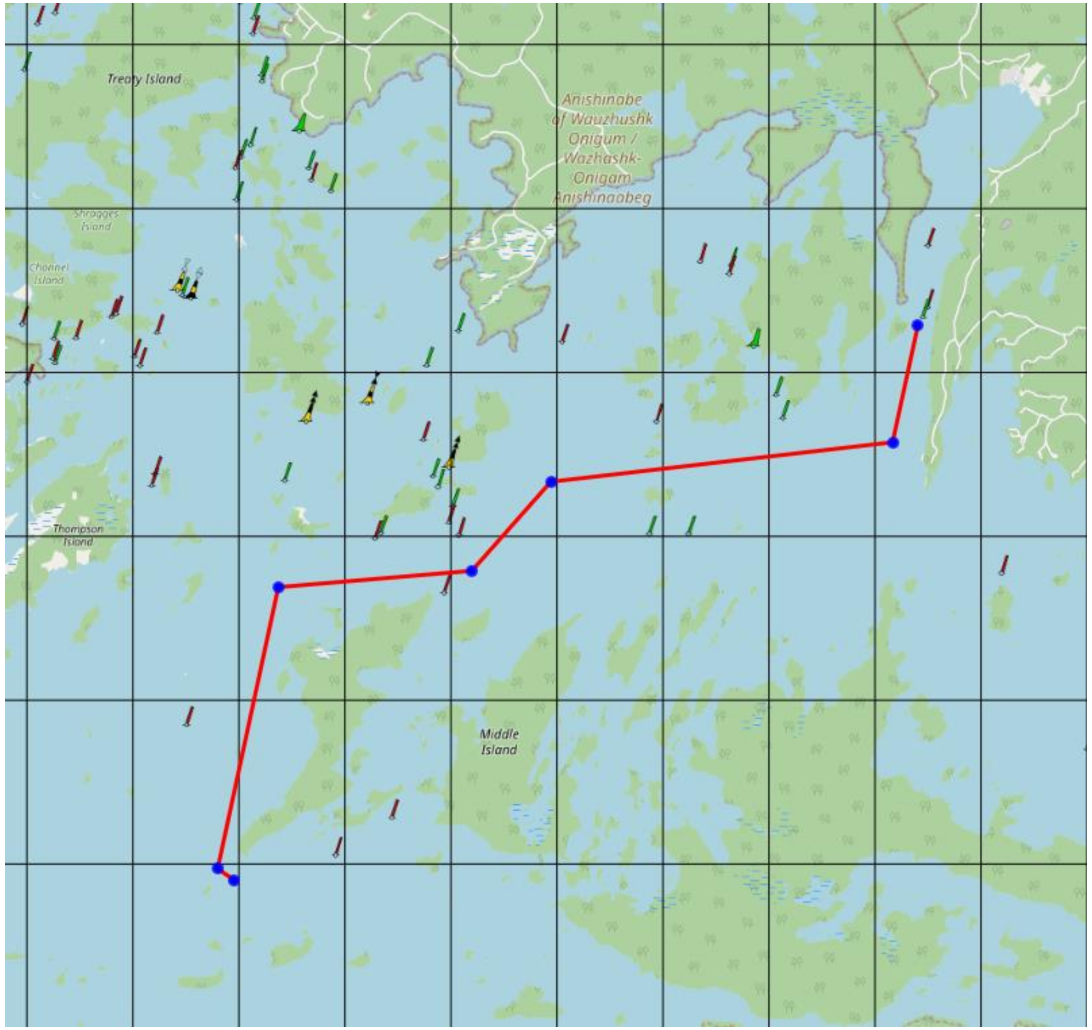
SAIL MAPPER

This feature would have been easier to implement if I was a flat earther

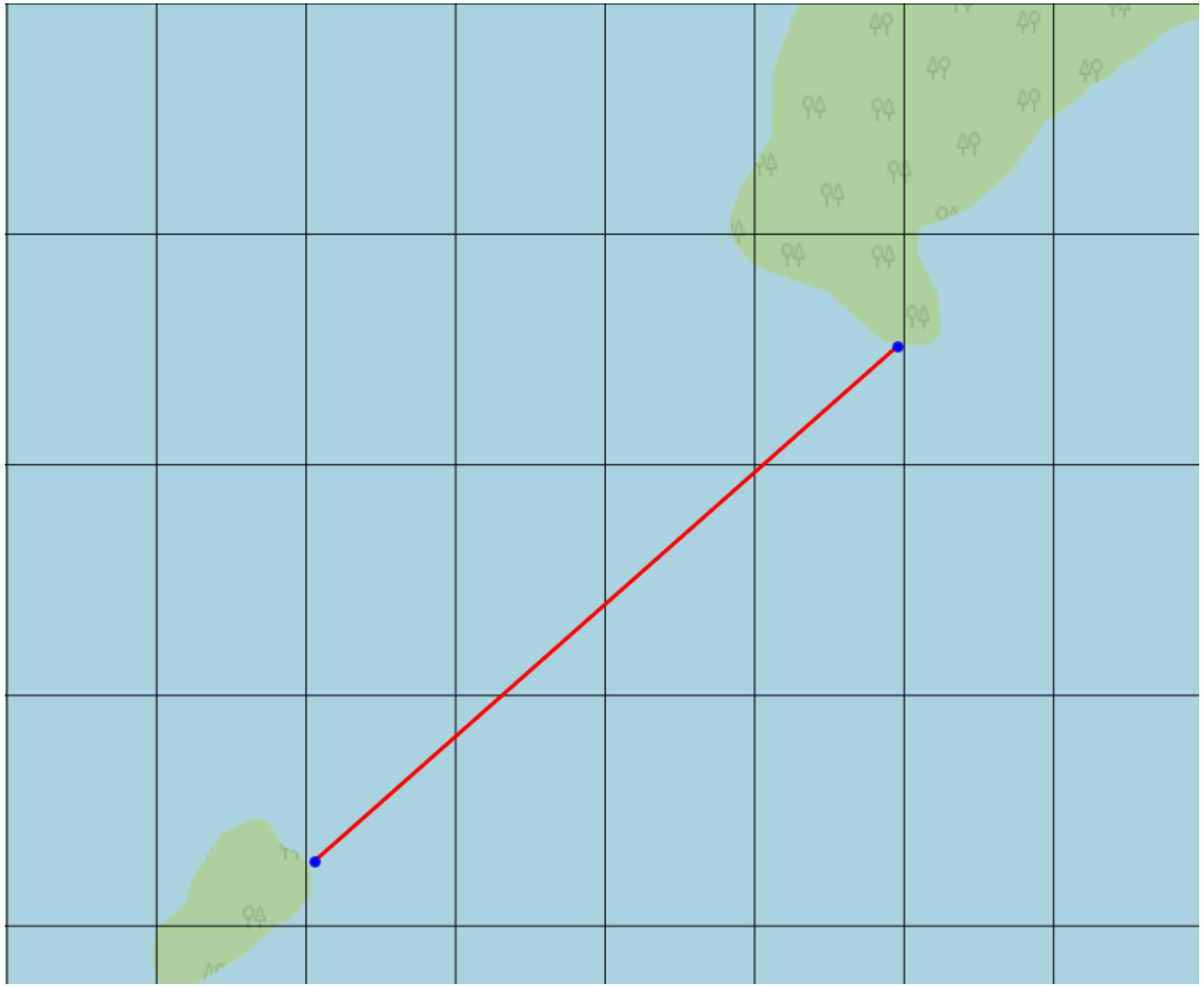
SAIL MAPPER

- Tracking and recording sailboat races



-Open Sea Map

Check path

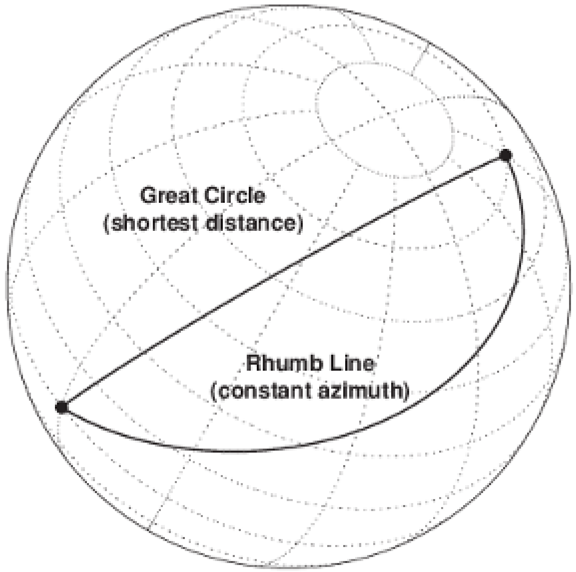


-Open Sea Map

The earth is round :(

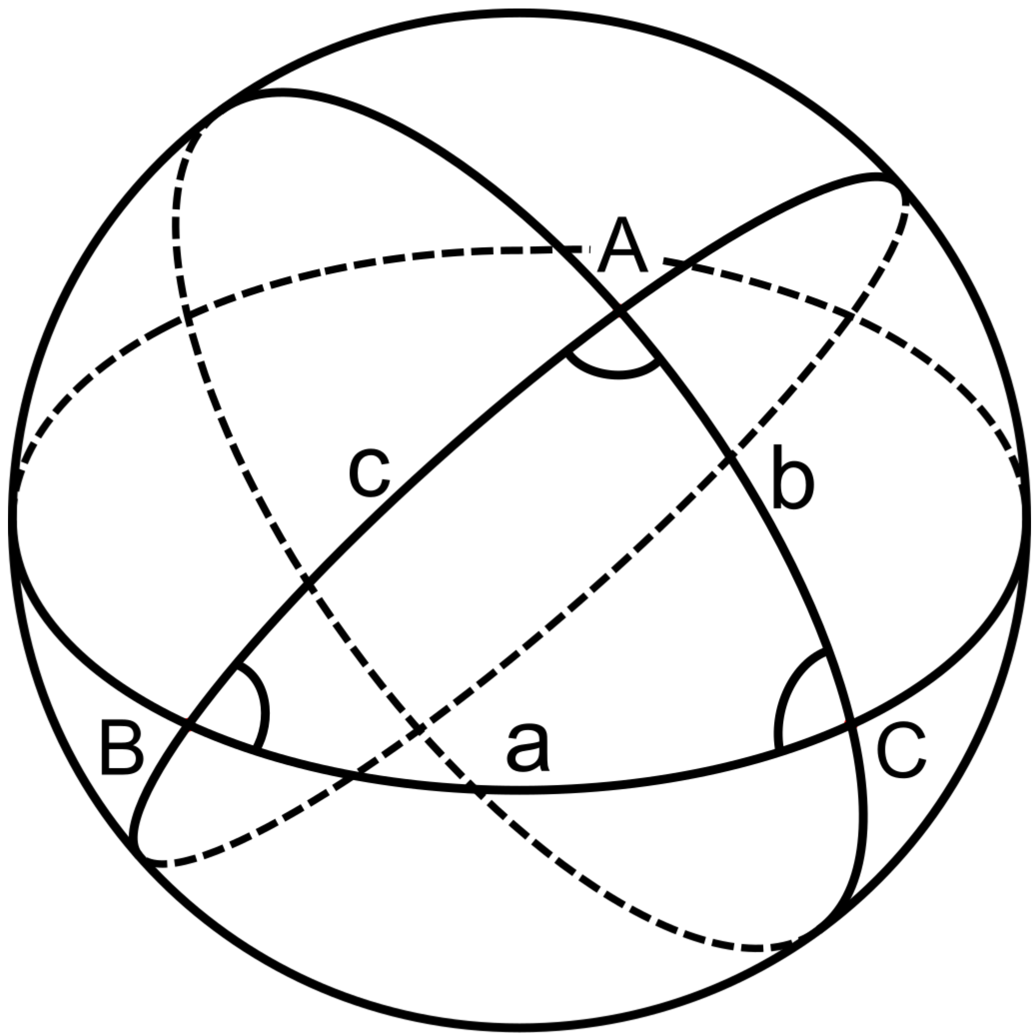
-Wikipedia

Great circle distances



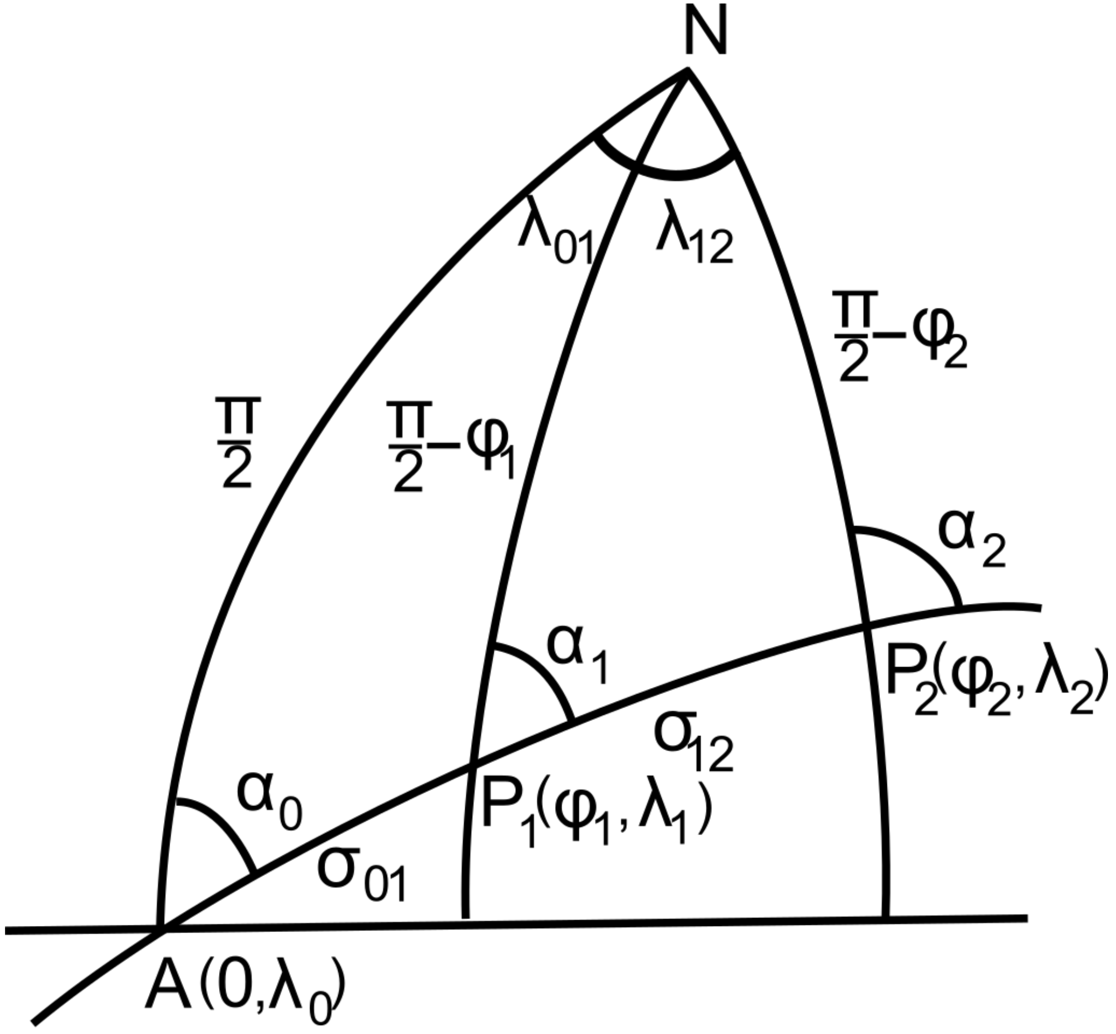
-https://www.mathworks.com/help/map/rhumb-lines.html

Spherical Trig



-Wikipedia

I’m not smart enough for this



-Wikipedia

Find code on the internet

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

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
$\theta_b = \text{acos}((\sin\varphi_1 - \sin\varphi_2 \cdot \cos\delta_{12}) / (\sin\delta_{12} \cdot \cos\varphi_2))$ — 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 $\varphi_1, \lambda_1, \theta_{13}$: 1st start point & (initial) bearing from 1st point towards intersection point
$\varphi_2, \lambda_2, \theta_{23}$: 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

http://www.movable-type.co.uk/scripts/latlong.html

The code on the internet doesn't work

```
Test Detail Summary
  Tests.TrackTests.Calc_Distance(one: [0, 0], two: [0, 10], point: [5, 5], distance: 556)
     Source: TrackTests.cs line 260
     Duration: 2.6 sec

  Message:
     Assert.Equal() Failure: Values are not within 0 decimal places
     Expected: 556 (rounded from 556)
     Actual:   -786 (rounded from -785.76148093750851)

  Stack Trace:
     TrackTests.Calc_Distance(Single[] one, Single[] two, Double[] point, Double distance) line 273
     --- End of stack trace from previous location ---
```

Floating point numbers

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

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

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

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

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

http://www.movable-type.co.uk/scripts/latlong.html

Haversine formula

To solve for the distance d, apply the archaversine (inverse haversine) to $\operatorname{hav}(\theta)$ or use the arcsine (inverse sine) function:

$$d = r \operatorname{archav}(\operatorname{hav}\theta) = 2r \arcsin\left(\sqrt{\operatorname{hav}\theta}\right)$$

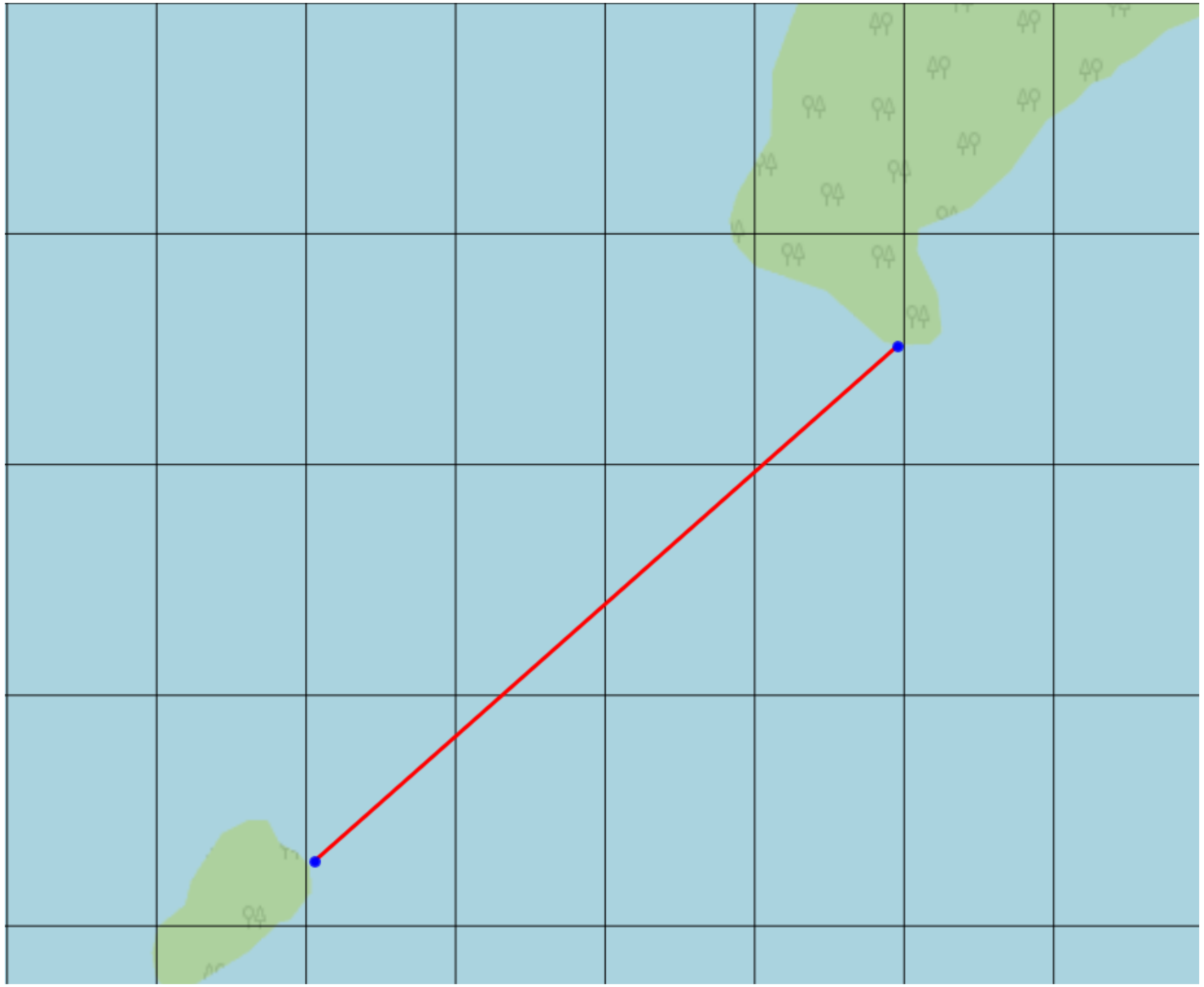
or more explicitly:

$$\begin{aligned}
d &= 2r \arcsin\left(\sqrt{\operatorname{hav}(\Delta\varphi) + (1 - \operatorname{hav}(\Delta\varphi) - \operatorname{hav}(2\varphi_{\mathrm{m}})) \cdot \operatorname{hav}(\Delta\lambda)}\right) \\
&= 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\varphi}{2}\right) + \left(1 - \sin^2\left(\frac{\Delta\varphi}{2}\right) - \sin^2(\varphi_{\mathrm{m}})\right) \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right)}\right) \\
&= 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\varphi}{2}\right) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right)}\right) \quad [9] \\
&= 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\varphi}{2}\right) \cdot \cos^2\left(\frac{\Delta\lambda}{2}\right) + \cos^2(\varphi_{\mathrm{m}}) \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right)}\right) \\
&= 2r \arcsin\left(\sqrt{\frac{1 - \cos(\Delta\varphi) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot (1 - \cos(\Delta\lambda))}{2}}\right)
\end{aligned}$$

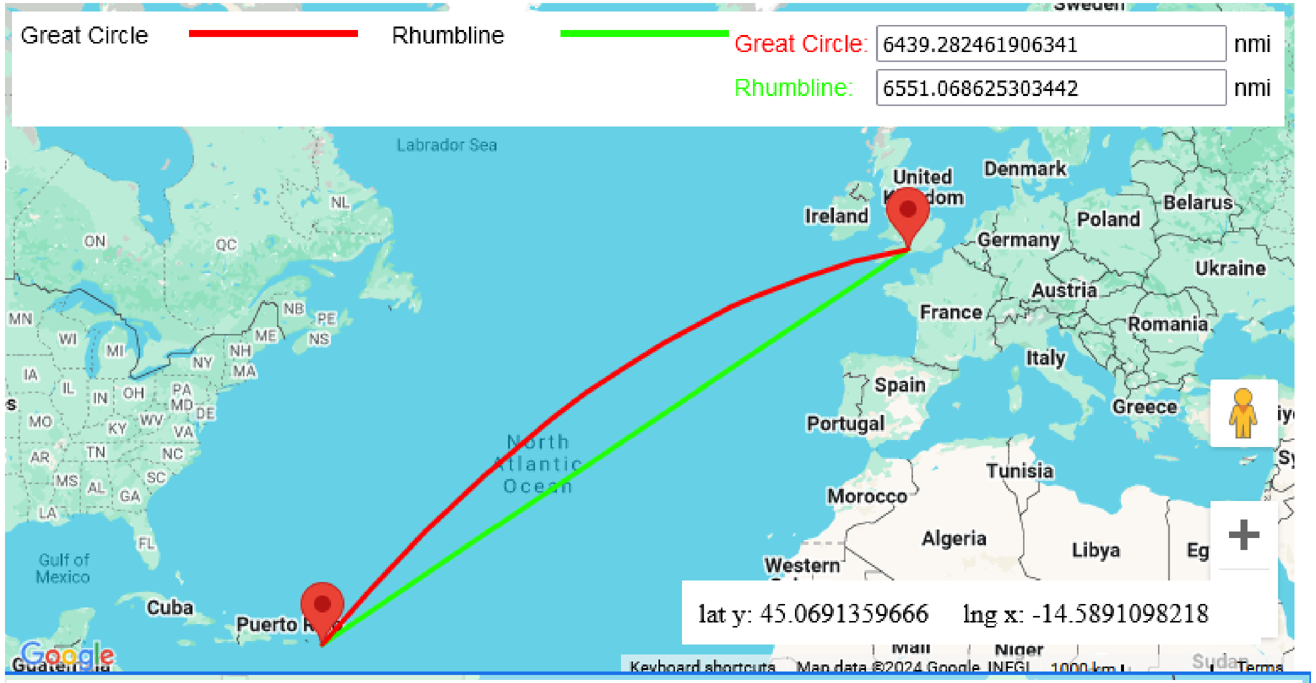
where $\varphi_{\mathrm{m}} = \dfrac{\varphi_2 + \varphi_1}{2}$.

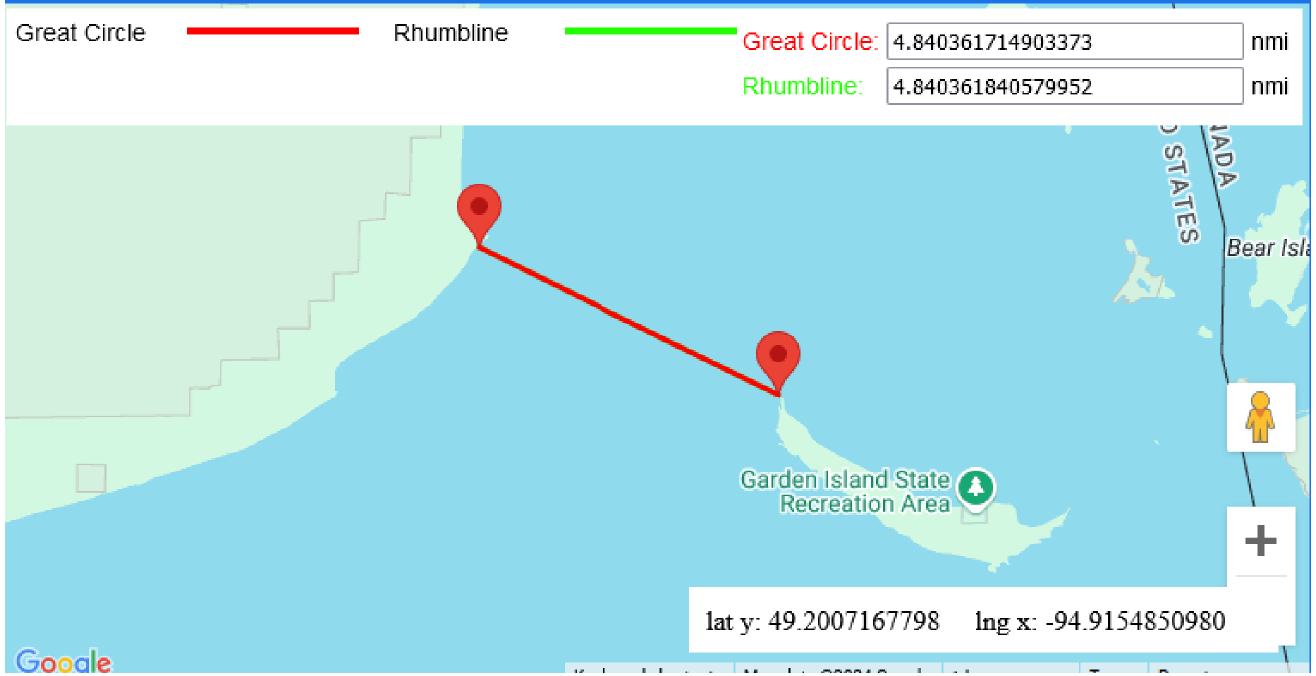
-Wikipedia

This was supposed to be easy

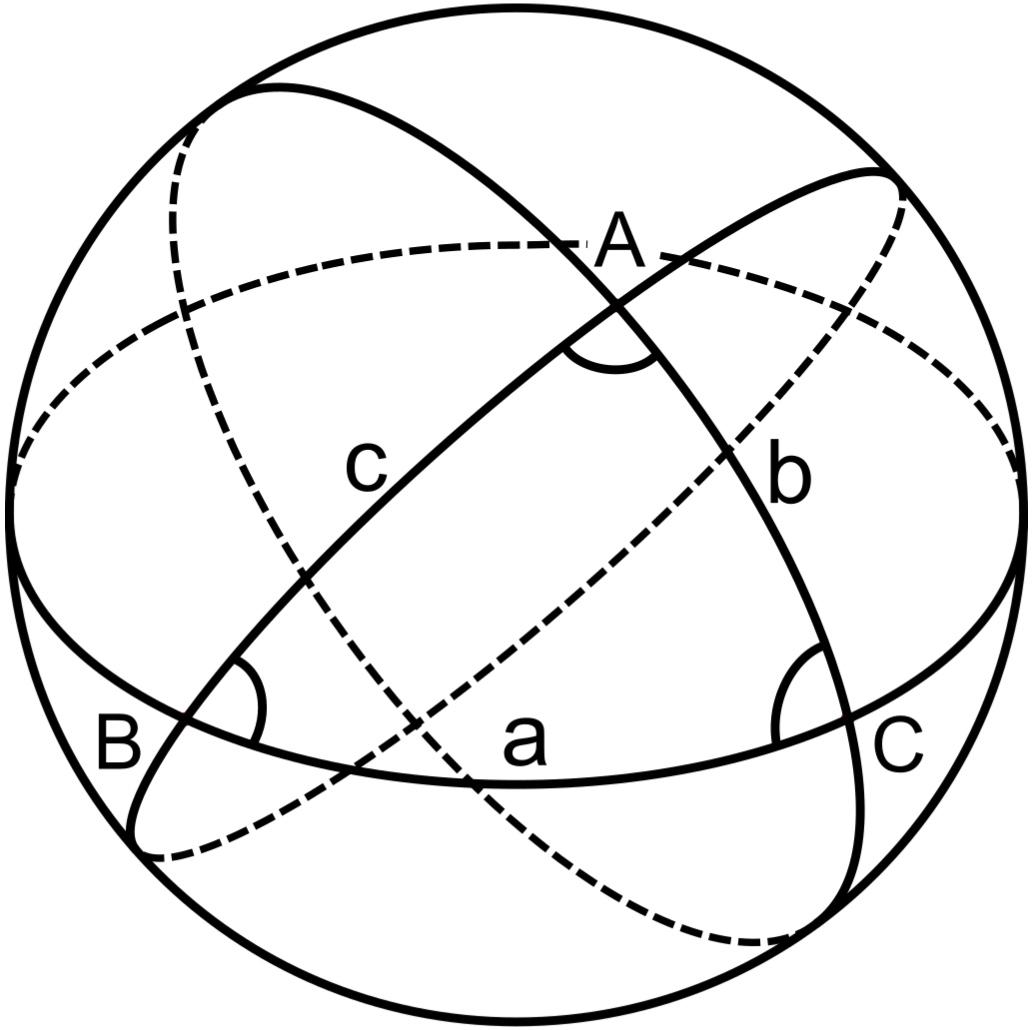


What is the difference actually?





None of it mattered



-Wikipedia

I am smart enough for this

```
// find points before and after finish line

double slope = (one.Latitude - two.Latitude) / (one.Longitude - two.Longitude);
double intercept = two.Latitude - slope * one.Longitude;

bool? side = null;
XmlNode prev = null;

foreach (XmlNode wpt in track.DocumentElement.ChildNodes)
{
    double lat = Convert.ToDouble(wpt.Attributes["lat"].Value);
    double lon = Convert.ToDouble(wpt.Attributes["lon"].Value);

    if (lat != null && lon != null)
    {

        if (side == null)
        {
            side = Side(slope, intercept, lon, lat);
        }
        else if (Side(slope, intercept, lon, lat) != side)
        {
            // check that the side switching occurs within the two ends

            if (WithinBox(lat, lon, one, two))
            {

                DateTime currTime = DateTime.Parse(wpt.Attributes["time"].Value);
                DateTime prevTime = DateTime.Parse(prev.Attributes["time"].Value);


                long wptSec = ((DateTimeOffset)currTime).ToUnixTimeMilliseconds();
                long prevSec = ((DateTimeOffset)prevTime).ToUnixTimeMilliseconds();

                finish = new DateTime(1970, 1, 1, 0, 0, 0, 0, DateTimeKind.Utc).AddMilliseconds((wptSec - prevSec) / 2 + prevSec);

                break;
            }
        }
    }

    prev = wpt;
}
```

Lessons Learned

- Scope smaller than you think
- Build in buffer time
- Domain knowledge is both helpful and dangerous
- Check requirements
- I will now avoid spherical trig and latitude/longitude calculations