

Doing Distance Calculations in PHP

By Dr. Tarique Sani

Distance calculations using latitude and longitude coordinates lends itself nicely to applications which rely upon finding points on Earth within a specific radius. Here we'll have a look at the code and the math needed to make this happen in a predictable, accurate manner

Introduction

It all started as a nagging thought, the kind of thought that disturbs the rhythm of that nice song you hum all day long. I just needed my program to find the distance between any two cities in the US and I was being asked to pay several hundred dollars for it!

It just wouldn't do! I searched the web for an easy way out but apparently nobody had thought of solving the problem for me. So I noted the facts that I would need to know to find the distance between City A and City B:

1. Latitude and Longitude of City A
2. Latitude and Longitude of City B
3. The mathematical formula needed to make sense out of points 1 and 2

Oh! if you are wondering what on earth (no pun intended) would any one use such information for, here is a short list:

1. Locate the nearest dealer for your car, computer, lawn mower
2. Locate a property near where you would want
3. Find an escort service near where you live
4. And fill in the blank with any thing' for which latitude and longitude are known

As you may recall, latitude lines are the horizontal lines on a map and mark the position of a place north or south of the equator. Similarly, longitude lines are the vertical lines on the map and mark the location of a place east or west of Greenwich, UK. Using latitude and longitude coordinates, one can pin point the location of any place on earth.

I hope the cartographers among you will forgive me for this over simplification of an intricate system of angles measured from the center of the earth. This explanation will have to suffice, and it works nicely for the purpose of this article.

To my pleasure I found that latitude and longitude data for all zip codes in the US was available at <http://www.census.gov/tiger/tms/gazetteer/> - though in a format slightly different from what I wanted. Nevertheless, the data is available, and what's more, I don't have to maintain it!

REQUIREMENTS

PHP Version: **4.04**

O/S: **Any**

Additional Software: **An RDBMS**

Code Directory: **distance**

Earth is flat?

I never was good in geometry, so all of my presumptions on the first set of calculations were based on the premise that City A and City B are located on a flat surface. Recalling high school geometry, I remembered that the distance between any two given points can be found by using the formula

$$\text{distance} = \text{sqrt} ((x_2 - x_1)^2 + (y_2 - y_1)^2)$$

So, in this case, latitudes provided the x co-ordinates and longitudes provided the y co-ordinates. This still did not give distance in miles, so, delving a bit into geography, I figured out that the linear distance covered by 1 degree latitude or longitude is approximately 69.1 miles. I'll explain how this works:

You can find the circumference of a circle by using the formula

$$\text{circumference} = 2 * \text{PI} * r$$

“Using latitude and longitude coordinates, one can pin point the location of any place on Earth. ”

The radius of earth is assumed to be 3963 miles. That makes the circumference of the earth 24887.6 miles. Since a circle has 360 degrees we divide the circumference by 360 and get 69.1323, which we round off as 69.1.

Thus the formula for finding distance now became

$$\text{distance} = \text{sqrt} ((69.1 * (\text{lat}_2 - \text{lat}_1))^2 + (69.1 * (\text{lon}_2 - \text{lon}_1))^2)$$

Great! The rest appeared simple, but read on.

Earth is spherical!

There were two flaws in my line of thinking, the first and obvious one was earth is not flat, the second was not so obvious and was the one which increased the inaccuracy most. As you go towards the poles, the longitude lines converge - implying that the distance between two longitudinal points decreases as one moves away from the equator. Naturally the results I got were not very accurate, as I had not taken this into account when I developed my formula.

Since I was working only with data from the US, and as the US is situated to the north of equator, the lines of longitude are closer to each other than they are at the equator. A bit of experimentation showed that, for the US, if I changed the 69.1 in longitude side to 53. The formula in question would look like:

$$\text{distance} = \text{sqrt} ((69.1 * (\text{lat}_2 - \text{lat}_1))^2 + (53 * (\text{lon}_2 - \text{lon}_1))^2)$$

The accuracy of results increased a lot but was still not good enough for my program. I also felt that if this code could be used for any location on earth it would be “Oh so very great”. I started to think in terms of finding a fractional number by which I could decrease the multiplying factor (69.1) to yield the correct results.



creative affordable solutions

Professional Hosting

Designed with the PHP Developer & Entrepreneurs in mind.

PHP4 MySQL FreeBSD 128 Bit SSL Web Based Control Panel

Industrial Strength Web Servers - 2000Mbit of available bandwidth - Reliability & Performance Guaranteed - Unlimited Technical Support

30 Day Free Trial

All for just US\$15.00 per month

Paid monthly via PayPal.com, No Setup Fee, No Minimum Contract.

Check out the full details of our PHP Architect offer at <http://www.creationinternet.co.uk/offers>

I will spare you the description of the agonizing process I went through, but it just could not be done. It was then that a mathematically inclined friend of mine told me that I might find a solution using trigonometry. At this point I would have surely given up had he also not told me what to do. He elucidated that the accuracy can be further increased by adding the cosine function thus:

```
distance = sqrt (      (69.1 * (lat2 - lat1))2 +
                      (69.1 * (lon2 - lon1) *
                      cos(lat1/57.3))2
                )
```

Here 57.3 is actually 180/pi and is required to convert latitude and longitude from degrees to radians. He also gave me a couple of whacks on the head to make me remember my basic trigonometry. Since I am not a sadist I have reproduced it for everyone's benefit in the sidebar.

This latest formula produced acceptable results in

most cases. However, the formula still assumed earth was a flat surface. Still further through this mathematical maze I deduced that if exact distances were to be calculated then I had to turn to the world of spherical geometry and apply what is popularly known as the *Great Circle Distance* formula. A-Ha! This formula is used for finding the distance between two points on the surface of a sphere.

Without going into unnecessary details of the formula, It states that:

```
distance = acos(      sin(lat1) * sin(lat2) +
                      cos(lat1) * cos(lat2) *
                      cos(lon2-lon1) ) * r
```

All of the latitude and longitude values have been converted to radians and "r" is the radius of earth in miles. One might argue that earth is not a perfect sphere, but rather it is squashed along the north-south axis. The differences are negligible for most purposes, though the point deserves being made. On with the show!

Listing 2

```
1  <?php
2
3  function findDistance($zipOne,$zipTwo)
4  {
5      $query = "SELECT * FROM zipData WHERE zipcode = '$zipOne'";
6      $result = mysql_query($query);
7
8      if(mysql_num_rows($result) < 1) {
9          return "First Zip Code not found";
10     } else {
11         $row = mysql_fetch_array($result, MYSQL_ASSOC);
12         $lat1 = $row["lat"];
13         $lon1 = $row["lon"];
14     }
15
16     $query = "SELECT * FROM zipData WHERE zipcode = '$zipTwo'";
17     $result = mysql_query($query);
18
19     if(mysql_num_rows($result) < 1) {
20         return "Second Zip Code not found";
21     }else{
22         $row = mysql_fetch_array($result, MYSQL_ASSOC);
23         $lat2 = $row["lat"];
24         $lon2 = $row["lon"];
25     }
26
27     /* Convert all the degrees to radians */
28     $lat1 = $lat1 * M_PI/180.0;
29     $lon1 = $lon1 * M_PI/180.0;
30     $lat2 = $lat2 * M_PI/180.0;
31     $lon2 = $lon2 * M_PI/180.0;
32
33     /* Find the Great Circle distance */
34     $EARTH_RADIUS = 3956;
35
36     $distance = acos(sin($lat1)*sin($lat2)+cos($lat1)*cos($lat2)*cos($lon2-$lon1)) * $EARTH_RADIUS ;
37
38     return $distance;
39
40 } // end func
41
42 ?>
```

Putting it to work.

Phew! If you think the above was mind bending, I can assure you that the worst is over. Everything can now be encapsulated in PHP code very easily.

The first task was to put the available data in a MySQL table with the structure in Listing 1.

Let me clarify here that MySQL was just a personal choice, you can substitute MySQL with any modern RDBMS.

Next task was to write the necessary PHP functions to use the data and the knowledge gleaned from the above process (Listing 2).

I will quickly explain what has happened in listing 2. It is presumed for the sake of simplicity that the code for connecting to the MySQL database is already present.

“...the linear distance covered by 1 degree latitude or longitude is approximately 69.1 miles”

The function `findDistance()` takes two parameters (`$zipOne` and `$zipTwo`), which are the zip codes for the two places between which you wish to find the distance. The important thing to note here is that zip code has to be used as a string rather than an integer unless you want strange results for zip codes starting

Listing 1

```
CREATE TABLE zipData (
  zipcode varchar(10) NOT NULL,
  lon float(3,5) DEFAULT '0.00000' NOT NULL,
  lat float(3,5) DEFAULT '0.00000' NOT NULL,
  city varchar(100) DEFAULT '' NOT NULL,
  state varchar(5) DEFAULT '' NOT NULL,
  PRIMARY KEY (zipcode)
);
```

with 0. So don't forget those single quotes around the zip codes. Lines 5 and 6 compose and execute a query selecting the latitude and longitude for the first zip code. Lines 8 to 14 do some rudimentary error checking in the form of checking that a record was indeed returned and assigning the returned latitude and longitude to `$lat1` and `$lon1`.

Lines 16 to 25 repeat the same with the second zip code (`$zipTwo`), assigning latitude and longitude to `$lat2` and `$lon2` respectively, after which the lines 28 to 31 convert these values into radians. `M_PI`, by the way, is a built-in PHP constant for the value of pi. Finally line 36 applies the *Great Circle Distance* formula to the values we have obtained and returns the result.

You can call this functions in your code like:

```
$distance = findDistance(12345, 23456);
```

`$distance` will now have the distance between zip 12345 and zip 23456 in miles.

Taking it further

My next problem involved writing code which would

Listing 3

```
1 <?php
2
3 function inRadius($zip,$radius)
4 {
5     $query="SELECT * FROM zipData WHERE zipcode='$zip'";
6     $result = mysql_query($query);
7
8     if(mysql_num_rows($result) > 0) {
9         $row = mysql_fetch_array($result, MYSQL_ASSOC);
10        $lat=$row["lat"];
11        $lon=$row["lon"];
12
13        /*This is the magical query */
14        $query="SELECT zipcode FROM zipData WHERE (POW((69.1 * (lon-\"$lon\") * cos($lat/57.3)), \"2\"
15                POW((69.1 * (lat-\"$lat\")), \"2\")) < ($radius*$radius) ";
16
17        $result = mysql_query($query);
18
19        if(mysql_num_rows($result) > 0) {
20            while($row = mysql_fetch_array($result, MYSQL_ASSOC)) {
21                $zipArray[]=$row["zipCode"];
22            }
23            return $zipArray;
24        } else {
25            return "Zip Code not found";
26        }
27 } // end func
28
29 ?>
```

return all zip codes within x miles of a given zip code. Here the problem was to have a function which gives acceptable results, is fast enough and does not clog the server CPU, nor should it take up too much space on disk.

Again recalling high school geometry I remembered that to determine if a point with co-ordinates x,y is within a circle of radius r we used the formula

$$x^2 + y^2 < r^2$$

Using the *Great Circle Distance* formula here was tested in code and deemed too CPU intensive. On a P4 2Ghz machine with 512Mb ram the query shot the CPU usage to 99% and then just stood there until I killed it. This was with a single user - doing this on even a mildly busy website would be nothing short of suicidal.

Combining the above formula with the earlier explained formula using a cosine function in an SQL query I wrote the function in Listing 3.

The function `inRadius()` takes two parameters `$zip` which is the Zip Code for which you want to find nearby zip codes and `$radius` is the radius in miles.

Lines 5 to 11 get the latitude and longitude of the supplied zip code. Using these values, a query is constructed. Many newcomers to MySQL and RDBMS do not realise that they can use mathematical functions within their SQL statements and try to make their PHP code do the heavy lifting with multiple queries. This slows down the code unnecessarily and dramatically.

Let us dissect the WHERE clause of the query in line 14 and see if we can make it more comprehensible.

```
POW((69.1 * (lon-$lon) * cos($lat/57.3)), 2)
```

Starting from the inner most brackets, 69.1 is the distance in miles covered by 1 degree latitude or longitude. To make the value of longitude in the database relative to the values of longitude of the supplied zip code we subtract it, and this is the `(lon - $lon)` part. To make this more accurate, since earth has the longitudes converging at poles, we use the cosine function and multiply it by `cos($lat/57.3)`. Here, as stated earlier, 57.3 is the approximate value of $180/\pi$ which is required to convert degrees to radians. Finally the POW function raises it to the power of 2. We add this to:

```
POW((69.1 * (lat-$lat)), "2")
```

This is the same as what we did for longitude except that the cosine part is not needed here, since latitudes are parallel to each other. Lastly the result of the above two is checked to see if it is less than

```
$radius * $radius
```

Once this query is executed, the function returns an

Trigonometry Primer

The study of angles and of the angular relationships between planar and three-dimensional objects is known as trigonometry. The most commonly encountered trigonometric functions are Sine, Cosine and Tangent. To better understand these we start with a circle with a radius of one unit (Figure 1).

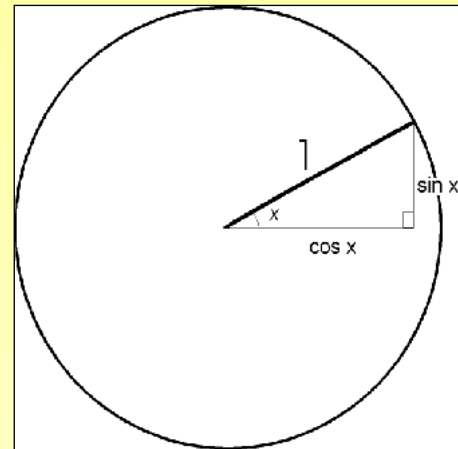


Figure 1

If you draw an angle of n degrees measured anticlockwise from the x -axis to a point on the circumference of the circle, then cosine n is the horizontal co-ordinate of the end point on the circumference and sine n is the vertical co-ordinate. The ratio of sine to cosine is the tangent.

Another way to visualise this is with the help of a right triangle. The three sides of a right angle triangle can be uniquely labeled as the hypotenuse, side adjacent to angle n , and side opposite to angle n . Sine, cosine and tangent can be represented as ratios of two of the three sides. Thus:

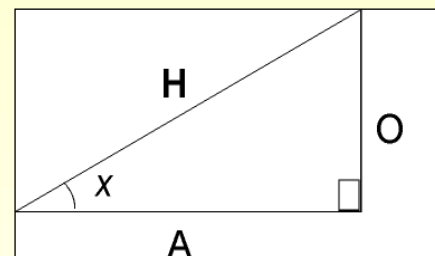


Figure 2

sine n = opposite / hypotenuse
 cosine n = adjacent / hypotenuse
 tangent n = opposite / adjacent

We used to remember this using the mnemonic oh, ah, oh - ah. Apart from these there is another function which is of relevance here which is arc cosine which is just the inverse of cosine and gives the angle for the given cosine.

Radians provide an alternate way of measuring angles and are especially useful when distances are to be interchanged with angles. One radian is equal to the angle formed from the center of the circle when the arc opposite the angle is equal to the radius of the circle.

array of zip codes which are in the given radius. In my programs I use this array of zip codes to build another query to find the necessary data from another table. You need not restrict yourself to this approach, of

“[the Great Circle Distance formula] is used for finding the distance between two points on the surface of a sphere.”

course. You might find it more efficient to join two or more tables in a single query. For example, you have a table called 'dealerData' which has all of the details of your dealers along with their zip codes. If the query in the above inRadius function is modified to include the dealerData table you can get the complete details of the dealers within \$radius miles. The modified query will look like this:

```
$query = "SELECT *.dealerData FROM zipData,
dealerData WHERE (POW((69.1*(lon-\$lon\`)
*cos(\$lat/57.3)),\`2\`) +
POW((69.1*(lat-\`$lat\`)),\`2\`)) <
(\$radius*\$radius) AND
zipcode.dealerData = zipcode.zipData";
```

Again for sake of completeness – you can use this function in your code this way:

```
$zipArray = inRadius(12345,5);
```

A complete working example

You can find the complete working code and example usage in the files accompanying the magazine...

OK! OK! I won't wriggle away from my responsibility and leave you hanging mid way. Let us make a mini app which will take a zip code as input from your visitors and show them the zip codes within the requested radii.

We start by making a simple HTML form as shown in Listing 4.

If you have reached this far I can safely assume that you have realized that the HTML in Listing 4 creates a form with one select element called radius and one text field called zipCode along with a Submit button. You will also presumably know that the fun takes place in zipFun.php.

Listing 4

```
<html>
<head>
<title>ZipFun</title>
</head>
<body>
<p>Please input Zip Code and select a Radius</p>
<form name="form1" method="post"
action="zipFun.php">
  Find Zip codes within
  <select name="radius" >
    <option selected>5</option>
    <option>10</option>
    <option>15</option>
    <option>20</option>
  </select>
  miles of Zip code
  <input name="zipCode" type="text" size="5">
  <input type="submit" name="Submit"
value="Submit">
</form>
</body>
</html>
```

Have you had your
PHP today?



Visit us at <http://www.phparch.com>
and subscribe today.

php | architect

“Many newcomers to MySQL and RDBMS do not realise that they can use mathematical functions within their SQL statements”

The contents of file zipFun.php is shown in Listing 5.

The first 8 lines in the listing just create the necessary link to the database. If you are trying this out, remember to substitute the relevant details to reflect your environment. Lines 10 to 30 define the inRadius function. Since we want to display all the details of the records in our script we have modified the query on line 19 to select * instead of just zipCode. The call to our inRadius function on line 35 results in \$zipArray. \$zipArray in this case is an array of associative arrays which hold information on all of the returned rows. Lines 38 to 40 use the foreach function

to loop through the \$zipArray and print out relevant information. We then add several lines of code to format the output in an HTML table, and making the output w3c compliant is left as an exercise to the reader.

Conclusion

So as you can see, if you have the data, calculating distances and doing radii calculations with various degrees of accuracy is simply a matter of applying a few geometry formulas. The results you obtain can be used in a variety of applications like dealer locators, match making sites, used car sites and more.

About The Author

?>

Dr Tarique Sani is a Pediatrician and Forensic Expert by education and a PHP coder by profession. He is CTO, SANISOFT (<http://www.sanisoft.com>), a web applications development company based in Nagpur, India. he can be reached at tarique@sanisoft.com

Click HERE To Discuss This Article

<http://www.phparch.com/discuss/viewforum.php?f=18>

Listing 5

```

1  <?php
2  $dbUsername = "root";
3  $dbPassword = "";
4  $dbHostname = "localhost";
5  $dbDatabase = "abcd";
6
7  $db = mysql_connect($dbHostname, $dbUsername,$dbPassword) or die("Could not connect");
8  mysql_select_db($dbDatabase) or die("Could not select database");
9
10 function inRadius($zip,$radius)
11 {
12     $query="SELECT * FROM zipData WHERE zipcode='$zip'";
13     $result = mysql_query($query);
14
15     if(mysql_num_rows($result) > 0) {
16         $row = mysql_fetch_array($result, MYSQL_ASSOC);
17         $lat=$row["lat"];
18         $lon=$row["lon"];
19         $query="SELECT * FROM zipData WHERE (POW((69.1*(lon-
20 \\"lon\\")*cos($lat/57.3)),\"2\")+POW((69.1*(lat-\"$lat\\\"),\"2\"))<($radius*$radius) ";
21         $result = mysql_query($query);
22         if(mysql_num_rows($result) > 0) {
23             while($row = mysql_fetch_array($result, MYSQL_ASSOC)) {
24                 $zipArray[]=$row;
25             }
26             return $zipArray;
27         } else {
28             return "Zip Code not found";
29         }
30     } // end func
31
32 $zipCode = $HTTP_POST_VARS["zipCode"];
33 $radius = $HTTP_POST_VARS["radius"];
34
35 $zipArray = inRadius($zipCode,$radius);
36
37 print "<h2>There are ".count($zipArray)." Zip codes within $radius Miles of $zipCode</h2>";
38 foreach($zipArray as $row) {
39     print "<br>ZipCode:$row[zipcode] Lon:$row[lon] Lat:$row[lat] City: $row[city]";
40 }
41 ?>
```