Navigating the Earth with pure SPARQL

Damien Graux(✉)

Inria, Université Côte d'Azur, CNRS, I3S, France
damien.graux@inria.fr

Abstract. Let's assume you are on a boat and have to navigate only basic items such as a map or a compass, and a ..... SPARQL engine! How to set a course? How to compute distances? In this article, we present a set of SPARQL code-blocks to be used in such situation and more generally in use-cases where practitioners need to compute mathematical calculus on geo-data represented by (lat,lon) pairs.

1 Introduction

During the past two decades, Semantic Web technologies for the Web have been developed and it is now possible to produce, share, analyze and interlink large knowledge graphs (sometimes containing billions of facts) structured using the RDF W3C standard [11]. Additionally, the W3C has standardized SPARQL [13], the *de facto* query language dedicated to RDF which has been more recently improved to add new features, see *e.g.* [14] for its current version. In parallel, hundreds of resources have been published online[1] and are often accompanied by associated SPARQL endpoints on which users are able to send queries in order to retrieve pieces of data. Among the various types of data able to be found within these resources (*e.g.* dates, texts in natural language), it is common to have geo-spatial data[2]; in particular, they are usually represented using their coordinates through the pair (latitude,longitude).

Consequently, use cases sometimes require the computation involving lat/lon points according to specific patterns, to *e.g.* filter by distance. However, in the current version of the standard[3], only the four basic mathematical operators are available ($+$, $-$, $*$, $/$) and some basic predefined functions, such as `CEIL` or `FLOOR`. To address this lack in the standard, some popular evaluators allow extensions to the SPARQL language to cover popular mathematical functions (*e.g.* trigonometric operations). Nonetheless, this results in queries specifically built to be executed by a specific system and which therefore cannot be shared between users without adapting them beforehand.

Regarding geo-spatial RDF data in particular, the Open Geospatial Consortium has been active to propose standardized methods to structure geo-data

[1] See the current number of listed resources gathered on the Linked Open Data cloud: 1 301 as of May 2021. https://lod-cloud.net/

[2] For example, WikiData contains information about specific places such as cities.

[3] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#expressions

and to retrieve information stored as such. In 2011, Battle & Kolas presented GeoSPARQL [1]: a geographic query language for RDF data. As of now, multiple triplestores offer (partial) support of GeoSPARQL [8].

Nevertheless, even if GeoSPARQL provides a wide set of functions[4], most of them focus on topological geometries. In this study, we want to mainly focus on navigation-related computations on Earth, and this implies to design new specific functions. Moreover, in order to guarantee interoperability between triplestores, (and to serve as a showcase of SPARQL's potential), we choose to make only use of SPARQL 1.1 standard operators. Therefore, in this article, we provide a set of ready to be used SPARQL instructions to compute various operations on (lat,lon)-represented points. In short, a SPARQL practitioner can directly copy and paste the block of code in order to enrich her query with her own additional specific computations.

The rest of the article is organised as follows: first we provide the reader with brief background notions about Earth-surface geometry (often used by sailors) in Section 2. Second, in Section 3, we describe how standard SPARQL can be used to compute advanced mathematical functions. Then, in the main Section 4, we provide the reader with several implementations of useful geometrical use-cases. Finally, after reminding the related work in Section 5, we conclude in Section 6.

2 Background on Earth-surface geometry

Great-circle path. A great circle, also known as an orthodrome, of a sphere is the intersection of the sphere and a plane that passes through the center point of the sphere. A great circle is the largest circle that can be drawn on any given sphere. Any diameter of any great circle coincides with a diameter of the sphere, and therefore all great circles have the same center and circumference as each other. For most pairs of distinct points on the surface of a sphere, there is a unique great circle through the two points[5]. Great-circle navigation or orthodromic navigation consists of navigating a vessel (ship or aircraft) along a great circle. Such routes yield the shortest distance between two points on the globe.

Rhumb line. In navigation, a rhumb line, or loxodrome is an arc crossing all meridians of longitude at the same angle, that is, a path with constant bearing as measured relative to true north.

Comparison. Figure 1 visually shows the difference between the path following the great-circle (in cyan) and the one on a rhumb line (in red). Practically, on a great circle, the bearing[6] to the destination point does not remain constant.

[4] http://www.opengis.net/def/function/geosparql/

[5] Exception: a pair of antipodal points, for which there are infinitely many great circles.

[6] In navigation, bearing is the horizontal angle between the direction of an object and another object, or between it and that of true north.

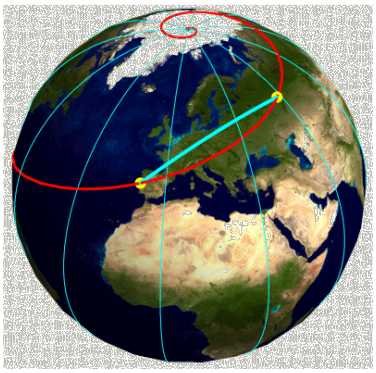

Fig. 1. A loxodrome is represented in red, passing through two points; the shortest great circle arc defined by the same two points is displayed in blue (shortest path).

3 Geometrical computations with only standard SPARQL

Depending on the use cases, one might have to deal with geospatial data and more particularly with (latitude,longitude) coordinates. If *e.g.* the discovery of points is the only thing needed, then a classic SPARQL query can be built in order to retrieve the relevant pieces of information. However, practitioners - when dealing with geo-data- often have to compare records based on the distance between points for example. For these use cases, it can be very hard to implement the filters directly within the SPARQL queries, as the SPARQL 1.1 standard only allows the four basic mathematical operators ($+$, $-$, $*$, $/$). The chosen approach is thereby to (1) either rely on built-in specific mathematical functions provided by the SPARQL engine itself or (2) to treat results afterwards (*i.e.* locally once having obtained results from the engine). The first case breaks the interoperability of the SPARQL query itself as it becomes engine-dependent. The second case moves computations out from the SPARQL engine and implies the query-designer to build a query having a larger scope so to further refine/filter the results later on, losing therefore in performance and network traffic.

In particular, geodata related computations, since (lat,lon) pairs use a spherical coordinate system, rely on trigonometric functions most of the time. However, such functions (*e.g.* cos, tan...) are not available by default in SPARQL. In order to compute mathematical expressions in SPARQL, one solution is to approximate the results using Taylor series as presented by Graux *et al.* [5]. For example, close to 0, sin can be developed using the following series (which makes only use of the 4 "basic" operators):

$$\sin x = \sum_{k=0}^{+\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

Easily, this trigonometric function can be coded in SPARQL and using the first terms could be enough to for example compare/sort entities. The first term of the sin series can be expressed as follows using SPARQL primitives[7]:

[7] Relying on MINDS to generate the bindings from the mathematical formula [5].

```
VALUES ?2PI {"6.28318530718"^^xsd:double}
BIND ( (?X-?2PI*FLOOR(?X/?2PI) ) AS ?X_in_0_2PI )
BIND ( (1*( 1* ?X_in_0_2PI )/1.0 ) AS ?sin_first_term )
```

In order to gain in precision, the value of the variable `?X` is projected within $[0, 2\pi]$ taking advantage of the 2π-periodicity of the sin.

More generally, this approach can be declined to various mathematical functions which admit Taylor series. This approach has the advantage of being fully compliant with the SPARQL standard. Nevertheless, it is practically prone to error if one has to write down manually the SPARQL bindings. Therefore, to ease the process, in Section 4, we provide the readers with the SPARQL code blocks for several common geospatial computations.

4 Computations on (Latitude,Longitude) Pairs

In the current section, we present a set of formulas and SPARQL-code blocks[8] to compute operations on (lat,lon) pairs. It is worth noting that these formulas are for calculations considering a spherical earth, *i.e.* ignoring the ellipsoidal effects.

General SPARQL Bindings. In the rest of the Section, we present several block of bindings to operate geospatial computations. In order to factorise the syntax, there are several bindings that could be put upfront the following blocks, such as constants like π or the Earth radius:

```
# Useful values.
BIND ( xsd:double("3.14159265359") AS ?PI  ) # π with 11 digits.
BIND ( xsd:double("6.28318530718") AS ?2PI ) # 2π with 11 digits.
BIND ( xsd:double("6371") AS ?E_radius )     # Earth's radius, in km.
```

Similarly, we assume that the considered variables in the SPARQL query for respectively the latitude and the longitude are `?lat` and `?lon` when represented in degrees and `?lar` and `?lor` when in radians. More generally, we can also use the following bindings to convert a variable in degrees to its radian-equivalent or to designate the *deltas* between 2 pairs of coordinates, *i.e.* `?dellar` and `?dellor`:

```
BIND ( (xsd:double(?lat) * ?PI/180) AS ?lar ) # degrees → radians.

# Having two pairs of coordinates, below are the deltas in radians.
BIND ( ((xsd:double(?lat2)-xsd:double(?lat1)) * ?PI/180) AS ?dellar )
BIND ( ((xsd:double(?lon2)-xsd:double(?lon1)) * ?PI/180) AS ?dellor )
```

[8] For space limitation, we do not provide all and limit the number of terms in series.

4.1 Considering the great-circle

Distance between two points. In order to calculate the great-circle distance between two points[9], the result is given using the **haversine**[10] **formula**:

$$a = \sin^2(dellar/2) + \cos(lor_1).\cos(lor_2).\sin^2(dellor/2)$$
$$c = 2.atan2(\sqrt{a}, \sqrt{1-a})$$
$$d = \text{E\_radius} \; . \; c$$

It is worth noting that this formula involves 7 trigonometric functions and 2 square roots. In particular, it requires an $atan2$ which is defined as follows:

$$atan2(y,x) = \begin{cases} 2.\arctan\dfrac{y}{\sqrt{x^2+y^2}+x} & \text{if } x > 0, \\ \pi & \text{if } x < 0 \text{ and } y = 0, \\ \text{undefined} & \text{if } x = 0 \text{ and } y = 0. \end{cases}$$

```
# A
BIND ((0+1*(1* ((?dellar/2)-?2PI*FLOOR((?dellar/2)/?2PI) ))/1.0 +-1*(1*
↪  ((?dellar/2)-?2PI*FLOOR((?dellar/2)/?2PI) )*
↪  ((?dellar/2)-?2PI*FLOOR((?dellar/2)/?2PI) )*
↪  ((?dellar/2)-?2PI*FLOOR((?dellar/2)/?2PI) ))/6.0 )AS ?Asub1)
BIND ((0+1*(1)/1.0 +-1*(1* (?lar1)-?2PI*FLOOR(?lar1)/?2PI) )*
↪  (?lar1)-?2PI*FLOOR(?lar1)/?2PI) ))/2.0 )AS ?Asub2)
BIND ((0+1*(1)/1.0 +-1*(1* (?lar2)-?2PI*FLOOR(?lar2)/?2PI) )*
↪  (?lar2)-?2PI*FLOOR(?lar2)/?2PI) ))/2.0 )AS ?Asub3)
BIND ((0+1*(1* ((?dellor/2)-?2PI*FLOOR((?dellor/2)/?2PI) ))/1.0 +-1*(1*
↪  ((?dellor/2)-?2PI*FLOOR((?dellor/2)/?2PI) )*
↪  ((?dellor/2)-?2PI*FLOOR((?dellor/2)/?2PI) )*
↪  ((?dellor/2)-?2PI*FLOOR((?dellor/2)/?2PI) ))/6.0 )AS ?Asub4)
BIND ( ( FLOOR(( (1*(?Asub1)*(?Asub1)) +?Asub2*?Asub3*
↪  (1*(?Asub4)*(?Asub4)) )*10000)/10000 ) AS ?A )

# √A
BIND ((0+(1*(((?A)-1)/((?A)+1)))/1.0
↪  +(1*(((?A)-1)/((?A)+1))*(((?A)-1)/((?A)+1))*(((?A)-1)/((?A)+1)))/3.0
↪  +(1*(((?A)-1)/((?A)+1))*(((?A)-1)/((?A)+1))*(((?A)-1)/((?A)+1))*
↪  (((?A)-1)/((?A)+1))*(((?A)-1)/((?A)+1)))/5.0 )AS ?sub1)
BIND ((0+(1)/1.0 +(1*?sub1)/1.0 +(1*?sub1*?sub1)/2.0 )AS ?sub2)
# √(1-A)
BIND ((0+(1*(((1-?A)-1)/((1-?A)+1)))/1.0
↪  +(1*(((1-?A)-1)/((1-?A)+1))*(((1-?A)-1)/((1-?A)+1))*
↪  (((1-?A)-1)/((1-?A)+1)))/3.0+(1*(((1-?A)-1)/((1-?A)+1))*
↪  (((1-?A)-1)/((1-?A)+1))*(((1-?A)-1)/((1-?A)+1))*(((1-?A)-1)/
↪  ((1-?A)+1))*(((1-?A)-1)/((1-?A)+1)))/5.0 )AS ?sub3)
```

[9] The shortest distance over the Earth's surface, considering an "as-the-crow-flies" distance, while ignoring hills and topography

[10] The *half versed sine*: $hav(\theta) = \dfrac{1}{2}.versine(\theta) = \sin^2\left(\dfrac{\theta}{2}\right) = \dfrac{1-\cos(\theta)}{2}$.

```
BIND ((0+(1)/1.0 +(1*?sub3)/1.0 +(1*?sub3*?sub3)/2.0 )AS ?sub4)

# sqrt((sqrt(A))^2 + (sqrt(1 - A))^2)
BIND (((?sub2)*(?sub2)+(?sub4)*(?sub4))AS ?sub5)
BIND ((0+(1*(( ?sub5-1)/( ?sub5+1)))/1.0 +(1*(( ?sub5-1)/( ?sub5+1))*((
↪  ?sub5-1)/( ?sub5+1))*(( ?sub5-1)/( ?sub5+1)))/3.0 +(1*(( ?sub5-1)/(
↪  ?sub5+1))*(( ?sub5-1)/( ?sub5+1))*(( ?sub5-1)/( ?sub5+1))*((
↪  ?sub5-1)/( ?sub5+1))*(( ?sub5-1)/( ?sub5+1)))/5.0 )AS ?sub6)
BIND ((0+(1)/1.0 +(1*?sub6)/1.0 +(1*?sub6*?sub6)/2.0 )AS ?sub7)

# arctan( sqrt(A) / (sqrt((sqrt(A))^2 + (sqrt(1 - A))^2) + sqrt(1 - A)) )
BIND (((?sub2)/( ?sub7+(?sub4)))AS ?sub8)
BIND ((0+1*(1* ?sub8)/1.0 +-1*(1* ?sub8* ?sub8* ?sub8)/3.0 +1*(1* ?sub8*
↪  ?sub8* ?sub8* ?sub8* ?sub8)/5.0 )AS ?sub9)

# Grand finale
BIND ( ( FLOOR((2 * ?sub9)*10000)/10000 ) AS ?C )
BIND ( (2 * ?E_radius * ?C) AS ?distance )
```

As presented displayed above, this haversine formula can be converted in SPARQL bindings, using series approximations. For clarity reason, we only used the first 3 terms for the series. Practically, such set of bindings is tedious to write down manually and the use of a tool (such as MINDS [5]) is much safer to avoid typos and guarantee a SPARQL compliant output.

Initial bearing. Most of the time, while navigating following a defined course, the current bearing will vary to follow a great circle path. The following formula computes the initial bearing[11] in radians:

$$\theta = atan2(\sin(dellor)\cos(lar_2)\ ,\ \cos(lar_1)\sin(lar_2) - \sin(lar_1)\cos(lar_2)\cos(dellor))$$

where (lar_1, lor_1) is the starting point and (lar_2, lor_2) the end point.

Mid-point. Similarly, the half-way point (lat_m, lon_m) of a course from (lat_1, lon_1) to (lat_2, lon_2) may be calculated using:

$$\begin{aligned} B_x &= \cos(lar_2).\cos(dellor) \\ B_y &= \cos(lar_2).\sin(dellor) \\ lar_m &= atan2\left(\sin(lar_1) + \sin(lar_2)\ ,\ \sqrt{(\cos(lar_1) + B_x)^2 + {B_y}^2}\right) \\ lor_m &= lor_1 + atan2\left(B_y\ ,\ \cos(lar_1) + B_x\right) \end{aligned}$$

[11] Also known as *forward azimuth*.

Intermediate point. More generally, an intermediate point at any fraction f along the great circle path between two points (lat_1, lon_1) and (lat_2, lon_2) can be calculated:

$$\begin{aligned} a &= \frac{\sin((1-f).\delta)}{\sin\delta} \\ b &= \frac{\sin(f.\delta)}{\sin\delta} \\ x &= a\cos(lar_1)\cos(lor_1) + b\cos(lar_2)\cos(lor_2) \\ y &= a\cos(lar_1)\sin(lor_1) + b\cos(lar_2)\sin(lor_2) \\ z &= a\sin(lar_1) + b\sin(lar_2) \\ lar_i &= atan2\left(z\ ,\ \sqrt{x^2+y^2}\right) \\ lor_i &= atan2(y\ ,\ x) \end{aligned}$$

with δ the angular distance $\frac{d}{E\_radius}$ between the two points. We note that for $f = 0$ and $f = 1$ respectively, we have point 1 and point 2.

Closest point to the poles. Using Clairaut's formula[12], we can obtain the maximum latitude lar_M of a great circle path, given a bearing B and latitude lar on the great circle:

$$lar_M = \text{acos}(|\sin B.\cos(lar)|)$$

Destination point. Considering an initial point (lar_1, lor_1), an initial bearing B and a distance d, it is possible to know the arrival point following a great-circle route:

$$\begin{aligned} lar_f &= \text{asin}(\sin(lar_1).\cos\delta + \cos(lar_1).\sin\delta.\cos B) \\ lor_f &= lar_1 + atan2(\sin B.\sin\delta.\cos(lar_1)\ ,\ \cos\delta - \sin(lar_1).\sin(lar_f)) \end{aligned}$$

with δ the angular distance $\frac{d}{E\_radius}$.

4.2 Considering rhumb lines

Most of the features presented for great-circle ways are applicable to rhumb lines too. As explained in Section 2, rhumb lines differ from great-circle paths in the sense that they consider constant bearing, crossing all meridians at the same angle. Traditionally, sailors used to 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. In addition, rhumb lines are straight lines on Mercator projection maps. However, rhumb lines are generally longer than great-circle routes.

[12] https://en.wikipedia.org/wiki/Clairaut%27s_relation_(differential_geometry)

Practically, most of rhumb-related computations are using the *inverse Gudermannian function*[13], which gives the height on a Mercator projection map of a given latitude φ:

$$gd^{-1} : \varphi \mapsto gd^{-1}(\varphi) = ln\left(\tan\left(\frac{\pi}{4} + \frac{\varphi}{2}\right)\right)$$

```
# π/4 + φ/2 renamed ?X
BIND ( ( FLOOR((?PI/4 + ?lar/2)*10000)/10000 ) AS ?X )

# Series of sin(?X)
BIND ((0+1*(1* ((?X)-?2PI*FLOOR((?X)/?2PI) ))/1.0 +-1*(1* ((?X)-?2PI*FLOOR((?X)/?2PI) )*
↪  ((?X)-?2PI*FLOOR((?X)/?2PI) )* ((?X)-?2PI*FLOOR((?X)/?2PI) ))/6.0 +1*(1*
↪  ((?X)-?2PI*FLOOR((?X)/?2PI) )* ((?X)-?2PI*FLOOR((?X)/?2PI) )*
↪  ((?X)-?2PI*FLOOR((?X)/?2PI) )* ((?X)-?2PI*FLOOR((?X)/?2PI) )*
↪  ((?X)-?2PI*FLOOR((?X)/?2PI) ))/120.0 )AS ?sub1)

# Series of cos(?X)
BIND ((0+1*(1)/1.0 +-1*(1* ((?X)-?2PI*FLOOR((?X)/?2PI) )* ((?X)-?2PI*FLOOR((?X)/?2PI)
↪  ))/2.0 +1*(1* ((?X)-?2PI*FLOOR((?X)/?2PI) )* ((?X)-?2PI*FLOOR((?X)/?2PI) )*
↪  ((?X)-?2PI*FLOOR((?X)/?2PI) )* ((?X)-?2PI*FLOOR((?X)/?2PI) ))/24.0 )AS ?sub2)

# Series for ln(tan(?X)) = ln(sin(?X)/cos(?X))
BIND (( 2*(0+(1*(((?sub1/?sub2)-1)/((?sub1/?sub2)+1)))/1.0
↪  +(1*(((?sub1/?sub2)-1)/((?sub1/?sub2)+1))*(((?sub1/?sub2)-1)/
↪  ((?sub1/?sub2)+1))*(((?sub1/?sub2)-1)/((?sub1/?sub2)+1)))/3.0
↪  +(1*(((?sub1/?sub2)-1)/((?sub1/?sub2)+1))*(((?sub1/?sub2)-1)/
↪  ((?sub1/?sub2)+1))*(((?sub1/?sub2)-1)/((?sub1/?sub2)+1))*(((?sub1/?sub2)-1)/
↪  ((?sub1/?sub2)+1))*(((?sub1/?sub2)-1)/((?sub1/?sub2)+1)))/5.0 ))AS ?sub3)
BIND ( ( FLOOR((?sub3)*10000)/10000 ) AS ?Gudermannian )
```

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 (using Pythagoras' theorem); but the distortion of the projection needs to be compensated for. On a constant latitude course (travelling East-West), this compensation is simply $\cos(lar)$; in the general case, it is $dellar/delproj$ where $delproj = ln\left(\frac{\tan(\pi/4 + lar_2/2)}{\tan(\pi/4 + lar_1/2)}\right)$ *i.e.* the projected latitude difference. Which leads to the following formula:

$$q = \frac{dellar}{delproj} \qquad \text{(or } \cos(lar) \text{ for E-W lines)}$$

$$d = \sqrt{dellar^2 + q^2.dellor^2} \text{ . E\_radius} \qquad \text{(Pythagoras)}$$

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

$$B = atan2(dellor, delproj)$$

[13] https://en.wikipedia.org/wiki/Gudermannian_function

4.3 Distance to the horizon from the "crow's nest"

At a height h above the ground, the distance to the horizon d, is given by: $d = \sqrt{(2 * R * h/b)}$ with $b = 0.8279$ is a factor that accounts for atmospheric refraction and depends on the atmospheric temperature lapse rate, which is taken to be standard[14], leading us to the following SPARQL bindings:

```
BIND ( "0.8279" AS ?b )
BIND ( (2*xsd:double(?E_radius)*xsd:double(?h)/xsd:double(?b)) AS ?int )
BIND ((0+(1*(((?int)-1)/((?int)+1)))/1.0
+(1*(((?int)-1)/((?int)+1))*(((?int)-1)/((?int)+1))*(((?int)-1)/((?int)+1)))/3.0
+(1*(((?int)-1)/((?int)+1))*(((?int)-1)/((?int)+1))*
↪  (((?int)-1)/((?int)+1))*(((?int)-1)/((?int)+1))*(((?int)-1)/((?int)+1)))/5.0
+(1*(((?int)-1)/((?int)+1))*(((?int)-1)/((?int)+1))*
↪  (((?int)-1)/((?int)+1))*(((?int)-1)/((?int)+1))*(((?int)-1)/((?int)+1))*
↪  (((?int)-1)/((?int)+1))*(((?int)-1)/((?int)+1)))/7.0
)AS ?sub1)
BIND ((0+(1)/1.0+(1*?sub1)/1.0+(1*?sub1*?sub1)/2.0 + (1*?sub1*?sub1*?sub1)/6.0 )AS ?sub2)
BIND ( ( FLOOR((?sub2)*10000)/10000 ) AS ?distance )
```

4.4 Faster approximations for great-circle distances

In practice, if performance is an issue and accuracy less important, for small distances Pythagoras' theorem[15] can be used on an *equirectangular projection*[16]:

$$
\begin{aligned}
x &= dellor \ . \ \cos\left(\frac{lar_1 + lar_2}{2}\right) \\
y &= dellar \\
d &= \text{E\_radius} \ . \ \sqrt{x^2 + y^2}
\end{aligned}
$$

This approximation uses one trigonometric and one square root function (as against the 7 trigonometric plus 2 square roots for haversine fescribed above). Technically, the accuracy is of the equirectangular method varies: along meridians there are no errors, otherwise it depends on distance, bearing, and latitude; however the errors are small enough for many purposes. Such a method would lead to the following SPARQL code block of bindings:

```
BIND ((0+1*(1)/1.0 +-1*(1* (((?lar1+?lar2)/2)-?2PI*FLOOR(((?lar1+?lar2)/2)/?2PI) )*
↪  (((?lar1+?lar2)/2)-?2PI*FLOOR(((?lar1+?lar2)/2)/?2PI) ))/2.0 +1*(1*
↪  (((?lar1+?lar2)/2)-?2PI*FLOOR(((?lar1+?lar2)/2)/?2PI) )*
↪  (((?lar1+?lar2)/2)-?2PI*FLOOR(((?lar1+?lar2)/2)/?2PI) )*
↪  (((?lar1+?lar2)/2)-?2PI*FLOOR(((?lar1+?lar2)/2)/?2PI) )*
↪  (((?lar1+?lar2)/2)-?2PI*FLOOR(((?lar1+?lar2)/2)/?2PI) ))/24.0 )AS ?sub1)
BIND ( ( FLOOR((xsd:double(?dellor)*?sub1)*10000)/10000 ) AS ?X )
BIND ( ( FLOOR(( (1*?X*?X) + (1*?dellar*?dellar) )*10000)/10000 ) AS ?root )
BIND ((0+(1*(((?root)-1)/((?root)+1)))/1.0
↪  +(1*(((?root)-1)/((?root)+1))*(((?root)-1)/((?root)+1))*(((?root)-1)/((?root)+1)))/3.0
↪  +(1*(((?root)-1)/((?root)+1))*(((?root)-1)/((?root)+1))*(((?root)-1)/((?root)+1))
↪  *(((?root)-1)/((?root)+1))*(((?root)-1)/((?root)+1)))/5.0 )AS ?sqsub1)
BIND ((0+(1)/1.0 +(1*?sqsub1)/1.0 +(1*?sqsub1*?sqsub1)/2.0 )AS ?sqsub2)
BIND ( ( FLOOR((xsd:double(?E_radius)*?sqsub2)*10000)/10000 ) AS ?d )
```

[14] See Table 12 from the American Practical Navigator [2].
[15] https://en.wikipedia.org/wiki/Pythagorean_theorem
[16] https://en.wikipedia.org/wiki/Equirectangular_projection

Alternatively, the polar coordinate "flat-earth" formula can be used. Considering the co-latitudes $colar_1 = \frac{\pi}{2} - lar_1$ and $colar_2 = \frac{\pi}{2} - lar_2$, then:

$$d = \text{E\_radius} \cdot \sqrt{{colar_1}^2 + {colar_2}^2 - 2.colar_1.colar_2.\cos(dellor)}$$

5 Related Work

Due to SPARQL's lack of essential basic math functions in its standard[17], different approaches have emerged to serve this need.

Some SPARQL evaluators actually do not give the possibility of computing mathematical functions inside queries at all. This is for instance the case with 4store [6], RDF3X [12] or SPARQLGX [4] which are nonetheless popular evaluators from the literature known for their performances evaluating conjunctive queries. However, arguably, the research focus of these systems was on optimization of joins and indexes and less on feature completeness.

Currently, all practical relevant SPARQL evaluators offer the opportunity of computing mathematical functions inside the `BIND` elements and projections. While the SPARQL standard defines the built-in functions as part of the syntax[18], the widely adopted approach by evaluator developers is to take advantage of the `Function Call` rule, which allows arbitrary IRIs to be used as function names. Hence, function extensions typically require no changes to the SPARQL syntax. However, the lack of standardization implies two drawbacks:

- Firstly, the namespaces, local names and signatures of functions may vary between SPARQL engines, which makes it tedious –if not prohibitive– to exchange backends.
- Secondly, the means of computation of a function and therefore the results may differ between evaluators.

All popular SPARQL evaluators –often used to serve public endpoints– such as Virtuoso [3], Jena-Fuseki [7], GraphDB[19] and Stardog[20] feature mathematical functions, yet, using different IRIs. For instance, Virtuoso uses the `bif:` namespace, whereas Stardog reuses the XPath function namespace[21]. Using such an approach of naming differently similar function/operator[22] implies a loss of interoperability, especially, it make the design of federated SPARQL queries far more complex.

[17] Currently, the *SPARQL 1.2 Community Group* which aims to advance SPARQL functionalities, is describing several mathematical operators that could be added in the next iteration of the standard. https://github.com/w3c/sparql-12/

[18] https://www.w3.org/TR/sparql11-query/#grammar

[19] https://ontotext.com/products/graphdb/

[20] https://www.stardog.com/

[21] https://www.w3.org/2005/xpath-functions/math#

[22] Implementations for built-in STDEV in Virtuoso, Fuseki, Stardog, Sesame: https://gist.github.com/albertmeronyo/c6ab285d0b73b05392e2f9b8a5bbea82

Regarding the spatial aspects, several solutions have been proposed by the community to represent and query geo-data in RDF. For instance, stRDF and stSPARQL [9,10] were proposed to deal with spatiotemporal data. Similarly, the Open Geospatial Consortium runs the GeoSPARQL initiatives [1] to provide a geographic query language for RDF data together with methods to structure and represent geo-information. Nowadays, considering its adoption within the community, GeoSPARQL is the *de facto* solution to deal with geo-RDF-data. Indeed, multiple evaluators implement GeoSPARQL giving then access to spatial functions for use in SPARQL queries such as finding a distance or computing a convex hull having sets of points or geometries. Therefore, GeoSPARQL provides explicit namespaces (and signatures) for the functions, and the calculations are based on other standards previously defined by the OGC, which should guarantee the use and the results of these functions from one triplestore to an other one. Unfortunately, these supports from the triplestores are not yet complete, as shown by Jovanovik *et al.* in [8]: "none of these widely used RDF storage solutions fully comply with the GeoSPARQL standard". Furthermore, triplestores may come with specific spatial functions like Jena[23]. As a consequence, to guarantee full interoperability of our code blocks and since some of the computations we wanted were not provided by default, we focused on standard SPARQL 1.1 operators.

6 Conclusion

In this article, taking the example of navigating/sailing the Earth, we provide the reader with a list of SPARQL fragments which can be used to compute complex (often tedious and prone-to-error to write) computations on latitude and longitude pairs while combining trigonometric functions. To go further, finer-grained and additional code blocks (with typically more terms in the Taylor series to gain in accuracy) are openly available for copy-pasting purposes on the following repository:

https://github.com/dgraux/Navigating-with-SPARQL

As our expressions make only use of standard SPARQL 1.1, the suggested code-blocks are able to be run on all endpoints. In addition, from a query optimisation point of view, our solution helps to keep as much computation on the endpoint side as possible and therefore gives the opportunity to the endpoints for bespoke optimisation or caching strategies. Overall, while basing this study on navigating features on (lat,lon) pairs, we hope it showcases the potential of SPARQL and that it will foster SPARQL practitioners to rely on endpoints as much as possible.

[23] https://jena.apache.org/documentation/geosparql/index.html

References

1. Battle, R., Kolas, D.: GeoSPARQL: enabling a geospatial semantic web. Semantic Web Journal **3**(4), 355–370 (2011)
2. Bowditch, N.: The american practical navigator, 1995 edition. Deffense Mapping Agency Hydrographic/Topographic Center, Maryland pp. 363–371 (1995)
3. Erling, O., Mikhailov, I.: RDF support in the virtuoso DBMS. In: Networked Knowledge-Networked Media, pp. 7–24. Springer (2009)
4. Graux, D., Jachiet, L., Geneves, P., Layaïda, N.: SPARQLGX: Efficient distributed evaluation of SPARQL with apache spark. In: International Semantic Web Conference. pp. 80–87. Springer (2016)
5. Graux, D., Sejdiu, G., Stadler, C., Napolitano, G., Lehmann, J.: MINDS: a translator to embed mathematical expressions inside SPARQL queries. In: International Conference on Semantic Systems. pp. 104–117. Springer, Cham (2020)
6. Harris, S., Lamb, N., Shadbolt, N.: 4store: The design and implementation of a clustered RDF store. In: 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009). pp. 94–109 (2009)
7. Jena, A.: Apache jena fuseki. The Apache Software Foundation (2014)
8. Jovanovik, M., Homburg, T., Spasić, M.: A GeoSPARQL compliance benchmark. ISPRS International Journal of Geo-Information **10**(7), 487 (2021)
9. Koubarakis, M., Kyzirakos, K.: Modeling and querying metadata in the semantic sensor web: The model stRDF and the query language stSPARQL. In: Extended Semantic Web Conference. pp. 425–439. Springer (2010)
10. Kyzirakos, K., Karpathiotakis, M., Koubarakis, M.: Strabon: A semantic geospatial DBMS. In: International Semantic Web Conference. pp. 295–311. Springer (2012)
11. Manola, F., Miller, E., McBride, B., et al.: RDF primer. W3C recommendation **10**(1-107), 6 (2004)
12. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. The VLDB Journal—The International Journal on Very Large Data Bases **19**(1), 91–113 (2010)
13. Prud'Hommeaux, E., Seaborne, A., et al.: SPARQL query language for RDF. W3C recommendation **15** (2008), www.w3.org/TR/rdf-sparql-query/
14. W3C SPARQL Working Group, et al.: SPARQL 1.1 overview (2013), http://www.w3.org/TR/sparql11-overview/