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:

```
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

Historical aside: The height of techsmall 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 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).

nology 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 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:

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

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°!).

Baghdad to Osaka – not a constant bearing!

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

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
```

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:

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

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

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

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

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:

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

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

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

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

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

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

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

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)

```
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°	Intersection point: **50°54′27″N, 004°30′31″E**
Point 2:	49.0034 N,	2.5735 E	Brng 2:	32.44°	

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

$\theta_a = \text{acos}(\ (\ \sin\varphi_2 - \sin\varphi_1 \cdot \cos\delta_{12}\ ) / (\ \sin\delta_{12} \cdot \cos\varphi_1\ )\ )$
$\theta_b = \text{acos}(\ (\ \sin\varphi_1 - \sin\varphi_2 \cdot \cos\delta_{12}\ ) / (\ \sin\delta_{12} \cdot \cos\varphi_2\ )\ )$ initial / final bearings between points 1 & 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 $\sin \alpha_1 = 0$ and $\sin \alpha_2 = 0$: infinite solutions
if $\sin \alpha_1 \cdot \sin \alpha_2 < 0$: 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 δ13 = d13 / R;
var dXt = Math.asin(Math.sin(δ13)*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 formulas 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 δ13 = d13 / R;
var dAt = Math.acos(Math.cos(δ13)/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 formulas 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 formulas 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*, Δλ *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*, Δλ *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: $\delta = d/R$ (angular distance)

$\varphi_2 = \varphi_1 + \delta \cdot \cos\theta$

$\Delta\psi = \ln(\tan(\pi/4 + \varphi_2/2) / \tan(\pi/4 + \varphi_1/2))$ (‘projected’ latitude difference)

$q = \Delta\varphi/\Delta\psi$ (or $\cos\varphi$ for E-W line)

$\Delta\lambda = \delta \cdot \sin\theta / q$

$\lambda_2 = \lambda_1 + \Delta\lambda$

where φ *is latitude*, λ *is longitude*, Δλ *is taking shortest route (<180°),* ln *is natural log,* R *is the earth’s radius*

JavaScript:
(all angles in radians)

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

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 type="module">
        import LatLon from 'https://cdn.jsdelivr.net/gh/chrisveness/geodesy@2.0.0/latlon-spherical.min.js';
        document.addEventListener('DOMContentLoaded', function() {
            document.querySelector('#calc-dist').onclick = function() {
                calculateDistance();
            }
        });
        function calculateDistance() {
            const p1 = LatLon.parse(document.querySelector('#point1').value);
            const p2 = LatLon.parse(document.querySelector('#point2').value);
            const dist = parseFloat(p1.distanceTo(p2).toPrecision(4));
            document.querySelector('#result-distance').textContent = dist + ' metres';
        }
    </script>
</head>
<body>
<form>
    Point 1: <input type="text" name="point1" id="point1" placeholder="lat1,lon1">
    Point 2: <input type="text" name="point2" id="point2" placeholder="lat2,lon2">
    <button type="button" id="calc-dist">Calculate distance</button>
    <output id="result-distance"></output>
</form>
</body>
</html>
```

Convert between degrees-minutes-seconds & decimal degrees

	Latitude	Longitude		
d	51.47788°N	000.00147°W	1° ≈ 111 km (110.57 eq'l — 111.70 polar)	
dm	51°28.673′N	000°00.088′W	1′ ≈ 1.85 km (= 1 nm)	0.01° ≈ 1.11 km
dms	51°28′40.4″N	000°00′05.3″W	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 formulas.
- 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. Full documentation is available, as well as a test suite.

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">`), and use UTF-8 encoding when saving files).

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 formulas as spreadsheets, in a form which breaks down the all stages involved to illustrate the operation.

February 2019: I have refactored the library to use ES modules, as well as extending it in scope; see the GitHub README and CHANGELOG for details.

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.
- Vahan Aghajanyan has made a C++ 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 2002-2019  */
/*                                                                                   MIT Licence  */
/* www.movable-type.co.uk/scripts/latlong.html                                                    */
/* www.movable-type.co.uk/scripts/geodesy-library.html#latlon-spherical                           */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

import Dms from './dms.js';

const π = Math.PI;


/**
 * Library of geodesy functions for operations on a spherical earth model.
 *
 * Includes distances, bearings, destinations, etc, for both great circle paths and rhumb lines,
 * and other related functions.
 *
```

```
 * All calculations are done using simple spherical trigonometric formulae.
 *
 * @module latlon-spherical
 */

// note greek letters (e.g. φ, λ, θ) are used for angles in radians to distinguish from angles in
// degrees (e.g. lat, lon, brng)


/* LatLonSpherical - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */


/**
 * Latitude/longitude points on a spherical model earth, and methods for calculating distances,
 * bearings, destinations, etc on (orthodromic) great-circle paths and (loxodromic) rhumb lines.
 */
class LatLonSpherical {

    /**
     * Creates a latitude/longitude point on the earth's surface, using a spherical model earth.
     *
     * @param  {number} lat - Latitude (in degrees).
     * @param  {number} lon - Longitude (in degrees).
     * @throws {TypeError} Non-numeric lat/lon.
     *
     * @example
     *   import LatLon from '/js/geodesy/latlon-spherical.js';
     *   const p = new LatLon(52.205, 0.119);
     */
    constructor(lat, lon) {
        if (isNaN(lat)) throw new TypeError(`Invalid lat ‘${lat}’`);
        if (isNaN(lon)) throw new TypeError(`Invalid lon ‘${lon}’`);

        this._lat = Dms.wrap90(lat);
        this._lon = Dms.wrap180(lon);
    }


    /**
     * Latitude in degrees north from equator (including aliases lat, latitude): can be set as
     * numeric or hexagesimal (deg-min-sec); returned as numeric.
     */
    get lat()       { return this._lat; }
    get latitude()  { return this._lat; }
    set lat(lat) {
        this._lat = isNaN(lat) ? Dms.wrap90(Dms.parse(lat)) : Dms.wrap90(lat);
        if (isNaN(this._lat)) throw new TypeError(`Invalid lat ‘${lat}’`);
    }
    set latitude(lat) {
        this._lat = isNaN(lat) ? Dms.wrap90(Dms.parse(lat)) : Dms.wrap90(lat);
        if (isNaN(this._lat)) throw new TypeError(`Invalid latitude ‘${lat}’`);
    }

    /**
     * Longitude in degrees east from international reference meridian (including aliases lon, lng,
     * longitude): can be set as numeric or hexagesimal (deg-min-sec); returned as numeric.
     */
    get lon()       { return this._lon; }
    get lng()       { return this._lon; }
    get longitude() { return this._lon; }
    set lon(lon) {
        this._lon = isNaN(lon) ? Dms.wrap180(Dms.parse(lon)) : Dms.wrap180(lon);
        if (isNaN(this._lon)) throw new TypeError(`Invalid lon ‘${lon}’`);
    }
    set lng(lon) {
        this._lon = isNaN(lon) ? Dms.wrap180(Dms.parse(lon)) : Dms.wrap180(lon);
        if (isNaN(this._lon)) throw new TypeError(`Invalid lng ‘${lon}’`);
    }
    set longitude(lon) {
        this._lon = isNaN(lon) ? Dms.wrap180(Dms.parse(lon)) : Dms.wrap180(lon);
        if (isNaN(this._lon)) throw new TypeError(`Invalid longitude ‘${lon}’`);
    }


    /** Conversion factors; 1000 * LatLon.metresToKm gives 1. */
    static get metresToKm()            { return 1/1000; }
    /** Conversion factors; 1000 * LatLon.metresToMiles gives 0.621371192237334. */
    static get metresToMiles()         { return 1/1609.344; }
    /** Conversion factors; 1000 * LatLon.metresToMiles gives 0.5399568034557236. */
    static get metresToNauticalMiles() { return 1/1852; }


    /**
     * Parses a latitude/longitude point from a variety of formats.
     *
```

```
     * Latitude & longitude (in degrees) can be supplied as two separate parameters, as a single
     * comma-separated lat/lon string, or as a single object with { lat, lon } or GeoJSON properties.
     *
     * The latitude/longitude values may be numeric or strings; they may be signed decimal or
     * deg-min-sec (hexagesimal) suffixed by compass direction (NSEW); a variety of separators are
     * accepted. Examples -3.62, '3 37 12W', '3°37′12″W'.
     *
     * Thousands/decimal separators must be comma/dot; use Dms.fromLocale to convert locale-specific
     * thousands/decimal separators.
     *
     * @param   {number|string|Object} lat|latlon - Latitude (in degrees) or comma-separated lat/lon or lat/lon object.
     * @param   {number|string}        [lon]      - Longitude (in degrees).
     * @returns {LatLon} Latitude/longitude point.
     * @throws  {TypeError} Invalid coordinate.
     *
     * @example
     *   const p1 = LatLon.parse(52.205, 0.119);                                    // numeric pair (≡ new LatLon)
     *   const p2 = LatLon.parse('52.205', '0.119');                                // numeric string pair (≡ new LatLon)
     *   const p3 = LatLon.parse('52.205, 0.119');                                  // single string numerics
     *   const p4 = LatLon.parse('52°12′18.0″N', '000°07′08.4″E');                  // DMS pair
     *   const p5 = LatLon.parse('52°12′18.0″N, 000°07′08.4″E');                    // single string DMS
     *   const p6 = LatLon.parse({ lat: 52.205, lon: 0.119 });                      // { lat, lon } object numeric
     *   const p7 = LatLon.parse({ lat: '52°12′18.0″N', lng: '000°07′08.4″E' });    // { lat, lng } object DMS
     *   const p8 = LatLon.parse({ type: 'Point', coordinates: [ 0.119, 52.205] }); // GeoJSON
     */
    static parse(...args) {
        if (args.length == 0) throw new TypeError('Invalid (empty) coordinate');
        if (args[0]===null || args[1]===null) throw new TypeError('Invalid (null) coordinate');

        let lat=undefined, lon=undefined;

        if (args.length == 2) { // regular (lat, lon) arguments
            [ lat, lon ] = args;
            lat = Dms.wrap90(Dms.parse(lat));
            lon = Dms.wrap180(Dms.parse(lon));
            if (isNaN(lat) || isNaN(lon)) throw new TypeError(`Invalid coordinate '${args.toString()}'`);
        }

        if (args.length == 1 && typeof args[0] == 'string') { // single comma-separated lat,lon string
            [ lat, lon ] = args[0].split(',');
            lat = Dms.wrap90(Dms.parse(lat));
            lon = Dms.wrap180(Dms.parse(lon));
            if (isNaN(lat) || isNaN(lon)) throw new TypeError(`Invalid coordinate '${args[0]}'`);
        }

        if (args.length == 1 && typeof args[0] == 'object') { // single { lat, lon } object
            const ll = args[0];
            if (ll.type == 'Point' && Array.isArray(ll.coordinates)) { // GeoJSON
                [ lon, lat ] = ll.coordinates;
            } else { // regular { lat, lon } object
                if (ll.latitude  != undefined) lat = ll.latitude;
                if (ll.lat       != undefined) lat = ll.lat;
                if (ll.longitude != undefined) lon = ll.longitude;
                if (ll.lng       != undefined) lon = ll.lng;
                if (ll.lon       != undefined) lon = ll.lon;
                lat = Dms.wrap90(Dms.parse(lat));
                lon = Dms.wrap180(Dms.parse(lon));
            }
            if (isNaN(lat) || isNaN(lon)) throw new TypeError(`Invalid coordinate '${JSON.stringify(args[0])}'`);
        }

        if (isNaN(lat) || isNaN(lon)) throw new TypeError(`Invalid coordinate '${args.toString()}'`);

        return new LatLonSpherical(lat, lon);
    }


    /**
     * Returns the distance along the surface of the earth from ‘this’ point to destination point.
     *
     * Uses haversine formula: a = sin²(Δφ/2) + cosφ1·cosφ2 · sin²(Δλ/2); d = 2 · atan2(√a, √(a-1)).
     *
     * @param   {LatLon} point - Latitude/longitude of destination point.
     * @param   {number} [radius=6371e3] - Radius of earth (defaults to mean radius in metres).
     * @returns {number} Distance between this point and destination point, in same units as radius.
     * @throws  {TypeError} Radius is not a number.
     *
     * @example
     *   const p1 = new LatLon(52.205, 0.119);
     *   const p2 = new LatLon(48.857, 2.351);
     *   const d = p1.distanceTo(p2);       // 404.3×10³ m
     *   const m = p1.distanceTo(p2, 3959); // 251.2 miles
     */
    distanceTo(point, radius=6371e3) {
        if (!(point instanceof LatLonSpherical)) point = LatLonSpherical.parse(point); // allow literal forms
```

```
    if (isNaN(radius)) throw new TypeError('Radius is not a number');

    // a = sin²(Δφ/2) + cos(φ1)⋅cos(φ2)⋅sin²(Δλ/2)
    // δ = 2·atan2(√(a), √(1−a))
    // see mathforum.org/library/drmath/view/51879.html for derivation

    const R = radius;
    const φ1 = this.lat.toRadians(),  λ1 = this.lon.toRadians();
    const φ2 = point.lat.toRadians(), λ2 = point.lon.toRadians();
    const Δφ = φ2 - φ1;
    const Δλ = λ2 - λ1;

    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;

    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 (0°..360°).
 *
 * @example
 *   const p1 = new LatLon(52.205, 0.119);
 *   const p2 = new LatLon(48.857, 2.351);
 *   const b1 = p1.initialBearingTo(p2); // 156.2°
 */
initialBearingTo(point) {
    if (!(point instanceof LatLonSpherical)) point = LatLonSpherical.parse(point); // allow literal forms

    // tanθ = sinΔλ⋅cosφ2 / cosφ1⋅sinφ2 − sinφ1⋅cosφ2⋅cosΔλ
    // see mathforum.org/library/drmath/view/55417.html for derivation

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

    const x = Math.cos(φ1) * Math.sin(φ2) - Math.sin(φ1) * Math.cos(φ2) * Math.cos(Δλ);
    const y = Math.sin(Δλ) * Math.cos(φ2);
    const θ = Math.atan2(y, x);

    const bearing = θ.toDegrees();

    return Dms.wrap360(bearing);
}


/**
 * Returns final bearing arriving at 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 (0°..360°).
 *
 * @example
 *   const p1 = new LatLon(52.205, 0.119);
 *   const p2 = new LatLon(48.857, 2.351);
 *   const b2 = p1.finalBearingTo(p2); // 157.9°
 */
finalBearingTo(point) {
    if (!(point instanceof LatLonSpherical)) point = LatLonSpherical.parse(point); // allow literal forms

    // get initial bearing from destination point to this point & reverse it by adding 180°

    const bearing = point.initialBearingTo(this) + 180;

    return Dms.wrap360(bearing);
}


/**
 * Returns the midpoint between ‘this’ point and destination point.
 *
 * @param   {LatLon} point - Latitude/longitude of destination point.
 * @returns {LatLon} Midpoint between this point and destination point.
 *
 * @example
 *   const p1 = new LatLon(52.205, 0.119);
 *   const p2 = new LatLon(48.857, 2.351);
 *   const pMid = p1.midpointTo(p2); // 50.5363°N, 001.2746°E
 */
midpointTo(point) {
```

```
        if (!(point instanceof LatLonSpherical)) point = LatLonSpherical.parse(point); // allow literal forms

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

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

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

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

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

        const lat = φ3.toDegrees();
        const lon = λ3.toDegrees();

        return new LatLonSpherical(lat, Dms.wrap180(lon));
    }


    /**
     * Returns the point at given fraction between ‘this’ point and given 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
     *   const p1 = new LatLon(52.205, 0.119);
     *   const p2 = new LatLon(48.857, 2.351);
     *   const pInt = p1.intermediatePointTo(p2, 0.25); // 51.3721°N, 000.7073°E
     */
    intermediatePointTo(point, fraction) {
        if (!(point instanceof LatLonSpherical)) point = LatLonSpherical.parse(point); // allow literal forms

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

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

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

        const x = A * Math.cos(φ1) * Math.cos(λ1) + B * Math.cos(φ2) * Math.cos(λ2);
        const y = A * Math.cos(φ1) * Math.sin(λ1) + B * Math.cos(φ2) * Math.sin(λ2);
        const z = A * Math.sin(φ1) + B * Math.sin(φ2);

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

        const lat = φ3.toDegrees();
        const lon = λ3.toDegrees();

        return new LatLonSpherical(lat, Dms.wrap180(lon));
    }


    /**
     * 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
     *   const p1 = new LatLon(51.47788, -0.00147);
     *   const p2 = p1.destinationPoint(7794, 300.7); // 51.5136°N, 000.0983°W
     */
    destinationPoint(distance, bearing, radius=6371e3) {
        // sinφ2 = sinφ1⋅cosδ + cosφ1⋅sinδ⋅cosθ
        // tanΔλ = sinθ⋅sinδ⋅cosφ1 / cosδ−sinφ1⋅sinφ2
        // see mathforum.org/library/drmath/view/52049.html for derivation
```

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

    const φ1 = this.lat.toRadians(), λ1 = this.lon.toRadians();

    const sinφ2 = Math.sin(φ1) * Math.cos(δ) + Math.cos(φ1) * Math.sin(δ) * Math.cos(θ);
    const φ2 = Math.asin(sinφ2);
    const y = Math.sin(θ) * Math.sin(δ) * Math.cos(φ1);
    const x = Math.cos(δ) - Math.sin(φ1) * sinφ2;
    const λ2 = λ1 + Math.atan2(y, x);

    const lat = φ2.toDegrees();
    const lon = λ2.toDegrees();

    return new LatLonSpherical(lat, Dms.wrap180(lon));
}


/**
 * 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
 *   const p1 = new LatLon(51.8853, 0.2545), brng1 = 108.547;
 *   const p2 = new LatLon(49.0034, 2.5735), brng2 =  32.435;
 *   const pInt = LatLon.intersection(p1, brng1, p2, brng2); // 50.9078°N, 004.5084°E
 */
static intersection(p1, brng1, p2, brng2) {
    if (!(p1 instanceof LatLonSpherical)) p1 = LatLonSpherical.parse(p1); // allow literal forms
    if (!(p2 instanceof LatLonSpherical)) p2 = LatLonSpherical.parse(p2); // allow literal forms

    // see www.edwilliams.org/avform.htm#Intersection

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

    // angular distance p1-p2
    const 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
    const cosθa = (Math.sin(φ2) - Math.sin(φ1)*Math.cos(δ12)) / (Math.sin(δ12)*Math.cos(φ1));
    const cosθb = (Math.sin(φ1) - Math.sin(φ2)*Math.cos(δ12)) / (Math.sin(δ12)*Math.cos(φ2));
    const θa = Math.acos(Math.min(Math.max(cosθa, -1), 1)); // protect against rounding errors
    const θb = Math.acos(Math.min(Math.max(cosθb, -1), 1)); // protect against rounding errors

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

    const α1 = θ13 - θ12; // angle 2-1-3
    const α2 = θ21 - θ23; // 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

    const cosα3 = -Math.cos(α1)*Math.cos(α2) + Math.sin(α1)*Math.sin(α2)*Math.cos(δ12);

    const δ13 = Math.atan2(Math.sin(δ12)*Math.sin(α1)*Math.sin(α2), Math.cos(α2) + Math.cos(α1)*cosα3);

    const φ3 = Math.asin(Math.sin(φ1)*Math.cos(δ13) + Math.cos(φ1)*Math.sin(δ13)*Math.cos(θ13));

    const Δλ13 = Math.atan2(Math.sin(θ13)*Math.sin(δ13)*Math.cos(φ1), Math.cos(δ13) - Math.sin(φ1)*Math.sin(φ3));
    const λ3 = λ1 + Δλ13;

    const lat = φ3.toDegrees();
    const lon = λ3.toDegrees();

    return new LatLonSpherical(lat, Dms.wrap180(lon));
}


/**
 * 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
     *   const pCurrent = new LatLon(53.2611, -0.7972);
     *   const p1 = new LatLon(53.3206, -1.7297);
     *   const p2 = new LatLon(53.1887, 0.1334);
     *   const d = pCurrent.crossTrackDistanceTo(p1, p2);  // -307.5 m
     */
    crossTrackDistanceTo(pathStart, pathEnd, radius=6371e3) {
        if (!(pathStart instanceof LatLonSpherical)) pathStart = LatLonSpherical.parse(pathStart); // allow literal forms
        if (!(pathEnd instanceof LatLonSpherical)) pathEnd = LatLonSpherical.parse(pathEnd);       // allow literal forms
        const R = radius;

        const δ13 = pathStart.distanceTo(this, R) / R;
        const θ13 = pathStart.initialBearingTo(this).toRadians();
        const θ12 = pathStart.initialBearingTo(pathEnd).toRadians();

        const δxt = Math.asin(Math.sin(δ13) * Math.sin(θ13 - θ12));

        return δxt * R;
    }


    /**
     * Returns how far ‘this’ point is along a path from from start-point, heading towards end-point.
     * That is, if a perpendicular is drawn from ‘this’ point to the (great circle) path, the
     * along-track distance is the distance from the start point to where the perpendicular crosses
     * the path.
     *
     * @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 along great circle to point nearest ‘this’ point.
     *
     * @example
     *   const pCurrent = new LatLon(53.2611, -0.7972);
     *   const p1 = new LatLon(53.3206, -1.7297);
     *   const p2 = new LatLon(53.1887,  0.1334);
     *   const d = pCurrent.alongTrackDistanceTo(p1, p2);  // 62.331 km
     */
    alongTrackDistanceTo(pathStart, pathEnd, radius=6371e3) {
        if (!(pathStart instanceof LatLonSpherical)) pathStart = LatLonSpherical.parse(pathStart); // allow literal forms
        if (!(pathEnd instanceof LatLonSpherical)) pathEnd = LatLonSpherical.parse(pathEnd);       // allow literal forms
        const R = radius;

        const δ13 = pathStart.distanceTo(this, R) / R;
        const θ13 = pathStart.initialBearingTo(this).toRadians();
        const θ12 = pathStart.initialBearingTo(pathEnd).toRadians();

        const δxt = Math.asin(Math.sin(δ13) * Math.sin(θ13-θ12));

        const δat = Math.acos(Math.cos(δ13) / Math.abs(Math.cos(δxt)));

        return δat*Math.sign(Math.cos(θ12-θ13)) * R;
    }


    /**
     * 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.
     * @returns {number} Maximum latitude reached.
     */
    maxLatitude(bearing) {
        const θ = Number(bearing).toRadians();

        const φ = this.lat.toRadians();

        const 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.
     */
    static crossingParallels(point1, point2, latitude) {
        const φ = Number(latitude).toRadians();

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

        const Δλ = λ2 - λ1;

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

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

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

        const lon1 = λi1.toDegrees();
        const lon2 = λi2.toDegrees();

        return {
            lon1: Dms.wrap180(lon1),
            lon2: Dms.wrap180(lon2),
        };
    }


    /* 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
     *   const p1 = new LatLon(51.127, 1.338);
     *   const p2 = new LatLon(50.964, 1.853);
     *   const d = p1.distanceTo(p2); //  40.31 km
     */
    rhumbDistanceTo(point, radius=6371e3) {
        // see www.edwilliams.org/avform.htm#Rhumb

        if (!(point instanceof LatLonSpherical)) point = LatLonSpherical.parse(point); // allow literal forms

        const R = radius;
        const φ1 = this.lat.toRadians(), φ2 = point.lat.toRadians();
        const Δφ = φ2 - φ1;
        let Δλ = Math.abs(point.lon - this.lon).toRadians();
        // if dLon over 180° take shorter rhumb line across the anti-meridian:
        if (Math.abs(Δλ) > π) Δλ = Δλ > 0 ? -(2 * π - Δλ) : (2 * π + 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
        const Δψ = Math.log(Math.tan(φ2 / 2 + π / 4) / Math.tan(φ1 / 2 + π / 4));
        const q = Math.abs(Δψ) > 10e-12 ? Δφ / Δψ : Math.cos(φ1);

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

        return d;
    }


    /**
     * 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.
     * @throws  {TypeError} Invalid coordinate.
     *
     * @example
```

```
 *   const p1 = new LatLon(51.127, 1.338);
 *   const p2 = new LatLon(50.964, 1.853);
 *   const d = p1.rhumbBearingTo(p2); // 116.7°
 */
rhumbBearingTo(point) {
    if (!(point instanceof LatLonSpherical)) point = LatLonSpherical.parse(point); // allow literal forms

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

    const Δψ = Math.log(Math.tan(φ2 / 2 + π / 4) / Math.tan(φ1 / 2 + π / 4));

    const θ = Math.atan2(Δλ, Δψ);

    const bearing = θ.toDegrees();

    return Dms.wrap360(bearing);
}


/**
 * 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
 *   const p1 = new LatLon(51.127, 1.338);
 *   const p2 = p1.rhumbDestinationPoint(40300, 116.7); // 50.9642°N, 001.8530°E
 */
rhumbDestinationPoint(distance, bearing, radius=6371e3) {
    const φ1 = this.lat.toRadians(), λ1 = this.lon.toRadians();
    const θ = Number(bearing).toRadians();

    const δ = distance / radius; // angular distance in radians

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

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

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

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

    const lat = φ2.toDegrees();
    const lon = λ2.toDegrees();

    return new LatLonSpherical(lat, Dms.wrap180(lon));
}


/**
 * 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
 *   const p1 = new LatLon(51.127, 1.338);
 *   const p2 = new LatLon(50.964, 1.853);
 *   const pMid = p1.rhumbMidpointTo(p2); // 51.0455°N, 001.5957°E
 */
rhumbMidpointTo(point) {
    if (!(point instanceof LatLonSpherical)) point = LatLonSpherical.parse(point); // allow literal forms

    // see mathforum.org/kb/message.jspa?messageID=148837

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

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

    const φ3 = (φ1 + φ2) / 2;
    const f1 = Math.tan(π / 4 + φ1 / 2);
    const f2 = Math.tan(π / 4 + φ2 / 2);
    const f3 = Math.tan(π / 4 + φ3 / 2);
```

```
        let 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

        const lat = φ3.toDegrees();
        const lon = λ3.toDegrees();

        return new LatLonSpherical(lat, Dms.wrap180(lon));
    }


    /* 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
     *   const polygon = [new LatLon(0,0), new LatLon(1,0), new LatLon(0,1)];
     *   const area = LatLon.areaOf(polygon); // 6.18e9 m²
     */
    static areaOf(polygon, radius=6371e3) {
        // 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

        const R = radius;

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

        const nVertices = polygon.length - 1;

        let S = 0; // spherical excess in steradians
        for (let v=0; v<nVertices; v++) {
            const φ1 = polygon[v].lat.toRadians();
            const φ2 = polygon[v+1].lat.toRadians();
            const Δλ = (polygon[v+1].lon - polygon[v].lon).toRadians();
            const 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*π;

        const 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(p) {
            // TODO: any better test than this?
            let ΣΔ = 0;
            let prevBrng = p[0].initialBearingTo(p[1]);
            for (let v=0; v<p.length-1; v++) {
                const initBrng = p[v].initialBearingTo(p[v+1]);
                const finalBrng = p[v].finalBearingTo(p[v+1]);
                ΣΔ += (initBrng - prevBrng + 540) % 360 - 180;
                ΣΔ += (finalBrng - initBrng + 540) % 360 - 180;
                prevBrng = finalBrng;
            }
            const initBrng = p[0].initialBearingTo(p[1]);
            ΣΔ += (initBrng - prevBrng + 540) % 360 - 180;
            // TODO: fix (intermittant) edge crossing pole - eg (85,90), (85,0), (85,-90)
            const 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 have identical latitude and longitude values.
     *
     * @example
     *   const p1 = new LatLon(52.205, 0.119);
     *   const p2 = new LatLon(52.205, 0.119);
     *   const equal = p1.equals(p2); // true
     */
    equals(point) {
        if (!(point instanceof LatLonSpherical)) point = LatLonSpherical.parse(point); // allow literal forms

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

        return true;
    }


    /**
     * Converts ‘this’ point to a GeoJSON object.
     *
     * @returns {Object} this point as a GeoJSON ‘Point’ object.
     */
    toGeoJSON() {
        return { type: 'Point', coordinates: [ this.lon, this.lat ] };
    }


    /**
     * Returns a string representation of ‘this’ point, formatted as degrees, degrees+minutes, or
     * degrees+minutes+seconds.
     *
     * @param   {string} [format=d] - Format point as 'd', 'dm', 'dms', or 'n' for signed numeric.
     * @param   {number} [dp=4|2|0] - Number of decimal places to use: default 4 for d, 2 for dm, 0 for dms.
     * @returns {string} Comma-separated formatted latitude/longitude.
     * @throws  {RangeError} Invalid format.
     *
     * @example
     *   const greenwich = new LatLon(51.47788, -0.00147);
     *   const d = greenwich.toString();                        // 51.4779°N, 000.0015°W
     *   const dms = greenwich.toString('dms', 2);              // 51°28′40.37″N, 000°00′05.29″W
     *   const [lat, lon] = greenwich.toString('n').split(','); // 51.4779, -0.0015
     */
    toString(format='d', dp=undefined) {
        // note: explicitly set dp to undefined for passing through to toLat/toLon
        if (![ 'd', 'dm', 'dms', 'n' ].includes(format)) throw new RangeError(`Invalid format ‘${format}’`);

        if (format == 'n') { // signed numeric degrees
            if (dp == undefined) dp = 4;
            return `${this.lat.toFixed(dp)},${this.lon.toFixed(dp)}`;
        }
        const lat = Dms.toLat(this.lat, format, dp);
        const lon = Dms.toLon(this.lon, format, dp);
        return `${lat}, ${lon}`;
    }

}


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

export { LatLonSpherical as default, Dms };
```

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */
/* Geodesy representation conversion functions                        (c) Chris Veness 2002-2019  */
/*                                                                                   MIT Licence  */
/* www.movable-type.co.uk/scripts/latlong.html                                                    */
/* www.movable-type.co.uk/scripts/js/geodesy/geodesy-library.html#dms                             */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */

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


/**
 * Latitude/longitude points may be represented as decimal degrees, or subdivided into sexagesimal
 * minutes and seconds. This module provides methods for parsing and representing degrees / minutes
 * / seconds.
 *
 * @module dms
 */


/* Degree-minutes-seconds (& cardinal directions) separator character */
let dmsSeparator = '\u202f'; // U+202F = 'narrow no-break space'
```

```
/**
 * Functions for parsing and representing degrees / minutes / seconds.
 */
class Dms {

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

    /**
     * Separator character to be used to separate degrees, minutes, seconds, and cardinal directions.
     *
     * Default separator is U+202F ‘narrow no-break space’.
     *
     * To change this (e.g. to empty string or full space), set Dms.separator prior to invoking
     * formatting.
     *
     * @example
     *   import LatLon, { Dms } from '/js/geodesy/latlon-spherical.js';
     *   const p = new LatLon(51.2, 0.33).toString('dms');  // 51° 12′ 00″ N, 000° 19′ 48″ E
     *   Dms.separator = '';                                // no separator
     *   const p′ = new LatLon(51.2, 0.33).toString('dms'); // 51°12′00″N, 000°19′48″E
     */
    static get separator()     { return dmsSeparator; }
    static set separator(char) { dmsSeparator = char; }


    /**
     * 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. Examples -3.62,
     * '3 37 12W', '3°37′12″W'.
     *
     * Thousands/decimal separators must be comma/dot; use Dms.fromLocale to convert locale-specific
     * thousands/decimal separators.
     *
     * @param   {string|number} dms - Degrees or deg/min/sec in variety of formats.
     * @returns {number}        Degrees as decimal number.
     *
     * @example
     *   const lat = Dms.parse('51° 28′ 40.37″ N');
     *   const lon = Dms.parse('000° 00′ 05.29″ W');
     *   const p1 = new LatLon(lat, lon); // 51.4779°N, 000.0015°W
     */
    static parse(dms) {
        // check for signed decimal degrees without NSEW, if so return it directly
        if (!isNaN(parseFloat(dms)) && isFinite(dms)) return Number(dms);

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

        if (dmsParts == '') return NaN;

        // and convert to decimal degrees...
        let deg = null;
        switch (dmsParts.length) {
            case 3:  // interpret 3-part result as d/m/s
                deg = dmsParts[0]/1 + dmsParts[1]/60 + dmsParts[2]/3600;
                break;
            case 2:  // interpret 2-part result as d/m
                deg = dmsParts[0]/1 + dmsParts[1]/60;
                break;
            case 1:  // just d (possibly decimal) or non-separated dddmmss
                deg = dmsParts[0];
                // check for fixed-width unseparated format eg 0033709W
                //if (/[NS]/i.test(dmsParts)) 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(dms.trim())) deg = -deg; // take '-', west and south as -ve

        return Number(deg);
    }


    /**
     * 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.
     *  - degrees are zero-padded to 3 digits; for degrees latitude, use .slice(1) to remove leading
     *    zero.
```

```
     *
     * @private
     * @param   {number} deg - Degrees to be formatted as specified.
     * @param   {string} [format=d] - Return value as 'd', 'dm', 'dms' for deg, deg+min, deg+min+sec.
     * @param   {number} [dp=4|2|0] - Number of decimal places to use – default 4 for d, 2 for dm, 0 for dms.
     * @returns {string} Degrees formatted as deg/min/secs according to specified format.
     */
    static toDms(deg, format='d', dp=undefined) {
        if (isNaN(deg)) return null;  // give up here if we can't make a number from deg
        if (typeof deg == 'string' && deg.trim() == '') return null;
        if (typeof deg == 'boolean') return null;
        if (deg == Infinity) return null;
        if (deg == null) return null;

        // default values
        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 = 'd'; dp = 4; break; // be forgiving on invalid format
            }
        }

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

        let dms = null, d = null, m = null, s = null;
        switch (format) {
            default: // invalid format spec!
            case 'd': case 'deg':
                d = deg.toFixed(dp);                     // round/right-pad degrees
                if (d<100) d = '0' + d;                  // left-pad with leading zeros (note may include decimals)
                if (d<10) d = '0' + d;
                dms = d + '°';
                break;
            case 'dm': case 'deg+min':
                d = Math.floor(deg);                     // get component deg
                m = ((deg*60) % 60).toFixed(dp);         // get component min & round/right-pad
                if (m == 60) { m = (0).toFixed(dp); d++; } // check for rounding up
                d = ('000'+d).slice(-3);                 // left-pad with leading zeros
                if (m<10) m = '0' + m;                   // left-pad with leading zeros (note may include decimals)
                dms = d + '°'+Dms.separator + m + '′';
                break;
            case 'dms': case 'deg+min+sec':
                d = Math.floor(deg);                     // get component deg
                m = Math.floor((deg*3600)/60) % 60;      // get component min
                s = (deg*3600 % 60).toFixed(dp);         // get component sec & round/right-pad
                if (s == 60) { s = (0).toFixed(dp); m++; } // check for rounding up
                if (m == 60) { m = 0; d++; }             // check for rounding up
                d = ('000'+d).slice(-3);                 // left-pad with leading zeros
                m = ('00'+m).slice(-2);                  // left-pad with leading zeros
                if (s<10) s = '0' + s;                   // left-pad with leading zeros (note may include decimals)
                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=d] - Return value as 'd', 'dm', 'dms' for deg, deg+min, deg+min+sec.
     * @param   {number} [dp=4|2|0] - Number of decimal places to use – default 4 for d, 2 for dm, 0 for dms.
     * @returns {string} Degrees formatted as deg/min/secs according to specified format.
     *
     * @example
     *   const lat = Dms.toLat(-3.62, 'dms'); // 3°37′12″S
     */
    static toLat(deg, format, dp) {
        const lat = Dms.toDms(Dms.wrap90(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=d] - Return value as 'd', 'dm', 'dms' for deg, deg+min, deg+min+sec.
     * @param   {number} [dp=4|2|0] - Number of decimal places to use – default 4 for d, 2 for dm, 0 for dms.
     * @returns {string} Degrees formatted as deg/min/secs according to specified format.
     *
```

```
     * @example
     *   const lon = Dms.toLon(-3.62, 'dms'); // 3°37′12″W
     */
    static toLon(deg, format, dp) {
        const lon = Dms.toDms(Dms.wrap180(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=d] - Return value as 'd', 'dm', 'dms' for deg, deg+min, deg+min+sec.
     * @param   {number} [dp=4|2|0] - Number of decimal places to use – default 4 for d, 2 for dm, 0 for dms.
     * @returns {string} Degrees formatted as deg/min/secs according to specified format.
     *
     * @example
     *   const lon = Dms.toBrng(-3.62, 'dms'); // 356°22′48″
     */
    static toBrng(deg, format, dp) {
        const brng =  Dms.toDms(Dms.wrap360(deg), format, dp);
        return brng===null ? '–' : brng.replace('360', '0');  // just in case rounding took us up to 360°!
    }


    /**
     * Converts DMS string from locale thousands/decimal separators to JavaScript comma/dot separators
     * for subsequent parsing.
     *
     * Both thousands and decimal separators must be followed by a numeric character, to facilitate
     * parsing of single lat/long string (in which whitespace must be left after the comma separator).
     *
     * @param   {string} str - Degrees/minutes/seconds formatted with locale separators.
     * @returns {string} Degrees/minutes/seconds formatted with standard Javascript separators.
     *
     * @example
     *   const lat = Dms.fromLocale('51°28′40,12″N');                          // '51°28′40.12″N' in France
     *   const p = new LatLon(Dms.fromLocale('51°28′40,37″N, 000°00′05,29″W'); // '51.4779°N, 000.0015°W' in France
     */
    static fromLocale(str) {
        const locale = (123456.789).toLocaleString();
        const separator = { thousands: locale.slice(3, 4), decimal: locale.slice(7, 8) };
        return str.replace(separator.thousands, '⁜').replace(separator.decimal, '.').replace('⁜', ',');
    }


    /**
     * Converts DMS string from JavaScript comma/dot thousands/decimal separators to locale separators.
     *
     * Can also be used to format standard numbers such as distances.
     *
     * @param   {string} str - Degrees/minutes/seconds formatted with standard Javascript separators.
     * @returns {string} Degrees/minutes/seconds formatted with locale separators.
     *
     * @example
     *   const Dms.toLocale('123,456.789');                   // '123.456,789' in France
     *   const Dms.toLocale('51°28′40.12″N, 000°00′05.31″W'); // '51°28′40,12″N, 000°00′05,31″W' in France
     */
    static toLocale(str) {
        const locale = (123456.789).toLocaleString();
        const separator = { thousands: locale.slice(3, 4), decimal: locale.slice(7, 8) };
        return str.replace(/,([0-9])/, '⁜$1').replace('.', separator.decimal).replace('⁜', separator.thousands);
    }


    /**
     * 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
     *   const point = Dms.compassPoint(24);    // point = 'NNE'
     *   const point = Dms.compassPoint(24, 1); // point = 'N'
     */
    static compassPoint(bearing, precision=3) {
        if (![ 1,2,3 ].includes(Number(precision))) throw new RangeError('‘precision’ must be 1, 2 or 3'); // eslint-disable-line comma-spacing
        // note precision could be extended to 4 for quarter-winds (eg NbNW), but I think they are little used

        bearing = Dms.wrap360(bearing); // normalise to range 0..360°

        const cardinals = [
            'N', 'NNE', 'NE', 'ENE',
```

```
            'E', 'ESE', 'SE', 'SSE',
            'S', 'SSW', 'SW', 'WSW',
            'W', 'WNW', 'NW', 'NNW' ];
        const n = 4 * 2**(precision-1); // no of compass points at req'd precision (1=>4, 2=>8, 3=>16)
        const cardinal = cardinals[Math.round(bearing*n/360)%n * 16/n];

        return cardinal;
    }


    /**
     * Constrain degrees to range 0..360 (e.g. for bearings); -1 => 359, 361 => 1.
     *
     * @private
     * @param {number} degrees
     * @returns degrees within range 0..360.
     */
    static wrap360(degrees) {
        if (0<=degrees && degrees<360) return degrees; // avoid rounding due to arithmetic ops if within range
        return (degrees%360+360) % 360; // sawtooth wave p:360, a:360
    }

    /**
     * Constrain degrees to range -180..+180 (e.g. for longitude); -181 => 179, 181 => -179.
     *
     * @private
     * @param {number} degrees
     * @returns degrees within range -180..+180.
     */
    static wrap180(degrees) {
        if (-180<degrees && degrees<=180) return degrees; // avoid rounding due to arithmetic ops if within range
        return (degrees+540)%360-180; // sawtooth wave p:180, a:±180
    }

    /**
     * Constrain degrees to range -90..+90 (e.g. for latitude); -91 => -89, 91 => 89.
     *
     * @private
     * @param {number} degrees
     * @returns degrees within range -90..+90.
     */
    static wrap90(degrees) {
        if (-90<=degrees && degrees<=90) return degrees; // avoid rounding due to arithmetic ops if within range
        return Math.abs((degrees%360 + 270)%360 - 180) - 90; // triangle wave p:360 a:±90 TODO: fix e.g. -315°
    }

}


// Extend Number object with methods to convert between degrees & radians
Number.prototype.toRadians = function() { return this * Math.PI / 180; };
Number.prototype.toDegrees = function() { return this * 180 / Math.PI; };

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

export default Dms;
```