# Multilateration Index - Masters Thesis

Chip Lynch

6/24/2018

## Contents



# A Multilateration Alternate Coordinate System

**Abstract**

We present an alternative method for pre-processing and storing multilateration point data, particularly for Geospatial points, by storing distances to fixed points rather than coordinates such as Latitude and Longitude. We explore the use of this data to improve query performance for some distance related queries such as nearest neighbor and query-within-radius (i.e. "find all points in a set $P$ within distance $d$ of query point $q$").

Our construction includes storing the distances from fixed points (typically three, as in trilateration) as an alternative to Latitude and Longitude. This effectively creates a coordinate system where the coordinates are the trilateration distances. We explore this alternative coordinate system and the theoretical, technical, and practical implications of using it. Multilateration itself is a common technique in surveying and geolocation

widely used in cartography, surveying, and orienteering, although algorithmic use of these concepts for NN-style problems are scarce. GPS uses the concept of detecting the distance of a device to multiple satellites to determine the location of the device; a concept known as true-range multilateration. However while the approach is common, the distance values from multilateration are typically immediately converted to Latitude/Longitude and then discarded. Here we attempt to use those intermediate distance values to computational benefit. Conceptually, our multilateration construction is applicable to metric spaces in any number of dimensions.

Rather than requiring the complex pre-calculated tree structures, or high cost pre-calculated nearest-neighbor sub-trees (as in FAISS), we rely only on sorted arrays as indexes. This approach also allows for processing computationally intensive distance queries (such as nearest-neighbor) in a way that is easily implemented with data manipulation languages such as SQL, where we see a roughly 10x performance improvement to existing query logic. Other modern nearest-neighbor solutions such as KD-Tree, Ball-Tree, and FAISS require capability beyond that of standard SQL to be performant (such as efficient implementations of tree-structures).

Outside of SQL, our approach shows significant performance gains - 30x performance using the ann-benchmark tool for nearest-neighbors - when the cost of the atomic distance calculation itself is high, such as with geodesic distances on earth using accurate elliptical, rather than spherical or euclidean models of the earth. Of note, our algorithms do not improve on the time-complexity of nearest-neighbor searches; we remain $O(n)$ in the worst case. The main point of improvement is that we can substitute simple subtraction for the distance function itself, after performing $d * n$ distance calculations to create our initial multilateration structure. In effect, remembering that $O(C * f(x)) == O(f(x))$ where $C$ is a constant, our approach focuses on reducing $C$ rather than reducing the time complexity of $f(x)$ itself.

Further, we discuss the problem of "Network Adequacy" common to medical and communications businesses, to analyze questions such as "are at least 90% of patients living within 50 miles of a covered emergency room". This is in fact the class of question that led to the creation of our pre-processing and algorithms, and is a generalization of a class of Nearest-Neighbor problems.

While we focus primarily on geospatial data, potential applications to this approach extend to any distance-measured n-dimensional metric space, and we touch on those (briefly, here, to constrain our scope). For example, we consider applying the technique to Levenshtein distance, MINST datasets via cosine-similarity, and even facial recognition or taxicab systems where distances do not follow the triangle inequality.

# 1 The Trilateration Geospatial Index

## 1.1 Trilateration Index – General Definition

Given an n-dimensional metric space $(M, d)$ (the universe of points $M$ in the space and a distance function $d$ which respects the triangle inequality), a typical point $X$ in the coordinate system will be described by coordinates $x_1, x_2, \ldots, x_n$, which, typically, represents the decomposition of a vector $V$ from an "origin" point $O : 0, 0, \ldots, 0$ to $X$ into orthogonal vectors $0, x1, 0, x2, \ldots, 0, xn$ along each of the n dimensional axes of the space.

The Trilateration of such a point requires $n + 1$ fixed points $F_p$ ($p$ from 1 to $n + 1$), no three of which occupy the same $(n - 1)$-dimensional hyperplane. The Trilateration Coordinate for the point $X$ is then: $t_1, t_2, \ldots, t_{n+1}$ where $t_i$ is the distance (according to $d$) from $X$ to $F_i$ (in units applicable to the system).

## 1.2 Underlying Theory Concepts

The benefit of storing geographic points as a set of trilateration distances rather than latitude and longitude boils down to the simplification of comparing distances between points by shortcutting complex distance queries using simple subtractions. We discuss the math behind the geospatial queries, to exhibit their complexity, and set some theoretical bounds on quick distance calculations using the trilateration index.

### 1.2.1 High Cost of Geospatial Calculations

Calculating the distance between two points around the globe with precision is required for Satellite Communications and Geospatial Positioning Systems (GPS), as well as for ground based surveying and generally all applications requiring precise (sub-meter) measurements accounting for the curvature of the earth.(ASPRS 2015)

#### 1.2.1.1 Haversine

One of the simplest distance calculations between two points on the earth's surface – namely the Haversine Formula (Gade 2010), which dates to the early 1800s– works by assuming the earth is a sphere. The calculation for the distance between two points on the earth, using this formula goes as:

Given the radius of a spherical representation of the earth as $r = 6356.752km$ and the coordinates of two points (latitude, longitude) given by $(\phi_1, \lambda_1)$ and $(\phi_2, \lambda_2)$, the distance $d$ between those points along the surface of the earth is:

$$d = 2r \sin^{-1}(\sqrt{\sin^2(\frac{\phi_2 - \phi_1}{2}) + \cos(\phi_1)\cos(\phi_2)\sin^2(\frac{\lambda_2 - \lambda_1}{2})})$$

Obviously this is somewhat computationally complex, comprising five trigonometric functions, two subtractions and a square root. While it is a closed form solution, it causes an error over long distances of up to 0.3%, which can mean distances are off by up to 3 meters over distances of 1000 kilometers. From the equator to the north pole, which on a sphere is defined as precisely 10,000 km, the actual distance is off by over 2 km, which is a sizeable error for even the most robust applications.

#### 1.2.1.2 Vincenty and Karney's Improvements (Geodesics)

The shortcomings of the spherical calculation was thoroughly discussed by Walter Lambert in 1942.(Lambert 1942) However it wasn't until 1975 that an iterative computational approach came about to give more accurate distance measurements with a model of the earth more consistent with reality. By considering the earth as

an ellipsoid, rather than a sphere, the distance calculations are more complex, but far more precise. Vincenty was able to create an iterative approach accurate down to the millimeter level on an ideal elliptical earth; far more accurate than the Haversine calculations(Vincenty 1975). This algorithm, however, was a series which failed to converge for points at near opposite sides of the earth. Karney was able to improve upon this in 2013 to fix these antipodal non-convergences, and the resulting formulae are now widely available in geospatial software libraries where precision is required (commonly referred to as "Geodesic" distances. (Karney 2013)

To get an idea of the relative complexity, we ran some basic timings using widely available python libraries that perform both calculations. The Haversine is about 22 times faster than Karney's iterative approach. For comparison, we include Euclidean functions, which are of course computationally simple, although their usefulness on curved surfaces are minimal:

Table 1: Timings (seconds) of 5000 Calls to Distance Functions

| title | time | ratio |
|---|---|---|
| Geodesic | 1.2110690 | 575.54843 |
| Haversine | 0.0553729 | 26.31542 |
| Euclidean | 0.0021042 | 1.00000 |

## 1.3 Simple Multilateration Index Operations

Before jumping into Network Adequacy and Nearest Neighbor algorithms let's look at the core usage of the trilateration data structure and its use in simple distance functions.

What we mean by 'simple distance functions' is one of the following primitive functions common to SQL or map related software libraries:

- $D(p, q)$: returns the distance between points p and q
- $Within(d, q, P)$: returns the set of all points in $P$ within distance $d$ of query point $q$
- $AnyWithin(d, q, P)$: returns a boolean result - True if $Within(d, q, P)$ is non-empty; False otherwise

### 1.3.1 Distance Function

How can we use the Trilateration Index ($TI$) to improve the performance of a single distance function $D(p, q)$? In the simplest case, we cannot... the construction of the $TI$ structures requires three distance functions to be calculated each for $p$ and $q$ (to the three fixed reference points).

However, for large datasets with fixed points where many distances need to be calculated between them, particularly if the distance function itself is computationally intensive (such as geospatial distances on an accurate ellipsoid model of earth) (Lambert 1942), we can use the $TI$ structure to create approximate distances, and provide upper and lower bounds on exact values.

For example, let's take our sample data:

| | x | y | d1 | d2 | d3 |
|---|---|---|---|---|---|
| 3 | 57.28944 | 7.950161 | 10.97000 | 76.11526 | 83.99465 |
| 2 | 43.73940 | 43.418684 | 43.87912 | 58.76869 | 48.68666 |
| 7 | 51.14533 | 47.157734 | 47.55169 | 50.76727 | 52.59935 |
| 1 | 58.52396 | 53.516719 | 54.19130 | 40.89001 | 56.13720 |
| 4 | 35.32139 | 58.321179 | 60.21479 | 60.36431 | 33.14272 |
| 5 | 86.12714 | 52.201894 | 63.60981 | 24.08779 | 82.65970 |
| 9 | 85.13531 | 64.090063 | 73.52568 | 15.32024 | 79.41866 |
| 6 | 41.08036 | 78.065907 | 78.73275 | 52.46354 | 34.64818 |
| 8 | 15.42852 | 80.836340 | 88.19694 | 78.32131 | 11.63066 |

| | x | y | d1 | d2 | d3 |
|---|---|---|---|---|---|
| 10 | 99.60833 | 78.055071 | 92.92245 | 10.63544 | 93.22300 |

Here, X and Y are euclidean cartesian coordinates, and d1, d2, d3 are the distances from these points to our three reference points respectively. See 2-D Bounded Example for more details on the construction. Note that in this case we have sorted the data by $d1$ – this is essential, and incurs only $O(n * log(n))$ overhead. This equates to how database indexes or arrays will hold the data in memory.

### 1.3.2 Distance between two points

If we compare points 1 and 2 here (lines 4 and 2 in the $d1$-sorted table), what can we say about those two points' distances without invoking a distance function? If we compare the distances, we can put lower bounds on their proximity using a direct, simple application of the triangle inequality. For example $|d1(P_1) - d1(P_2)| = |54.19130 - 43.87912| = 10.33$ which means the points can be ***no closer than*** 10.33 units to one another. Similarly with d2 and d3, we get $|58.76869 - 40.89001| = 17.88$ and $56.13720 - 48.68666 = 7.45054$. So now, the points can be no closer than 17.88 units, although they are closer relative to the $d1$ and $d3$ points.

### 1.3.3 Within/AnyWithin Distance

It's similarly easy to use this mechanism to approximate answers to "which points are within distance $d$ of query point $Q$?" and, relatedly, "is there at least one point in $P$ within distance $d$ to point $Q$?".

Looking back at our table, let's examine the question "which points are within distance 20 of point 5?". Point 5 has coordinates $(86.12714, 52.201894)$, and is 63.60981 units from $d1$. Since we've stored the list sorted by $d1$, we can instantly limit our search to a sequential walk from points between 43.60981 and 83.60981 – that is, points $(7, 1, 4, 9, 6)$ (excluding 5 itself). This is, immediately, a 50% reduction in the dataset.

While performing the walk, we look for $d2$ between $24.08779 \pm 20$ and $d3$ between $82.65970 \pm 20$. $d2$ rules out points $(7, 4, 6)$ and $d3$ rules out $(1)$, leaving only $(9)$ for consideration. To be completely certain, we can calculate $d = \sqrt{(86.12714 - 85.13531)^2 + (64.090063 - 78.065907)^2} = 14.0109936$ which is, indeed, within 20.

In pseudocode:

```
Within(d, P, TI):
   lowi = lowest i such that TI[i, d1] > P[d1] - d
   highi = highest i such that TI[i, d1] < P[d1] + d
   FOR i FROM lowi to highi:
       if TI[i, dx] between P[dx] - d and P[dx] + d for all x:
          ADD i to CANDIDATES
   FOR c in CANDIDATES:
       if D(c, P) < d:
          ADD c to RESULTS
   RETURN RESULTS
```

If we were answering the "is there at least one point…" version, it would be easy to shortcut the sequential walk when a match is reached.

### 1.3.4 Alternate Order Indexes

For an additional possible performance improvement, we can create alternate indexes which store the data in sorted order along $d2$ and $d3$ (or any/all distances for arbitarty dimensions). We search for the low and high indexes as before, but now we do so along each sorted index (for distances to each reference point). Once

we have the lists of individual candidates from each index, we need to find any point that is common to all candidate lists. In practice we have not seen this behave as effectively as the single-index function, but this seems to come down to the cost of merging n-lists to find common elements.

In pseudocode:

```
WithinMulti(d, P, TI):
   FOR each ref point rx:
      lowi = lowest i such that TI[i, dx] > P[dx] - d
      highi = highest i such that TI[i, dx] < P[dx] + d
      FOR i FROM lowi to highi:
          if TI[i, dx] between P[dx] - d and P[dx] + d:
             ADD i to CANDIDATES[x]
    FOR c in CANDIDATES[1]:
      if c in CANDIDATES[x] for all x:
          ADD c to POSSIBLE
    FOR c in POSSIBLE:
       if D(c, P) < d:
          ADD c to RESULTS
    RETURN RESULTS
```

# 2 Experimentation

## 2.1 Experimental Setup

We plan several experiments for Network Adequacy in SQL databases, and Nearest Neighbor applications using Python.

### 2.1.1 Experiment 1: Python Nearest Neighbors with ANN

We adapt the popular Python package scikit-learn to execute our Trilateration Index and perform nearest-neighbor search. We adapt the ann-benchmarks software, which is designed explicitly for comparing performance of nearest neighbor algorithms(Aumüller, Bernhardsson, and Faithfull 2020), to record results.

### 2.1.2 Experiment 2: SQL Nearest Neighbors

### 2.1.3 Experiment 3: SQL Network Adequacy

## 2.2 Experimental Results

Taa daa

## 2.3 Network Adequacy

The trilateration index was originally designed to improve efficiency of the "network adequacy" problem for health care. Network adequacy is a common legal requirement for medicare or insurance companies with constraints such as:

- 90% of members must live within 50 miles of a covered emergency room
- 80% of female members over the age of 13 must live within 25 miles of a covered OB/GYN
- 80% of members under the age of 16 must live within 25 miles of a covered pediatrician

- etc.

Note that these are all illustrative examples; the real "Medicare Advantage Network Adequacy Criteria Guidance" document for example, is a 75 page document.

Similar requirements, legal or otherwise, show up in cellular network and satellite communication technology (numbers are illustrative):

- Maximize the number of people living within 10 miles of a 5G cell tower
- 100% of all major highways should be within 5 miles of a 4G cell tower
- There must be at least 2 satellites within 200 km of a point 450 km directly above every ground station for satellite network connectivity at any given time
- There must be at least 1 satellite with access to a ground station within 50 km of a point 450 km directly above as many households as possible at any given time

The nearest-neighbor problem was called the "Post-Office Problem" in early incarnations, and the system of post offices lends itself to a similar construction: * Ensure that all US Postal addresses are within range of a post office

and so forth.

It is worth noting that the phrase "Network Adequacy" appears in studies of electric grids bearing a meaning that is NOT related to these distance algorithms. (Mahdavi and Mahdavi 2011; Ahmadi et al. 2019) Satellite "coverage" appears similar at first, and in some cases (like GPS or Satellite Internet) asks a similar question, but often the term "coverage" has a temporal component - for example with satellite imaging - where a satellite must pass over every point it wants to cover *at some point in time*. We do not explore this treatment for those problems with temporal components, although with some works the ideas may be extended there.

### 2.3.1 Formalization of Network Adequacy

We formalize the concept of "Network Adequacy" mathematically:

#### 2.3.1.1 Network Adequacy Definition

Given a non-empty set of points $P$ and a non-empty set of query points $Q$ in a metric space $M$ (where $P \cap Q$ comprises the 'network'), the network is 'adequate' for a distance $d$ and a distance function $D(a, b)$ describing the distance between points $a$ and $b$ for $a \in M$ and $b \in M$ if for every point $q$ (where $q \in Q$) $\exists$ at least one point $p$ ($p \in P$) $\ni D(p, q) <= d$. Otherwise the network is 'inadequate'.

#### 2.3.1.2 Network Adequacy Threshold Definition

We can generalize this slightly more by describing a network as 'adequate with threshold $T$' by introducing a percent $T$ ($0 <= T <= 1$) such that the same network is adequate if for at least $T * |Q|$ (or $T$ percent of points in $Q$) there exists at least one point $p \ni D(p, q) <= d$.

In this case, if $T == 1$ we have the original case. If $T == 0$ we have a trivial case where the network is always adequate (even if $Q$ and/or $P$ are empty, which is generally disallowed).

### 2.3.2 Existing solutions

We can find no literature where this topic is solved in a particular algorighmic way. There are numerous discussions in health care about satisfying network adequacy, but more as policy or health care topics than as computational approaches. (Wishner and Marks 2017),(Mahdavi and Mahdavi 2011)

In general, it appears that most practical solutions are done in SQL databases which are commonly the source of member and provider data for health care datasets. Still, there is little published here; this information is anecdotal based on the author's personal direct knowledge and informal research.

Satellite and cellular network discussions of this problem appear to be proprietary, but again anecdotally, appear to simply apply common Nearest-Neighbor

Where we can find references to actual applications, the implemented solutions tend to be iterative, exhaustive implementations of existing Nearest-Neighbor algorithms. That is, for each point $q$, find the nearest point $p$ and if $D(p,q) < d$ count it as conforming, otherwise count it as non-conforming. We then calculate the ratio of $r = \frac{conforming}{|Q|}$ to determing whether $r >= T$ or not.

If we set $m = |Q|$ and $n = |P|$, and if we use a Nearest-Neighbor algorithm with $O(n\ log\ n)$ then the time complexity for Network Adequacy becomes $O(mn\ log\ n)$.

In the worst-case, we cannot improve on this mathematically, but we can introduce what we believe are novel algorithms, based on the Trilateration Index, which execute efficiently compared to this iterative approach, by deeply reducing the search space and number of times the distance functions must be called in typical real-world cases. Also, trivially, it is unnecessary to find the nearest point $p$, merely prove that at least one such point exists (or none exists) for a given $q$.

## 2.4 Nearest Neighbor

The nearest neighbor ($NN$) problem should need no introduction. For our perspective, we talk about NN and k-NN as:

Given a non-empty set of points $P$, a non-empty set of query points $Q$ in a metric space $M$, and a distance function $D(a,b)$ describing the distance between points $a$ and $b$ for $a \in M$ and $b \in M$, the "Nearest Neighbor" of a given point $q \in Q$ is the point $p \in P$ such that $D(p,q)$ is the lowest value of $D(p',q)$ over all points $p' \in P$ (i.e. $D(p,q) < D(p',q) \forall p'$ with $p \neq p'$.

Note that it is possible that such a point does not exist if there are multiple points with the same lowest distance; we do not explore that situation here, as it does not affect our examination.

The k-nearest neighbors ($kNN$) of a given point $q$ as above is the list of $k$ points $R = p_1..p_k \in P$ such that $D(p_k,q)$ is the lowest value of $D(p,q)$ over all $p \in P$ such that $D(p',q)$ for $p' \neq p$ and $p' \notin R$. It should be evident that $NN = kNN$ when $k = 1$.

An approximate nearest neighbor ($ANN$) algorithm is one which will provide $k$ points $R' = p_1..p_k \in P$, however it does not guarantee that there exists no point in $P \notin R'$ closer than any point in $R'$. Some formulations require that, if there is such a point $p'$, that it cannot be more than some $\epsilon$ farther from $q$ than any point $p_i \in R'$. In general, $ANN$s are used when we can get a 'close enough' solution algorithmically faster than a perfect $kNN$ solution. For our purposes, we largely ignore $ANN$s except for their historical value, as our construction did not yield algorithms that exhibited this beneficial tradeoff. See the "TrilatApprox" algorithm section for some more discussion.

### 2.4.1 A History of k-NN Solving Algorithms

#### 2.4.1.1 Brute-Force

The naive approach to solving k-nn is a brute-force algorithm, iterating over every point $pinP$ and keeping track of the lowest k distances.

#### 2.4.1.2 Space Partitioning Trees

Space paritioning trees use a trie to arrange points from $p$ into groups with a hierarchical search structure, such that, generally, points which are close to one another exist in nearby hierarchies. The $k-d$ tree was

described in 1975.(Bentley 1975) This partitions a space by dividing the underlying points at the median point along one of the dimensional axes, recursively, resulting in a searchable trie. An adaptation of this - the Ball-Tree - partitions the space into hyperspheres, rather than along dimensional axes.(Liu, Moore, and Gray 2006)

These are straightforward structures that are easy to describe and implement.

#### 2.4.1.3 Locality Sensitive Hashing

LSH stands somewhat alone, as it is not literally space partitioning with strict divisions. LSH relies on creating a hash function that hashes points into bins with a property that two points with the same hash have a high likelihood of being nearer to each other than points with different hash values.(Indyk and Motwani 1998)

#### 2.4.1.4 Graph Based Search

More recent algorithms, such as Facebook Research's FAISS, follow a graph based search structure.(Johnson, Douze, and Jégou 2017)

A good overview of this approach was available from Liudmila Prokhorenkova: "Recently, graph-based approaches were shown to demonstrate superior performance over other types of algorithms in many large-scale applications of NNS (Aumüller, Bernhardsson, and Faithfull 2020). Most graph-based methods are based on constructing a nearest neighbor graph (or its approximation), where nodes correspond to the elements of D, and each node is connected to its nearest neighbors by directed edges.(Dong, Charikar, and Li 2011) Then, for a given query q, one first takes an element in D (either random or fixed predefined) and makes greedy steps towards q on the graph: at each step, all neighbors of a current node are evaluated, and the one closest to q is chosen."(Prokhorenkova 2019)

The construction costs of these structures can be very high. A Brute Force construction of a k-Nearest Neighbor Graph ($kNNG$) has time complexity $O(n^2)$ which is of course completely untenable for large data sets. Approaches exist to improve upon this, including improvements resulting in approximate results, but this class still tends to trade the highest construction cost for some of the fastest query times in high dimensions.(Dong, Charikar, and Li 2011), (Prokhorenkova 2019)

### 2.4.2 Comparing Algorithms

Training Time Complexity Memory Space Prediction Time Complexity Insertion/Move Complexity

## 2.5 Multilateration

## 2.6 Kentucky 2d Geodesic 150000 points:

## 2.7 Geodesic query points:

We have created a dataset specifically to test Geodesic queries. The dataset is a synthetic set of 150,000 geospatial points spread across and near Kentucky with roughly the distribution of the population.

As hoped, our Trilateration algorithm really shines when applying the complex but accurate geodesic distance function. The Trilateration algorithm is over 30 times faster than the next best candidate (the Ball Tree algorithm with leaf_size=10). The Brute Force algorithm is unbearably slow here, which is expected since it should be calling the expensive distance function more than any other algorithm in the usual case.

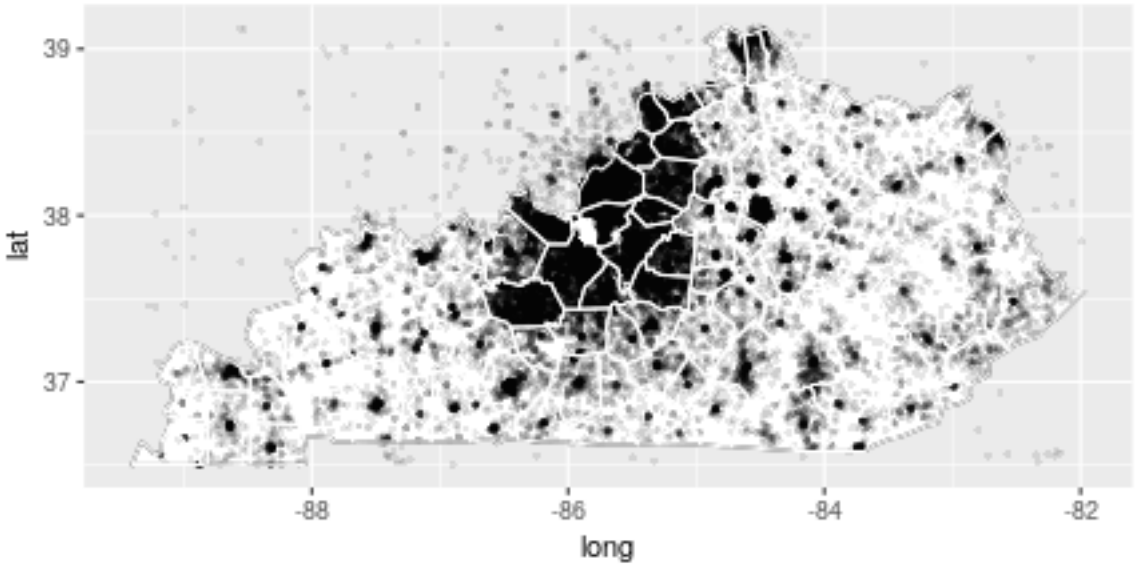



Figure 1: Geodesic Sample Data

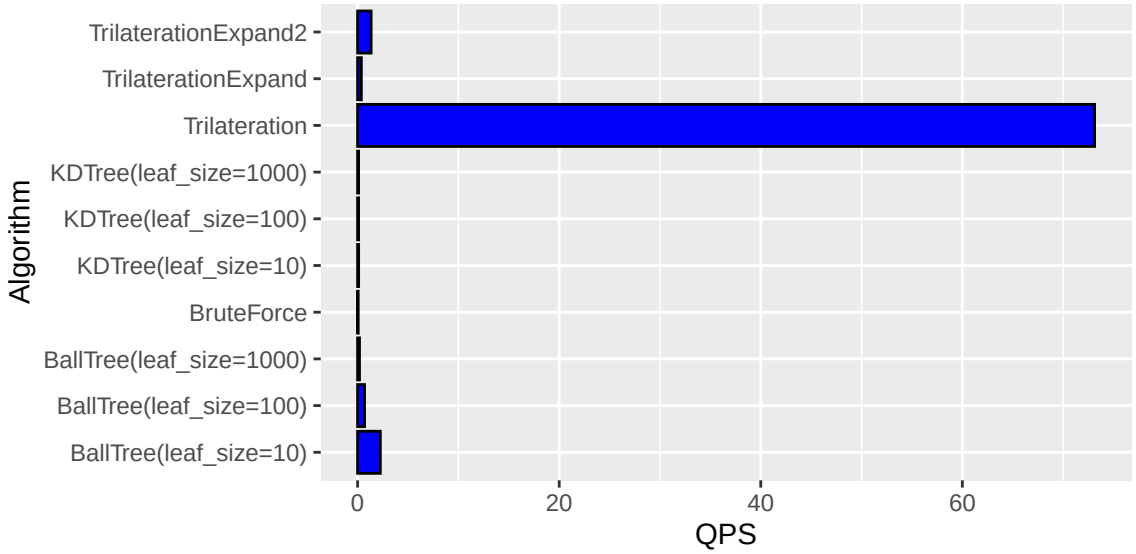



Figure 2: Queries Per Second for Geodesic (Q=150,000)

## 2.8 Random 20-dimension euclidean distance:

The ann-benchmark tool includes support for a randomly generated 20-dimension euclidean dataset, which is one of its most basic tests. We note that our performance here is abysmal for the initial "Trilateration" algorithm, even failing to beat the brute force approach. This seems to be due to the overhead we incur determining which candidates to test next. Remember that we traded a number of subtractions and some array navigation in exchange for fewer distance function calls. In this case, when the distance function itself is extremely fast, that overhead is a net loss.

It is worth noting that we tuned the two expansion based algorithms here as well. "TrialterationExpand" performs far better than the stock Trilateration algorithm, but still just below the Brute Force algorithm. The "TrilaterationExpand2" algorithm, however, is actually competitive here, more than doubling the queries per second of the Brute Force approach, and reaching more than 60% as fast as some tree algorithms.

For Euclidean distances, however, we cannot recommend our algorithms against competitors.

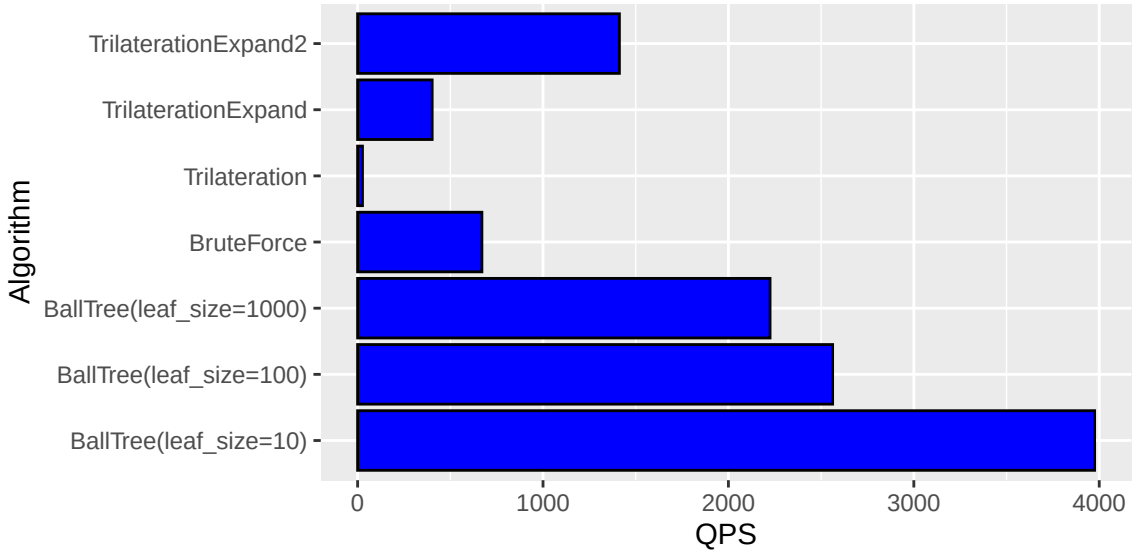



Figure 3: Queries Per Second and Time to Build Indexes for 20-dimension Euclidean

## 2.9 Glove 25-dimension Angular distance

The GloVe ("Global Vectors for Word Representation") dataset "is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space" according to the authors. (Pennington, Socher, and Manning 2014)

It is one of the built-in datasets in the ann-benchmark tool. Under the hood, this isusing a euclidean distance function, once the angular coordinates are transformed, so the relative performance is similar to the euclidean dataset.

Of note, we include only results that were full (not approximate) nearest neighbor solutions.

These results include the addition of the FAISS algorithm, one of the graph index based NN solvers which came out of Facebook Research in recent years.(Johnson, Douze, and Jégou 2017) On these datasets, FAISS is a beast, but unfortunately it is a very highly tuned GPU-based implementation which makes it difficult to adapt to unsupported distance functions, such as the Geodesic we target with Trilateration. We leave it here for information, but are unable to compare it on our core task.

Note that our performance has suffered again similarly to with the random euclidean dataset. Trilateration is extremely slow; as is TrilaterationExpand. TrilaterationExpand2 beats BruteForce and is somewhat shy of the Tree-based algorithms. But we are not competitive in this space.

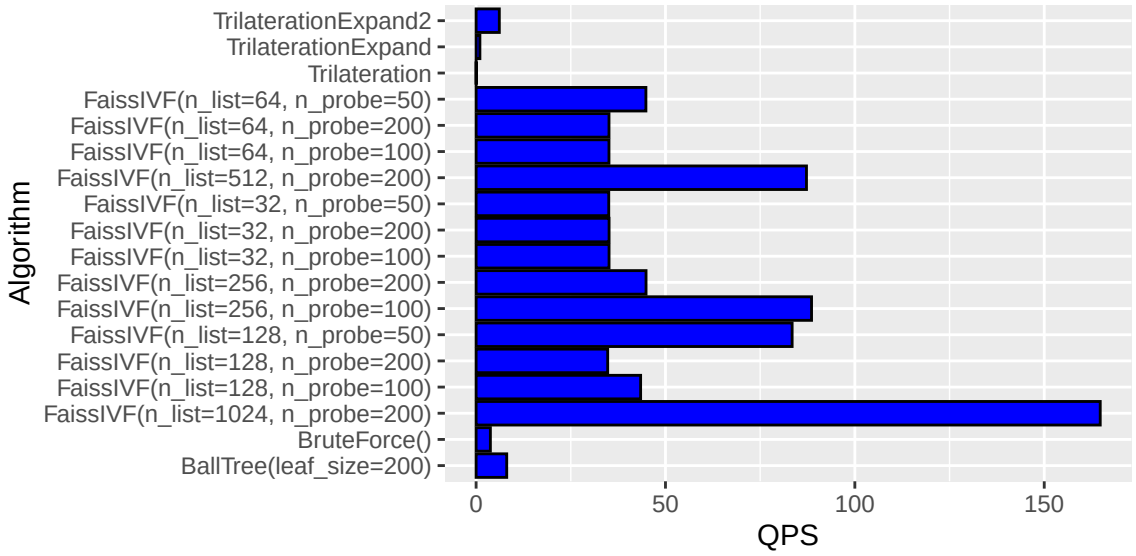



Figure 4: Queries Per Second and Time to Build Indexes for glove-25-angular

## 2.10 Review of Current Literature

In general, we split our current literature review into two major sections: 1. Geospatial considerations and 2. Nearest Neighbor Algorithms, and a third minor section for ancillary references outside those two areas:

### 2.10.1 Geospatial References

#### 2.10.1.1 Haversine

#### 2.10.1.2 Vincenty's Formula

Vincenty's Formula is a common non-spherical numeric solution to Earth-shaped ellipsoidal distance calculations [https://arxiv.org/pdf/1109.4448.pdf] however it "fails to converge for nearly antipodal points" #### Karney's Formula - https://arxiv.org/abs/1109.4448 Improvements upon Vincenty's Formula have already been implemented in Python (https://pypi.org/project/geographiclib/), which we use in our Python implementations.

---

## 2.11 2-D Bounded Example

Consider a 2-dimensional grid – a flattened map, a video game map, or any mathematical $x - y$ coordinate grid with boundaries. WOLOG in this example consider the two-dimensional Euclidean space $M = \mathbb{R}^{\nVdash}$ and bounded by $x, y \, \epsilon \, \{0..100\}$. Also, let us use the standard Euclidean distance function for $d$. This is, trivially, a valid metric space.

Since the space has dimension $n = 2$, we need 3 fixed points $F_p$. While the Geospatial example on Earth has a specific prescription for the fixed points, an arbitrary space does not. We therefore prescribe the following construction for bounded spaces:

Construct a circle (hypersphere for other dimensions) with the largest area inscribable in the space. In this example, that will be the circle centered at $(50, 50)$ with radius $r = 50$.

Select the point at which the circle touches the boundary at the first dimension (for spaces with uneven boundary ratios, select the point at which the circle touches the earliest boundary $x_i$). Such a point is guaranteed to exist since the circle is largest (if it does not, then the circle can be expanded since there is space between every point on the circle and an axis, and it is not a largest possible circle).

From this point, create a regular $n + 1$-gon (triangle here) which touches the circle at $n + 1$ points. These are the points we will use as $F_p$. They are, by construction, not all co-linear (or in general do not all exist on the same $n$-dimensional hyperplane) satisfying our requirement [proof].

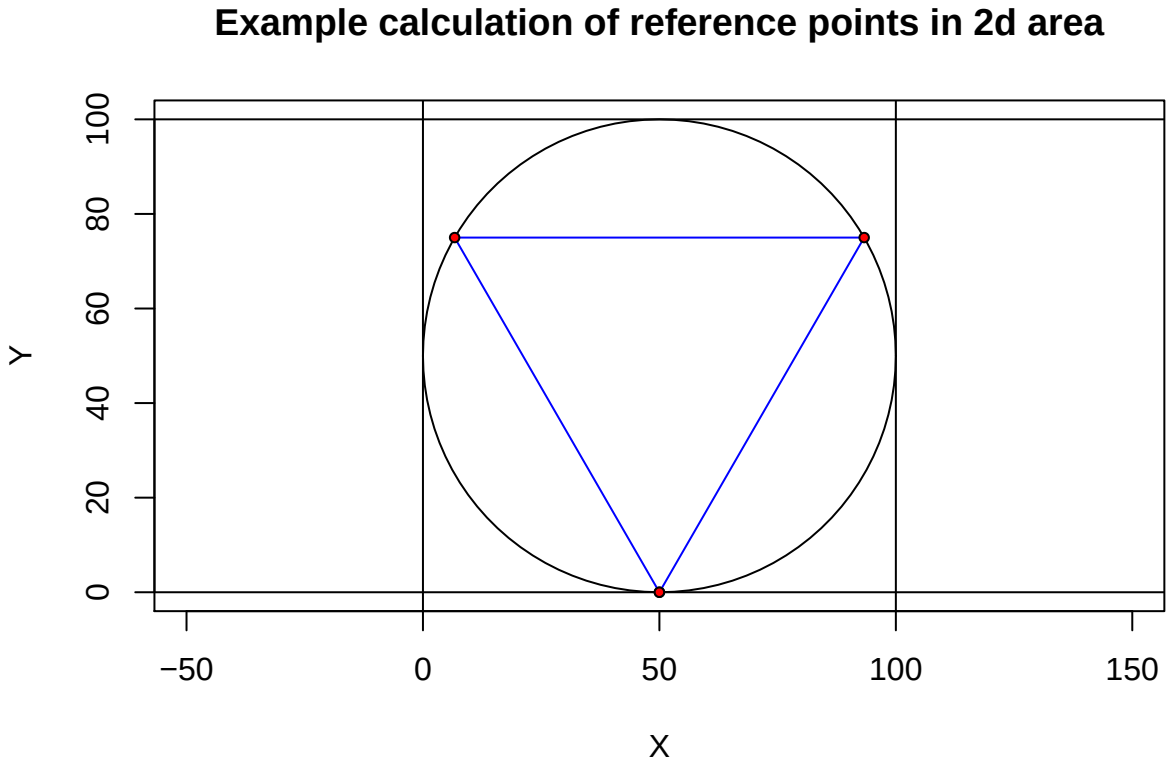
The point $y = 0$, $x = 50$ is the first point of the equilateral triangle. The slope of the triangle's line is $tan(\frac{pi}{3})$, so setting the equation of the circle:

$(x - 50)^2 + (y - 50)^2 = 50^2$ equal to the lines: $y = tan(\frac{\pi}{3})(x - 50)$ gives $x = 25(2 + \sqrt{3})$ on the right and $y = tan(\frac{-\pi}{3})(x - 50)$ gives $x = -25(\sqrt{3} - 2)$ on the left, and of course the original $(0, 50)$ point. Applying $x$ to our earlier equations for $y$ we get a final set of three points:

$$F_1 = (x = 50, y = 0)$$

$$F_2 = (x = 25(2 + \sqrt{3}), y = tan(\frac{\pi}{3})((25(2 + \sqrt{3})) - 50)$$

$$F_3 = (x = -25(\sqrt{3} - 2), y = tan(\frac{-\pi}{3})((-25(\sqrt{3} - 2)) - 50)$$




Remember, any three non-colinear points will do, but this construction spaces them fairly evenly throughout the space, which may be beneficial later* [Add section (reference) with discussions of precision and examples where reference points are very near one another].

The trilateration of any given point X in the space, now, is given by:

$$T(X) = d(F_1, X), d(F_2, X), d(F_3, X)$$

That is, the set of (three) distances $d$ from $X$ to $F_1$, $F_2$, and $F_3$ respectively.

#### 2.11.0.1 10 Random Points

As a quick example of the trilateration calculations, we use a basic collection of 10 data points:

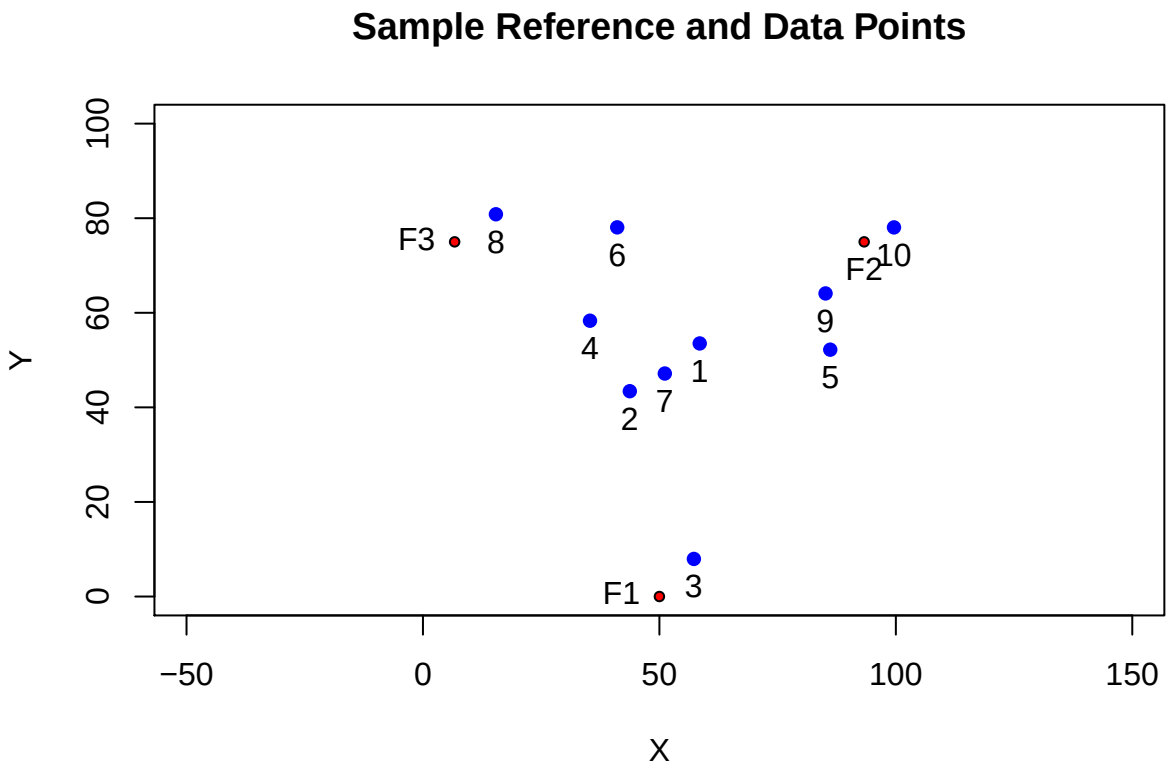
```
##           x        y
## 1  58.52396 53.516719
## 2  43.73940 43.418684
## 3  57.28944  7.950161
## 4  35.32139 58.321179
## 5  86.12714 52.201894
## 6  41.08036 78.065907
## 7  51.14533 47.157734
## 8  15.42852 80.836340
## 9  85.13531 64.090063
## 10 99.60833 78.055071
```

The trilateration of those points, that is, the three points $d_1, d_2, d_3 = d(F_1, X), d(F_2, X), d(F_3, X)$ are (next to the respective $x_n$):

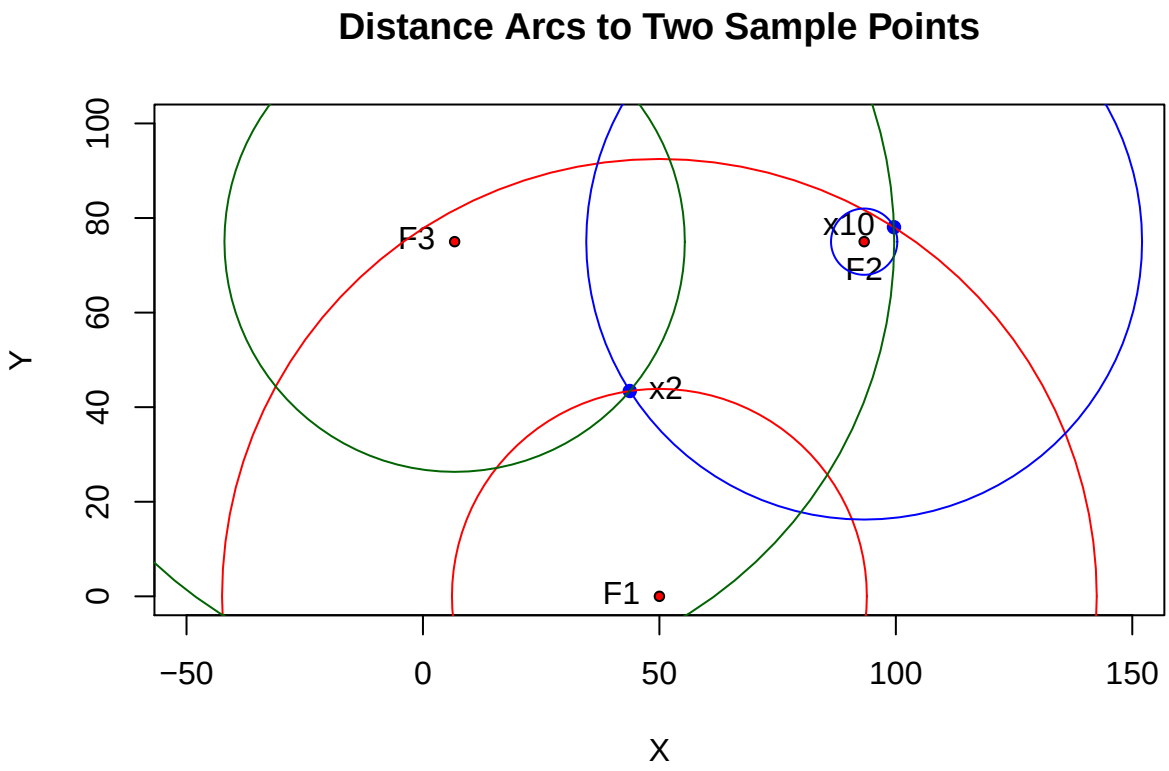
```
##           x         y       d1        d2       d3
## 1  58.52396 53.516719 54.19130 40.877779 56.10157
## 2  43.73940 43.418684 43.86772 58.768687 48.67639
## 3  57.28944  7.950161 10.78615 76.108693 83.99465
## 4  35.32139 58.321179 60.14002 60.331164 33.12763
## 5  86.12714 52.201894 63.48392 23.900246 82.63550
## 6  41.08036 78.065907 78.57382 52.310835 34.51806
## 7  51.14533 47.157734 47.17164 50.520446 52.44704
## 8  15.42852 80.836340 87.91872 78.091149 10.50105
## 9  85.13531 64.090063 73.08916 13.627535 79.19169
## 10 99.60833 78.055071 92.48557  7.008032 92.95982
```

Note that we do not need to continue to store the original latitude and longitude. We can convert the three $d_n$ distances back to Latitude and Longitude within some $\epsilon$ based on the available precision. Geospatial coordinates in Latitude and Longitude with six digits of precision are accurate to within $< 1$ *meter*, and 8 digits is accuract to within $< 1$ *centimeter*, although this varies based on the latitude and longitude itself; latitudes closer to the equator are less accurate than those at the poles. The distance values $d_x$ are more predictable, since they measure distances directly. While the units in this sample are arbitrary, $F(x)$ in a real geospatial example could be in kilometers, so three decimal digits would precisely relate to 1 *meter*, and so on. This is one reason that we will later examine using the trilateration values as an outright replacement for Longitude and Latitide, and this feature is important when considering storage requirements for this data in large real-world database applications.

For now, continuing with the example, those 10 points are shown here in blue with the three reference points $F_1, F_2, F_3$ in red:



To help understand the above values, the following chart shows the distances for points $x_2$ and $x_10$ above. Specifically, the distances $d_1$ from point $F_1$ are shown as arcs in red, the distances $d_2$ from point $F_2$ in blue, and $d_3$ from point $F_3$ in green.




#### 2.11.0.2 Use of trilateration as an index for nearest-neighbor

One of our expected benefits of this approach is an improvement in algorithms like nearest-neighbor search.

### 2.11.1 Geospatial Example

Applying this to real sample points; let the following be the initial reference points on the globe:

Point 1: $90.000000, 0.000000$ (The geographic north pole)

Point 2: $38.260000, -85.760000$ (Louisville, KY on the Ohio River)

Point 3: $-19.22000, 159.93000$ (Sandy Island, New Caledonia)

Optional Point 4: $-9.42000, 46.33000$ (Aldabra)

Optional Point 5: $-48.87000, -123.39000$ (Point Nemo)

Note that the reference points are defined precisely, as exact latitude and longitude to stated decimals (all remaining decimal points are 0). This is to avoid confusion, and why the derivation of the points is immaterial (Point Nemo, for example is actually at a nearby location requiring more than two digits of precision).

Only three points are required for trilateration (literally; thus the "tri" prefix of the term), but we include 5 points to explore the pros and cons of n-fold geodistance indexing for higher values of n.

## 2.12 Theoretical Discussion

### 2.12.1 Theoretical benefits:

**Precision**: Queries are not constrained by precision choices dictated by the index, as can be the case in Grid Indexes and similar R-tree indexes. R-tree indexes improve upon naïve Grid Indexes in this area, by allowing the data to dictate the size of individual grid elements, and even Grid Indexes are normally tunable to specific data requirements. Still, this involves analysis of the data ahead of time for optimal sizing, and causes resistance to changes in the data.

**Distributed Computing**: Trilateration distances can be used as hash values, compatible with distributed computing (I.e. MongoDB shards or Teradata AMP Hashes).

**Geohashing**: Trilateration distances can be used as the basis for Geohashes, which improve somewhat on Latitude/Longitude geohashes in that distances between similar geohashes are more consistent in their proximity.

**Bounding Bands**: The intersection of Bounding Bands create effective metaphors to bounding boxes, without having to artificially nest or constrain them, nor build them in advance.

**Readily Indexed (B-Tree compatible)**: Trilateration distances can be stored in traditional B-Tree indexes, rather than R-tree indexes, which can improve the sorting, merging, updating, and other functions performed on the data.

**Fault Tolerant**: This coordinate system is somewhat self-checking, in that many sets of coordinates that are individually within the correct bounds, cannot be real, and can therefore be identified as data quality issues. For example, a point cannot be 5 kilometers from the north pole (fixed point F1) and 5 kilometers from Louisville, KY (fixed point F2) at the same time. A point stored with those distances could be easily identified as invalid.

Theoretical shortcomings:

**Index Build Cost**: Up front calculation of each trilateration is expensive, when translating from standard coordinates. Each point requires three (at least) distance calculations from fixed points and the sorting of the resulting three lists of distances. This results in `O(n*logn)` just to set up the index.

*This could be mitigated by upgrading sensor devices and pushing the calculations back to the data acquisition step, in much the way that Latitude and Longitude are now trivial to calculate in practice by use of GPS devices. Also, we briefly discuss how GPS direct measurements (prior to converstion to Lat/Long) may be useful in constructing trilateration values.

**Storage**: The storing of three distances (32- or 64- bits per distance) is potentially a sizeable percent increase in storage requirement from storing only Latitude/Longitude and some R-Tree or similar index structure.

*Note that if the distances are stored instead of the Lat/Long, rather than in addition to them, storage need not increase.

**Projection-Bound**: The up-front distance calculations means that transforming from one spatial reference system (I.e. map projection – geodetic – get references to be specific) to another requires costly recalculations bearing no benefit from the calculation. For example a distance on a spherical projection of the earth between a given lat/long combination will be different than the distance calculated on the earth according to the standard WGS84 calculations).

*This said, we expect in most real-world situations, cross-geodetic comparisons are rare.

**Difficult Bounding Band Intersection**: Bounding Bands intersect in odd shapes, which, particularly on ellipsoids, but even on 2D grids, are difficult to describe mathematically. Bounding boxes on the other hand, while they distort on ellipsoids, are still easily understandable as rectangles.

Figure 1 - An example problem with radio towers R1, R2, and R3, and various receivers. Dashed lines represent the bounding bands with +/- a small distance from a given receiver (center black circle)

Figure 2 - A close look at the intersection of three bounding bands limiting an index search around a point with a search radius giving the circle in A. Note that the area B is an intersection of two of the three bands. Area C is the intersection of all three.

# 3 Appendixy stuff

## 3.1 Code and datasets

All code including markdown and custom sample data files to generate this document are available here: https://github.com/chipmonkey/TrilaterationIndex

A custom fork and branch of scikit-learn which includes the Python/Cython implementation of our Multilateration Nearest-Neighbor and related distance functions is available here: https://github.com/chipmonkey/scikit-learn/tree/feature/chip-test-index

A custom fork and branch of the ANN Benchmarks code which includes hooks to our Multilateration NN implementation (from the custom scikit-learn fork) is available here: https://github.com/chipmonkey/ann-benchmarks

## 3.2 The rest of this appendix needs review

Alternate Database Indexes

References (I need to finish reading and digesting these):

https://www.sciencedirect.com/science/article/pii/S1000936118301973

http://ieeexplore.ieee.org/document/5437947/

https://prezi.com/chnisgybkshy/gps-trilateration/

https://www.maa.org/sites/default/files/pdf/cms_upload/Thompson07734.pdf

https://www.researchgate.net/publication/2689264_The_X-tree_An_Index_Structure_for_High-Dimensional_Data

http://repository.cmu.edu/cgi/viewcontent.cgi?article=1577&context=compsci

https://link.springer.com/chapter/10.1007/10849171_83

http://ieeexplore.ieee.org.echo.louisville.edu/document/7830628/

https://link-springer-com.echo.louisville.edu/article/10.1007%2Fs11222-013-9422-4

https://boundlessgeo.com/2012/07/making-geography-faster/

http://ieeexplore.ieee.org.echo.louisville.edu/document/6045057/

http://www.sciencedirect.com.echo.louisville.edu/science/article/pii/S002002550900499X?_rdoc=1&_fmt=high&_origin=gateway&_docanchor=&md5=b8429449ccfc9c30159a5f9aeaa92ffb&ccp=y

https://en.wikipedia.org/wiki/K-d_tree

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.219.7269&rep=rep1&type=pdf

http://www.scholarpedia.org/article/B-tree_and_UB-tree

## Bibliography


Ahmadi, Seyed Alireza, Vahid Vahidinasab, Mohammad Sadegh Ghazizadeh, Kamyar Mehran, Damian Giaouris, and Phil Taylor. 2019. "Co-Optimising Distribution Network Adequacy and Security by Simultaneous Utilisation of Network Reconfiguration and Distributed Energy Resources." *IET Generation, Transmission and Distribution* 13 (20). https://doi.org/10.1049/iet-gtd.2019.0824.

ASPRS. 2015. "ASPRS Positional Accuracy Standards for Digital Geospatial Data." *Photogrammetric Engineering & Remote Sensing* 81 (3). American Society for Photogrammetry; Remote Sensing: 1–26. https://doi.org/10.14358/pers.81.3.a1-a26.

Aumüller, Martin, Erik Bernhardsson, and Alexander Faithfull. 2020. "ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms." *Information Systems* 87: 101374. https://doi.org/https://doi.org/10.1016/j.is.2019.02.006.

Bentley, Jon Louis. 1975. "Multidimensional Binary Search Trees Used for Associative Searching." *Communications of the ACM* 18 (9). https://doi.org/10.1145/361002.361007.

Dong, Wei, Moses Charikar, and Kai Li. 2011. "Efficient K-Nearest Neighbor Graph Construction for Generic Similarity Measures." In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011*. https://doi.org/10.1145/1963405.1963487.

Gade, Kenneth. 2010. "A Non-Singular Horizontal Position Representation." *Journal of Navigation* 63 (3). Cambridge University Press: 395–417. https://doi.org/10.1017/S0373463309990415.

Indyk, Piotr, and Rajeev Motwani. 1998. "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality." In *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*. https://doi.org/10.4086/toc.2012.v008a014.

Johnson, Jeff, Matthijs Douze, and Hervé Jégou. 2017. "Billion-Scale Similarity Search with Gpus." *arXiv Preprint arXiv:1702.08734*.

Karney, Charles F.F. 2013. "Algorithms for Geodesics." *Journal of Geodesy* 87 (1). https://doi.org/10.1007/s00190-012-0578-z.

Lambert, W. D. 1942. "The Distance Between Two Widely Separated Points on the Surface of the Earth." *J. Washington Academy of Sciences*, no. 32 (5).

Liu, Ting, Andrew W. Moore, and Alexander Gray. 2006. "New Algorithms for Efficient High-Dimensional Nonparametric Classification." *Journal of Machine Learning Research* 7. https://doi.org/10.7551/mitpress/4908.003.0008.

Mahdavi, Meisam, and Elham Mahdavi. 2011. "Transmission Expansion Planning Considering Network Adequacy and Investment Cost Limitation Using Genetic Algorithm." *World Academy of Science, Engineering and Technology* 80. https://doi.org/10.5281/zenodo.1086125.

Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. 2014. "GloVe: Global Vectors for Word Representation." In *Empirical Methods in Natural Language Processing (Emnlp)*, 1532–43. http://www.aclweb.org/anthology/D14-1162.

Prokhorenkova, Liudmila. 2019. "Graph-Based Nearest Neighbor Search: From Practice to Theory." *arXiv*.

Vincenty, T. 1975. "DIRECT and Inverse Solutions of Geodesics on the Ellipsoid with Application of Nested Equations." *Survey Review* 23 (176). Taylor & Francis: 88–93. https://doi.org/10.1179/sre.1975.23.176.88.

Wishner, Jane B, and Jeremy Marks. 2017. "Ensuring Compliance with Network Adequacy Standards: Lessons from Four States." *Robert Wood Johnson Foundation*, no. March.