param {LatLon} point2 - Second point defining great circle.
 * @param {number} latitude - Latitude crossings are to be determined for.
 * @returns {Object|null} Object containing { lon1, lon2 } or null if given latitude not reached.
 */
LatLon.crossingParallels = function(point1, point2, latitude) {
    var φ = Number(latitude).toRadians();

    var φ1 = point1.lat.toRadians();
    var λ1 = point1.lon.toRadians();
    var φ2 = point2.lat.toRadians();
    var λ2 = point2.lon.toRadians();

    var Δλ = λ2 - λ1;

    var x = Math.sin(φ1) * Math.cos(φ2) * Math.cos(φ) * Math.sin(Δλ);
    var y = Math.sin(φ1) * Math.cos(φ2) * Math.cos(φ) * Math.cos(Δλ) - Math.cos(φ1) * Math.sin(φ2) * Math.cos(φ);
    var z = Math.cos(φ1) * Math.cos(φ2) * Math.sin(φ) * Math.sin(Δλ);

    if (z*z > x*x + y*y) return null; // great circle doesn't reach latitude

    var λm = Math.atan2(-y, x);                  // longitude at max latitude
    var Δλi = Math.acos(z / Math.sqrt(x*x+y*y)); // Δλ from λm to intersection points

    var λi1 = λ1 + λm - Δλi;
    var λi2 = λ1 + λm + Δλi;

    return { lon1: (λi1.toDegrees()+540)%360-180, lon2: (λi2.toDegrees()+540)%360-180 }; // normalise to −180..+180°
};


/* Rhumb - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

/**
 * Returns the distance travelling from ‘this’ point to destination point along a rhumb line.
 *
 * @param   {LatLon} point - Latitude/longitude of destination point.
 * @param   {number} [radius=6371e3] - (Mean) radius of earth (defaults to radius in metres).
 * @returns {number} Distance in km between this point and destination point (same units as radius).
 *
 * @example
 *     var p1 = new LatLon(51.127, 1.338);
 *     var p2 = new LatLon(50.964, 1.853);
 *     var d = p1.distanceTo(p2); // 40.31 km
 */
LatLon.prototype.rhumbDistanceTo = function(point, radius) {
    if (!(point instanceof LatLon)) throw new TypeError('point is not LatLon object');
    radius = (radius === undefined) ? 6371e3 : Number(radius);

    // see http://williams.best.vwh.net/avform.htm#Rhumb

    var R = radius;
    var φ1 = this.lat.toRadians(), φ2 = point.lat.toRadians();
    var Δφ = φ2 - φ1;
    var Δλ = Math.abs(point.lon-this.lon).toRadians();
    // if dLon over 180° take shorter rhumb line across the anti-meridian:
    if (Math.abs(Δλ) > Math.PI) Δλ = Δλ>0 ? -(2*Math.PI-Δλ) : (2*Math.PI+Δλ);

    // on Mercator projection, longitude distances shrink by latitude; q is the 'stretch factor'
    // q becomes ill-conditioned along E-W line (0/0); use empirical tolerance to avoid it
    var Δψ = Math.log(Math.tan(φ2/2+Math.PI/4)/Math.tan(φ1/2+Math.PI/4));
    var q = Math.abs(Δψ) > 10e-12 ? Δφ/Δψ : Math.cos(φ1);

    // distance is pythagoras on 'stretched' Mercator projection
    var δ = Math.sqrt(Δφ*Δφ + q*q*Δλ*Δλ); // angular distance in radians
    var dist = δ * R;

    return dist;
};


/**
 * Returns the bearing from ‘this’ point to destination point along a rhumb line.
 *
 * @param   {LatLon} point - Latitude/longitude of destination point.
```

```
 * @returns {number} Bearing in degrees from north.
 *
 * @example
 *     var p1 = new LatLon(51.127, 1.338);
 *     var p2 = new LatLon(50.964, 1.853);
 *     var d = p1.rhumbBearingTo(p2); // 116.7 m
 */
LatLon.prototype.rhumbBearingTo = function(point) {
    if (!(point instanceof LatLon)) throw new TypeError('point is not LatLon object');

    var φ1 = this.lat.toRadians(), φ2 = point.lat.toRadians();
    var Δλ = (point.lon-this.lon).toRadians();
    // if dLon over 180° take shorter rhumb line across the anti-meridian:
    if (Math.abs(Δλ) > Math.PI) Δλ = Δλ>0 ? -(2*Math.PI-Δλ) : (2*Math.PI+Δλ);

    var Δψ = Math.log(Math.tan(φ2/2+Math.PI/4)/Math.tan(φ1/2+Math.PI/4));

    var θ = Math.atan2(Δλ, Δψ);

    return (θ.toDegrees()+360) % 360;
};


/**
 * Returns the destination point having travelled along a rhumb line from ‘this’ point the given
 * distance on the  given bearing.
 *
 * @param   {number} distance - Distance travelled, in same units as earth radius (default: metres).
 * @param   {number} bearing - Bearing in degrees from north.
 * @param   {number} [radius=6371e3] - (Mean) radius of earth (defaults to radius in metres).
 * @returns {LatLon} Destination point.
 *
 * @example
 *     var p1 = new LatLon(51.127, 1.338);
 *     var p2 = p1.rhumbDestinationPoint(40300, 116.7); // 50.9642°N, 001.8530°E
 */
LatLon.prototype.rhumbDestinationPoint = function(distance, bearing, radius) {
    radius = (radius === undefined) ? 6371e3 : Number(radius);

    var δ = Number(distance) / radius; // angular distance in radians
    var φ1 = this.lat.toRadians(), λ1 = this.lon.toRadians();
    var θ = Number(bearing).toRadians();

    var Δφ = δ * Math.cos(θ);
    var φ2 = φ1 + Δφ;

    // check for some daft bugger going past the pole, normalise latitude if so
    if (Math.abs(φ2) > Math.PI/2) φ2 = φ2>0 ? Math.PI-φ2 : -Math.PI-φ2;

    var Δψ = Math.log(Math.tan(φ2/2+Math.PI/4)/Math.tan(φ1/2+Math.PI/4));
    var q = Math.abs(Δψ) > 10e-12 ? Δφ / Δψ : Math.cos(φ1); // E-W course becomes ill-conditioned with 0/0

    var Δλ = δ*Math.sin(θ)/q;
    var λ2 = λ1 + Δλ;

    return new LatLon(φ2.toDegrees(), (λ2.toDegrees()+540) % 360 - 180); // normalise to −180..+180°
};


/**
 * Returns the loxodromic midpoint (along a rhumb line) between ‘this’ point and second point.
 *
 * @param   {LatLon} point - Latitude/longitude of second point.
 * @returns {LatLon} Midpoint between this point and second point.
 *
 * @example
 *     var p1 = new LatLon(51.127, 1.338);
 *     var p2 = new LatLon(50.964, 1.853);
 *     var pMid = p1.rhumbMidpointTo(p2); // 51.0455°N, 001.5957°E
 */
LatLon.prototype.rhumbMidpointTo = function(point) {
    if (!(point instanceof LatLon)) throw new TypeError('point is not LatLon object');

    // http://mathforum.org/kb/message.jspa?messageID=148837

    var φ1 = this.lat.toRadians(), λ1 = this.lon.toRadians();
    var φ2 = point.lat.toRadians(), λ2 = point.lon.toRadians();

    if (Math.abs(λ2-λ1) > Math.PI) λ1 += 2*Math.PI; // crossing anti-meridian

    var φ3 = (φ1+φ2)/2;
    var f1 = Math.tan(Math.PI/4 + φ1/2);
    var f2 = Math.tan(Math.PI/4 + φ2/2);
    var f3 = Math.tan(Math.PI/4 + φ3/2);
    var λ3 = ( (λ2-λ1)*Math.log(f3) + λ1*Math.log(f2) - λ2*Math.log(f1) ) / Math.log(f2/f1);

    if (!isFinite(λ3)) λ3 = (λ1+λ2)/2; // parallel of latitude

    var p = LatLon(φ3.toDegrees(), (λ3.toDegrees()+540)%360-180); // normalise to −180..+180°

    return p;
```

```
};


/* Area - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */


/**
 * Calculates the area of a spherical polygon where the sides of the polygon are great circle
 * arcs joining the vertices.
 *
 * @param   {LatLon[]} polygon - Array of points defining vertices of the polygon
 * @param   {number} [radius=6371e3] - (Mean) radius of earth (defaults to radius in metres).
 * @returns {number} The area of the polygon, in the same units as radius.
 *
 * @example
 *   var polygon = [new LatLon(0,0), new LatLon(1,0), new LatLon(0,1)];
 *   var area = LatLon.areaOf(polygon); // 6.18e9 m²
 */
LatLon.areaOf = function(polygon, radius) {
    // uses method due to Karney: osgeo-org.1560.x6.nabble.com/Area-of-a-spherical-polygon-td3841625.html;
    // for each edge of the polygon, tan(E/2) = tan(Δλ/2)·(tan(φ1/2) + tan(φ2/2)) / (1 + tan(φ1/2)·tan(φ2/2))
    // where E is the spherical excess of the trapezium obtained by extending the edge to the equator

    var R = (radius === undefined) ? 6371e3 : Number(radius);

    // close polygon so that last point equals first point
    var closed = polygon[0].equals(polygon[polygon.length-1]);
    if (!closed) polygon.push(polygon[0]);

    var nVertices = polygon.length - 1;

    var S = 0; // spherical excess in steradians
    for (var v=0; v<nVertices; v++) {
        var φ1 = polygon[v].lat.toRadians();
        var φ2 = polygon[v+1].lat.toRadians();
        var Δλ = (polygon[v+1].lon - polygon[v].lon).toRadians();
        var E = 2 * Math.atan2(Math.tan(Δλ/2) * (Math.tan(φ1/2)+Math.tan(φ2/2)), 1 + Math.tan(φ1/2)*Math.tan(φ2/2));
        S += E;
    }

    if (isPoleEnclosedBy(polygon)) S = Math.abs(S) - 2*Math.PI;

    var A = Math.abs(S * R*R); // area in units of R

    if (!closed) polygon.pop(); // restore polygon to pristine condition

    return A;

    // returns whether polygon encloses pole: sum of course deltas around pole is 0° rather than
    // normal ±360°: blog.element84.com/determining-if-a-spherical-polygon-contains-a-pole.html
    function isPoleEnclosedBy(polygon) {
        // TODO: any better test than this?
        var ΣΔ = 0;
        var prevBrng = polygon[0].bearingTo(polygon[1]);
        for (var v=0; v<polygon.length-1; v++) {
            var initBrng = polygon[v].bearingTo(polygon[v+1]);
            var finalBrng = polygon[v].finalBearingTo(polygon[v+1]);
            ΣΔ += (initBrng - prevBrng + 540) % 360 - 180;
            ΣΔ += (finalBrng - initBrng + 540) % 360 - 180;
            prevBrng = finalBrng;
        }
        var initBrng = polygon[0].bearingTo(polygon[1]);
        ΣΔ += (initBrng - prevBrng + 540) % 360 - 180;
        var enclosed = Math.abs(ΣΔ) < 90; // 0°-ish
        return enclosed;
    }
};


/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */


/**
 * Checks if another point is equal to 'this' point.
 *
 * @param   {LatLon} point - Point to be compared against this point.
 * @returns {bool}   True if points are identical.
 *
 * @example
 *   var p1 = new LatLon(52.205, 0.119);
 *   var p2 = new LatLon(52.205, 0.119);
 *   var equal = p1.equals(p2); // true
 */
LatLon.prototype.equals = function(point) {
    if (!(point instanceof LatLon)) throw new TypeError('point is not LatLon object');

    if (this.lat != point.lat) return false;
    if (this.lon != point.lon) return false;

    return true;
};
```

```
/**
 * Returns a string representation of 'this' point, formatted as degrees, degrees+minutes, or
 * degrees+minutes+seconds.
 *
 * @param   {string} [format=dms] - Format point as 'd', 'dm', 'dms'.
 * @param   {number} [dp=0|2|4] - Number of decimal places to use - default 0 for dms, 2 for dm, 4 for d.
 * @returns {string} Comma-separated latitude/longitude.
 */
LatLon.prototype.toString = function(format, dp) {
    return Dms.toLat(this.lat, format, dp) + ', ' + Dms.toLon(this.lon, format, dp);
};


/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

/** Extend Number object with method to convert numeric degrees to radians */
if (Number.prototype.toRadians === undefined) {
    Number.prototype.toRadians = function() { return this * Math.PI / 180; };
}

/** Extend Number object with method to convert radians to numeric (signed) degrees */
if (Number.prototype.toDegrees === undefined) {
    Number.prototype.toDegrees = function() { return this * 180 / Math.PI; };
}

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */
if (typeof module != 'undefined' && module.exports) module.exports = LatLon; // ≡ export default LatLon
```

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */
/* Geodesy representation conversion functions                        (c) Chris Veness 2002-2016  */
/*                                                                                   MIT Licence  */
/* www.movable-type.co.uk/scripts/latlong.html                                                    */
/* www.movable-type.co.uk/scripts/geodesy/docs/module-dms.html                                    */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

'use strict';
/* eslint no-irregular-whitespace: [2, { skipComments: true }] */


/**
 * Latitude/longitude points may be represented as decimal degrees, or subdivided into sexagesimal
 * minutes and seconds.
 *
 * @module dms
 */


/**
 * Functions for parsing and representing degrees / minutes / seconds.
 * @class Dms
 */
var Dms = {};

// note Unicode Degree = U+00B0. Prime = U+2032, Double prime = U+2033


/**
 * Parses string representing degrees/minutes/seconds into numeric degrees.
 *
 * This is very flexible on formats, allowing signed decimal degrees, or deg-min-sec optionally
 * suffixed by compass direction (NSEW). A variety of separators are accepted (eg 3° 37′ 09″W).
 * Seconds and minutes may be omitted.
 *
 * @param   {string|number} dmsStr - Degrees or deg/min/sec in variety of formats.
 * @returns {number} Degrees as decimal number.
 *
 * @example
 *     var lat = Dms.parseDMS('51° 28′ 40.12″ N');
 *     var lon = Dms.parseDMS('000° 00′ 05.31″ W');
 *     var p1 = new LatLon(lat, lon); // 51.4778°N, 000.0015°W
 */
Dms.parseDMS = function(dmsStr) {
    // check for signed decimal degrees without NSEW, if so return it directly
    if (typeof dmsStr == 'number' && isFinite(dmsStr)) return Number(dmsStr);

    // strip off any sign or compass dir'n & split out separate d/m/s
    var dms = String(dmsStr).trim().replace(/^-/, '').replace(/[NSEW]$/i, '').split(/[^0-9.,]+/);
    if (dms[dms.length-1]=='') dms.splice(dms.length-1);  // from trailing symbol

    if (dms == '') return NaN;

    // and convert to decimal degrees...
    var deg;
    switch (dms.length) {
        case 3:  // interpret 3-part result as d/m/s
            deg = dms[0]/1 + dms[1]/60 + dms[2]/3600;
            break;
        case 2:  // interpret 2-part result as d/m
```

```
            deg = dms[0]/1 + dms[1]/60;
            break;
        case 1:  // just d (possibly decimal) or non-separated dddmmss
            deg = dms[0];
            // check for fixed-width unseparated format eg 0033709W
            //if (/[NS]/i.test(dmsStr)) deg = '0' + deg;  // - normalise N/S to 3-digit degrees
            //if (/[0-9]{7}/.test(deg)) deg = deg.slice(0,3)/1 + deg.slice(3,5)/60 + deg.slice(5)/3600;
            break;
        default:
            return NaN;
    }
    if (/^-|[WS]$/i.test(dmsStr.trim())) deg = -deg; // take '-', west and south as -ve

    return Number(deg);
};


/**
 * Separator character to be used to separate degrees, minutes, seconds, and cardinal directions.
 *
 * Set to '\u202f' (narrow no-break space) for improved formatting.
 *
 * @example
 *   var p = new LatLon(51.2, 0.33);  // 51°12′00.0″N, 000°19′48.0″E
 *   Dms.separator = '\u202f';        // narrow no-break space
 *   var p' = new LatLon(51.2, 0.33); // 51° 12′ 00.0″ N, 000° 19′ 48.0″ E
 */
Dms.separator = '';


/**
 * Converts decimal degrees to deg/min/sec format
 *  - degree, prime, double-prime symbols are added, but sign is discarded, though no compass
 *    direction is added.
 *
 * @private
 * @param   {number} deg - Degrees to be formatted as specified.
 * @param   {string} [format=dms] - Return value as 'd', 'dm', 'dms' for deg, deg+min, deg+min+sec.
 * @param   {number} [dp=0|2|4] - Number of decimal places to use - default 0 for dms, 2 for dm, 4 for d.
 * @returns {string} Degrees formatted as deg/min/secs according to specified format.
 */
Dms.toDMS = function(deg, format, dp) {
    if (isNaN(deg)) return null;  // give up here if we can't make a number from deg

    // default values
    if (format === undefined) format = 'dms';
    if (dp === undefined) {
        switch (format) {
            case 'd':    case 'deg':         dp = 4; break;
            case 'dm':   case 'deg+min':     dp = 2; break;
            case 'dms':  case 'deg+min+sec': dp = 0; break;
            default:    format = 'dms'; dp = 0;  // be forgiving on invalid format
        }
    }

    deg = Math.abs(deg);  // (unsigned result ready for appending compass dir'n)

    var dms, d, m, s;
    switch (format) {
        default: // invalid format spec!
        case 'd': case 'deg':
            d = deg.toFixed(dp);    // round degrees
            if (d<100) d = '0' + d; // pad with leading zeros
            if (d<10) d = '0' + d;
            dms = d + '°';
            break;
        case 'dm': case 'deg+min':
            var min = (deg*60).toFixed(dp); // convert degrees to minutes & round
            d = Math.floor(min / 60);       // get component deg/min
            m = (min % 60).toFixed(dp);     // pad with trailing zeros
            if (d<100) d = '0' + d;         // pad with leading zeros
            if (d<10) d = '0' + d;
            if (m<10) m = '0' + m;
            dms = d + '°'+Dms.separator + m + '′';
            break;
        case 'dms': case 'deg+min+sec':
            var sec = (deg*3600).toFixed(dp); // convert degrees to seconds & round
            d = Math.floor(sec / 3600);       // get component deg/min/sec
            m = Math.floor(sec/60) % 60;
            s = (sec % 60).toFixed(dp);       // pad with trailing zeros
            if (d<100) d = '0' + d;           // pad with leading zeros
            if (d<10) d = '0' + d;
            if (m<10) m = '0' + m;
            if (s<10) s = '0' + s;
            dms = d + '°'+Dms.separator + m + '′'+Dms.separator + s + '″';
            break;
    }

    return dms;
};
```

```
/**
 * Converts numeric degrees to deg/min/sec latitude (2-digit degrees, suffixed with N/S).
 *
 * @param   {number} deg - Degrees to be formatted as specified.
 * @param   {string} [format=dms] - Return value as 'd', 'dm', 'dms' for deg, deg+min, deg+min+sec.
 * @param   {number} [dp=0|2|4] - Number of decimal places to use - default 0 for dms, 2 for dm, 4 for d.
 * @returns {string} Degrees formatted as deg/min/secs according to specified format.
 */
Dms.toLat = function(deg, format, dp) {
    var lat = Dms.toDMS(deg, format, dp);
    return lat===null ? '-' : lat.slice(1)+Dms.separator + (deg<0 ? 'S' : 'N');  // knock off initial '0' for lat!
};


/**
 * Convert numeric degrees to deg/min/sec longitude (3-digit degrees, suffixed with E/W)
 *
 * @param   {number} deg - Degrees to be formatted as specified.
 * @param   {string} [format=dms] - Return value as 'd', 'dm', 'dms' for deg, deg+min, deg+min+sec.
 * @param   {number} [dp=0|2|4] - Number of decimal places to use - default 0 for dms, 2 for dm, 4 for d.
 * @returns {string} Degrees formatted as deg/min/secs according to specified format.
 */
Dms.toLon = function(deg, format, dp) {
    var lon = Dms.toDMS(deg, format, dp);
    return lon===null ? '-' : lon+Dms.separator + (deg<0 ? 'W' : 'E');
};


/**
 * Converts numeric degrees to deg/min/sec as a bearing (0°..360°)
 *
 * @param   {number} deg - Degrees to be formatted as specified.
 * @param   {string} [format=dms] - Return value as 'd', 'dm', 'dms' for deg, deg+min, deg+min+sec.
 * @param   {number} [dp=0|2|4] - Number of decimal places to use - default 0 for dms, 2 for dm, 4 for d.
 * @returns {string} Degrees formatted as deg/min/secs according to specified format.
 */
Dms.toBrng = function(deg, format, dp) {
    deg = (Number(deg)+360) % 360;  // normalise -ve values to 180°..360°
    var brng =  Dms.toDMS(deg, format, dp);
    return brng===null ? '-' : brng.replace('360', '0');  // just in case rounding took us up to 360°!
};


/**
 * Returns compass point (to given precision) for supplied bearing.
 *
 * @param   {number} bearing - Bearing in degrees from north.
 * @param   {number} [precision=3] - Precision (1:cardinal / 2:intercardinal / 3:secondary-intercardinal).
 * @returns {string} Compass point for supplied bearing.
 *
 * @example
 *   var point = Dms.compassPoint(24);    // point = 'NNE'
 *   var point = Dms.compassPoint(24, 1); // point = 'N'
 */
Dms.compassPoint = function(bearing, precision) {
    if (precision === undefined) precision = 3;
    // note precision = max length of compass point; it could be extended to 4 for quarter-winds
    // (eg NEbN), but I think they are little used

    bearing = ((bearing%360)+360)%360; // normalise to 0..360

    var point;

    switch (precision) {
        case 1: // 4 compass points
            switch (Math.round(bearing*4/360)%4) {
                case 0: point = 'N'; break;
                case 1: point = 'E'; break;
                case 2: point = 'S'; break;
                case 3: point = 'W'; break;
            }
            break;
        case 2: // 8 compass points
            switch (Math.round(bearing*8/360)%8) {
                case 0: point = 'N';  break;
                case 1: point = 'NE'; break;
                case 2: point = 'E';  break;
                case 3: point = 'SE'; break;
                case 4: point = 'S';  break;
                case 5: point = 'SW'; break;
                case 6: point = 'W';  break;
                case 7: point = 'NW'; break;
            }
            break;
        case 3: // 16 compass points
            switch (Math.round(bearing*16/360)%16) {
                case  0: point = 'N';   break;
                case  1: point = 'NNE'; break;
                case  2: point = 'NE';  break;
                case  3: point = 'ENE'; break;
```

```
                case  4: point = 'E';   break;
                case  5: point = 'ESE'; break;
                case  6: point = 'SE';  break;
                case  7: point = 'SSE'; break;
                case  8: point = 'S';   break;
                case  9: point = 'SSW'; break;
                case 10: point = 'SW';  break;
                case 11: point = 'WSW'; break;
                case 12: point = 'W';   break;
                case 13: point = 'WNW'; break;
                case 14: point = 'NW';  break;
                case 15: point = 'NNW'; break;
            }
            break;
        default:
            throw new RangeError('Precision must be between 1 and 3');
    }

    return point;
};


/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

/** Polyfill String.trim for old browsers
 *  (q.v. blog.stevenlevithan.com/archives/faster-trim-javascript) */
if (String.prototype.trim === undefined) {
    String.prototype.trim = function() {
        return String(this).replace(/^\s\s*/, '').replace(/\s\s*$/, '');
    };
}

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */
if (typeof module != 'undefined' && module.exports) module.exports = Dms; // ≡ export default Dms
```