Calculating Geographic Distance: Concepts and Methods
Frank Ivis, Canadian Institute for Health Information, Toronto, Ontario, Canada

ABSTRACT
Calculating the distance between point locations is often an important component of many forms of spatial analysis in business and research. Although specialized software is available to perform this function, the power and flexibility of Base SAS® makes it easy to compute a variety of distance measures. An added advantage is that these measures can be customized according to specific user requirements and seamlessly incorporated with other SAS code. This paper outlines the basic concepts and methods involved in calculating geographic distance. Topics include distance calculations in two dimensions, considerations when dealing with latitude and longitude, measuring spherical distance, and map projections. In addition, several queries and calculations based on distances are also demonstrated.

INTRODUCTION
For many types of databases, determining the distance between locations is often of considerable interest. From a company’s customer database, for example, one might want to examine the distances between customers’ homes and the retail locations where they had made purchases. Similarly, the distance between patients and hospitals may be used as a measure of accessibility to healthcare. In both these cases, an appropriate formula is required to calculate distances. The proper formula will depend on the nature of the data, the goals of the analysis, and the type of co-ordinates. The purpose of this paper is to outline various considerations in calculating geographic distances and provide appropriate implementations of the formulae in SAS. Some additional summary measures based on calculated distances will also be presented.

Geographic Information Systems (GIS) are explicitly designed to store, handle, and retrieve spatially referenced data. In addition to basic Euclidean or straight-line distances, GIS is also capable of more complex forms of analysis using road networks or characteristics of the landscape. In the context of physical geography, GIS could account for elevation, slope, vegetation and bodies of water in calculating appropriate paths between two points. Using network analysis, GIS can also find the shortest path based on existing street networks, or calculate actual travel time or cost of a trip, measures which may be more meaningful than straight-line distance.

Although GIS is ideally suited for this type of work, it typically requires a considerable investment in hardware, software, data and training. More importantly, additional information about the trip is often required to take full advantage of more complex distance algorithms. For example, consider a situation in which you are interested in the distance customers travel to a retail outlet. In addition to the origin and destination, the actual route selected and associated speed limits would be important considerations, since they may not always take the shortest path. There may also be additional factors that cannot be adequately measured or captured but may impact travel times, factors such as the mode of transportation, time of the trip, traffic and weather conditions.

If a suitable GIS is not available or details of the routes are lacking or unimportant, then simpler and easier methods requiring only Base SAS (and perhaps PROC GPROJECT from SAS/GRAPH) may be used. Depending on the data, application, and required accuracy, these methods may be quite satisfactory compared to results calculated with GIS. Not only are they easy to implement and reasonably accurate, programming them in SAS allows you the flexibility to create various distance-based measures and seamlessly incorporate them with your other SAS programs.
DISTANCES IN TWO DIMENSIONS
Consider the situation faced by a botanist studying a stand of oak trees on a small plot of land. One component of the data analysis involves determining the location of these trees and calculating the distance between them. In this situation, straight line or Euclidean distance is the most logical choice. This only requires the use of the Pythagorean Theorem to calculate the shortest distance between two points:

\[
\text{straight\_line\_distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2};
\]

The variables \(x\) and \(y\) refer to co-ordinates in a two-dimensional plane and can reflect any unit of measurement, such as feet or miles.

Consider a different situation, an urban area, where the objective is to calculate the distance between customers' homes and various retail outlets. In this situation, distance takes on a more specific meaning, usually road distance, making straight line distance less suitable. Since streets in many cities are based on a grid system, the typical trip may be approximated by what is known as the Manhattan, city block or taxi cab distance (Fotheringham, 2002):

\[
\text{block\_distance} = (|x_2 - x_1| + |y_2 - y_1|);
\]

Instead of the hypotenuse of the right-angled triangle that was calculated for the straight line distance, the above formula simply adds the two sides that form the right angle. The straight line and city block formulae are closely related, and can be generalized by what are referred to as the Minkowski metrics, which in this case are restricted to two dimensions:

\[
\text{minkowski\_metric} = (|x_2 - x_1|^k + |y_2 - y_1|^k)^{1/k};
\]

The advantage of this formula is that you only need to vary the exponent to get a range of distance measures. When \(k = 1\), it is equivalent to the city block distance; when \(k = 2\), it is the Euclidean distance. Less commonly, other values of \(k\) may be used if desired, usually between 1 and 2. In some situations, it may have been determined that actual distances were greater than the straight line, but less than the city block, in which case a value such as 1.4 may be more appropriate. One of the interesting features of the Minkowski metric is that for values considerably larger than 2 (approaching infinity), the distance is the larger of two sides used in the city block calculation, although this is typically not applicable in a geographic context.

Given the flexibility of the Minkowski metric, it would be advantageous to convert it to a function using the macro language, thereby facilitating its use in a variety of contexts. A suitable macro for distance in two dimensions might be something like the following:

\[
\%macro distance2d(x1=, y1=, x2=, y2=, k=);
\]

\[
\text{max}(\text{abs}(\&x2 - \&x1), \text{abs}(\&y2 - \&y1)) \times (\&k > 2) + (\text{abs}(\&x2 - \&x1)^{\&k} + \text{abs}(\&y2 - \&y1)^{\&k})^{(1/\&k)} \times (1 <= \&k <= 2)
\]

\%mend distance2d;

For the sake of brevity, the above macro contains no error checking or documentation, though this is strongly encouraged, especially if the macro is to be shared. The macro contains five named parameters: \(x_1\) and \(y_1\) refer to the co-ordinates of the origin or first point; \(x_2\) and \(y_2\) refer to the co-ordinates of the destination or second point. The exponent \(k\) is typically 1 or 2 as discussed earlier. The macro actually consists of two equations, controlled by two Boolean variables. If \(1 <= k <= 2\), the basic Minkowski metric is applied, since \((1 <= \&k <= 2)\) resolves to 1 and \((\&k > 2)\) resolves to 0. If \(k > 2\), an alternate formula is applied, since computations become increasingly intensive for large values of \(k\). This second formula is not really necessary, but is useful in demonstrating how modifications can be easily incorporated in distance measures. Using the macro is straightforward. Consider the following Data step:
** Calculating distances in two dimensions ** ;
data points;
  input id $2. +1 org_x 2. +1 org_y 2. +1 dest_x 2. dest_y +1;
  d_block = %distance2d (x1 = org_x, y1 = org_y , x2 = dest_x, y2 = dest_y, k = 1);
  d_Euclid = %distance2d (x1 = org_x, y1 = org_y , x2 = dest_x, y2 = dest_y, k = 2);
  d_other = %distance2d (x1 = org_x, y1 = org_y , x2 = dest_x, y2 = dest_y, k = 1.45);
  d_k3 = %distance2d (x1 = org_x, y1 = org_y , x2 = dest_x, y2 = dest_y, k = 3);
  format d_block d_Euclid d_other 8.2;
  put 'd_block = ' d_block 'd_Euclid = ' d_Euclid 'd_other = ' d_other 'd_k3 = ' d_k3;
datalines;
 1 11 53 19 45
 2 82 74 68 58
 3 20 20 15 33
 4 13 61 63 17
 5 67 31 92 41
;run;

Four different calls to the macro %distance2d are made in the DATA step above. The only difference between them is the value of the exponent k. For k = 1, it calculates the city block distance. For k = 2, the distance is the standard straight line or Euclidean. The third call demonstrates a less common, but potentially useful example of a value between 1 and 2. In some cases, it may be possible to arrive at this value empirically, or it simply may be a best estimate. The last example illustrates the condition of a value greater than 2, in which case the macro interprets it as approaching infinity. Although not as intuitive as the others, the resulting distance when k > 2 may be thought of as the maximum distance required in one direction to reach the other point, which may be useful in some situations.

Which distance to use? This will obviously depend on the context of the particular problem. In some cases, if the relative distances are of interest, the discrepancies may not matter. Although the distances in the above example may appear to be quite different, the Pearson product-moment correlations between them are .98 or higher. Empirical studies that have compared straight line distance to travel time have found comparable results. Phibbs and Luft (1995) for example, examined distances and travel times to hospital in upstate New York and found correlations of .987 and .826 for all distances and distances less than 15 miles, respectively.

** SPHERICAL DISTANCE **
The previous distance measures are based on the concept of distance in two dimensions. For small areas like cities or counties, this is a reasonable simplification. For longer distances such as those that span larger countries or continents, measures based on two dimensions are no longer appropriate, since they fail to account for the curvature of the earth. Consequently, global distance measures need to use the graticule, the coordinate system comprised of latitude and longitude along with special formulae to calculate the distances. Lines of latitude run in an east to west direction either above or below the equator. Lines of longitude run north and south through the poles, often with the Prime Meridian (running through Greenwich, England) measured at 0°. Further details of latitude and longitude are available (Slocum et al., 2005).

One issue with using latitude and longitude is that the co-ordinates may require some transformation and preparation before they are suitable to use in distance calculations. Coordinates are often expressed in the sexagesimal system (similar to time) of degrees, minutes, and seconds, in which each degree consists of 60 minutes and each minute is 60 seconds. Furthermore, it is also necessary to provide and indication of the position relative to the equator (North or South) and the Prime Meridian (East or West). The full co-ordinates may take on a variety of formats; below is a typical example that corresponds approximately to the city of Philadelphia:

39° 55' 48" N     75° 12' 12" W
Note that the seconds provide a level of precision that may not be necessary for large cities such as Philadelphia, since the seconds refer only to a specific location within in the city. Seconds are often omitted altogether when referencing cities, but are included in the following examples for demonstration purposes.

To be able to use these co-ordinates to calculate distances, it is necessary to remove the special characters, convert the co-ordinates to decimal degrees, and properly represent direction. The following DATA step illustrates two approaches to handling these issues using co-ordinates for a variety of major cities. In addition to the co-ordinates and name of the city, other variables include OD (a flag indicating whether the city is an origin or destination, to be used in some later examples) and continent.

```sas
*** Reading the sexagesimal format of latitude and longitude ***;
*** and converting them to decimal degrees ***;
data cities;

*** Two methods to read in sexagesimal degrees: ***;
/*
*** Method 1: Selective Input ***;
input OD $1. +1 city $char14. +2 continent $char2. +1
_lat_deg 2. +2 _lat_min 2. +2 _lat_sec 2. +2 _lat_direction $1. +4
_long_deg 3. +2 _long_min 2. +2 _long_sec 2. +2 _long_direction $1.;
*/

*** Method 2: Read in single string and select required characters;
input OD $1. +1 city $char14. +2 continent $char2. +1
lat_long $21 - 55 ;
* translate various delimiters to a single delimiter;
_lat_long_clean = translate(lat_long,"#","","","","","","");
_lat_long_clean = tranwrd(_lat_long_clean,'N','N#');
_lat_long_clean = tranwrd(_lat_long_clean,'S','S#');
* remove any remaining blanks;
_lat_long_clean = compress(_lat_long_clean,' '); 
* create separate variables for degrees, minutes, seconds
and convert values to numeric;
_lat_deg = input( scan(_lat_long_clean,1,'#'),3. );
_lat_min = input( scan(_lat_long_clean,2,'#'),2. );
_lat_sec = input( scan(_lat_long_clean,3,'#'),2. );
length _lat_direction $1 ;
_lat_direction = scan(_lat_long_clean,4,'#');
_long_deg = input(scan(_lat_long_clean,5,'#'),3. );
_long_min = input(scan(_lat_long_clean,6,'#'),2. );
_long_sec = input(scan(_lat_long_clean,7,'#'),2. );
length _long_direction $1 ;
_long_direction = scan(_lat_long_clean,8,'#');
*** End of second method for reading sexagesimal degrees ***;

*** Transform values to decimal degrees ***;
latitude_decimal = ( _lat_deg + _lat_min/60 + _lat_sec/3600 ) ;
if _lat_direction = 'S' then latitude_decimal = latitude_decimal* -1 ;
longitude_decimal = ( _long_deg + _long_min/60 + _long_sec/3600 ) ;
if _long_direction = 'W' then longitude_decimal = longitude_decimal* -1 ;
drop _: ;
```
The first method exploits the fact that the data are fixed width, so the degrees, minutes, and seconds symbols can be avoided by simply skipping over them. Each of the degree, minute and second values, as well as the direction symbols are stored in separate variables in preparation for the conversion to decimal degrees. The second method is more realistic, as it begins with a single string variable of both latitude and longitude and no fixed widths are assumed. This solution consists of translating the various delimiters to a single delimiter (the # sign), removing any remaining blanks, scanning the string and converting the resulting character fields to numeric. The code can be shortened considerably by nesting more functions, but each step was illustrated separately in this example for clarity.

The conversion simply divides minutes by 60 and seconds by 3600 before adding degrees, minutes and seconds together. Lastly, the latitude is converted to a negative value if it is south of the equator; longitude is typically represented as a positive value if it is east of the Prime Meridian. Now that the co-ordinates are in signed decimal degrees, they are almost ready for distance calculations. Because distance formulae using latitude and longitude require the use of trigonometric functions, which in SAS require the angle to be in radians, it is necessary to convert the decimal degrees to radians by multiplying them by $\pi/180$. In the following examples, the conversion is part of the formulae, but the initial coordinates could also be stored as radians.

The following DATA step demonstrates the two basic methods of calculating spherical distances.

```sas
*** Basic formulae for calculating spherical distance ***;
data _null_;  
  ct = constant('pi')/180;  
  radius = 3959; /* 6371 km */

  ** Both latitude and longitude are in decimal degrees;  
  lat1 = 36.12;
  long1 = -86.67;
  lat2 = 33.94;
  long2 = -118.40;

  ** Law of Cosines **;
  a = sin(lat1*ct) * sin(lat2*ct);
  b = cos(lat1*ct) * cos(lat2*ct) * cos((long2 - long1)*ct);
  c = arcos(a + b);
  d = radius * c;
  put 'Distance using Law of Cosines=' d;
```

** Haversine ** ;
a2 = sin( ((lat2 - lat1)*ct)/2)**2 +
    cos(lat1*ct) * cos(lat2*ct) * sin( ((long2 - long1)*ct)/2)**2 ;
c2 = 2 * arsin(min(1,sqrt(a2))) ;
d2 = radius * c2 ;
put 'Distance using Haversine formula=' d2  ;
run ;

In addition to the constant that will be used to convert degrees to radians, the radius of the earth is required,
which on average is equal to 6371 kilometres or 3959 miles. The Law of Cosines uses spherical geometry to
calculate the great circle distance for two points on the globe. The formula is analogous to the Law of Cosines for
plane geometry, in which three connected great arcs correspond to the three sides of the triangle. The Haversine
formula is mathematically equivalent to the Law of Cosines, but is often preferred since it is less sensitive to
round-off error that can occur when measuring distances between points that are located very close together
(Sinnott, 1984). With the Haversine, the error can occur for points that are on opposite sides of the earth, but this
is usually less of a problem. As with the distance measures in two dimensions, a macro for spherical distance is
straightforward:

```
** Macro for spherical distance, using Haversine formula, and parameter
to specify either miles or kilometers ;

%macro spherical_distance (lat1=,long1=,lat2=,long2=, unit= ) ;
   %local ct ;
   %let ct = constant('pi')/180 ;
   %if %upcase(&unit) = KM %then %let radius = 6371 ;
   %else %if %upcase(&unit) = MI %then %let radius = 3959 ;
   &radius * ( 2 * arsin(min(1,sqrt( sin( ((&lat2 - &lat1)*&ct)/2 )**2
      + cos(&lat1*&ct) * cos(&lat2*&ct) * sin(((&long2-
      &long1)*&ct)/2)**2)))
%mend spherical_distance ;
```

This macro employs the Haversine formula and includes a parameter for the unit of measurement: miles (MI) or
kilometers (KM). Depending on your requirements, the macro could be expanded and embellished in various
ways, including options for the units and level of precision. The following query in PROC SQL demonstrates the
use of the macro to calculate the distance from the origin cities to the various destinations:

```
title1 'Distances (in miles) between all cities';
proc sql number;
   select o.city, d.city
      , d.latitude_decimal as destination_lat
      , d.longitude_decimal as destination_long
      , o.latitude_decimal as origin_lat
      , o.longitude_decimal as origin_long
      ,%spherical_distance(lat1 = origin_lat, long1=origin_long
      , lat2 = destination_lat, long2 = destination_long, unit = MI)
         as distance format = 8.2
   from cities (where = (OD eq 'D')) as d
      , cities (where = (OD eq 'O')) as o
   order by o.city, d.city
; quit ;
```

Note that with the flexibility of PROC SQL, it is possible to read selected rows of the same data set, rename the
columns, join them together and calculate all possible distances from each of the origin destinations.
It should be noted that the great arc distance as computed by either formula is only an approximation due to several limitations. First, it is based on the assumption that the earth is a sphere, when it is in fact an oblate spheroid, bulging slightly at the equator and flatter at the poles. Variations in elevation, topography, and bodies of water may also be a factor when calculating certain distances, but are not accounted for by these formulae. Depending on your requirements and the desired level of accuracy, specialized software may be more appropriate. However, in general, the great arc distance is reasonably accurate for use in a wide variety of applications.

**PROJECTIONS**

Although spherical distance is appropriate for co-ordinates using latitude and longitude, there may be certain situations when it is desirable to project these co-ordinates to two dimensions and calculate distance with the basic Euclidean method discussed earlier. Why would you want to do this? In some situations, the co-ordinates may be used in some form of statistical analysis that assumes the points are in two dimensions. Alternatively, some analysis may involve the use of maps to display the points, in which case it may be preferable to have the distances consistent with the displayed points on the map. In order for this approach to work properly, a suitable projection must be chosen; serious distortions can occur if the wrong projection is used. Any projection will result in some distortion of angles, areas, distance or direction and the challenge is to minimize the distortion in the characteristics of most importance. Choosing the best projection can be a complicated process, and will largely depend on the size and location of the area in question. Formal selection criteria have been developed that can aide in choosing an appropriate projection (Slocum et al., 2005).

In SAS, map projections can be performed with PROC GPROJECTION, a part of SAS/GRAPH which actually does not produce any graphic output. To demonstrate, the cities data will be used. Prior to running the procedure, the required variables x and y are created, since these are the only acceptable variable names for use with PROC GPROJECTION.

```plaintext
*** Prepare data for PROC GPROJECT, selecting cities from North America *** ;
data cities_NA ;
   set cities ;
   x = longitude_decimal ;
   y = latitude_decimal ;
   where continent eq 'NA';
run ;

** Project co-ordinates using an Albers projection ** ;
proc gproject data=cities_NA  out=cities_NA_p
   project=albers   degree;
   id city;
run;
```

This procedure uses simple syntax, and in this particular case, most of the defaults are accepted with the exception of ‘degree’ which indicates that the values are decimal degrees and not radians. SAS9 has a variety of experimental map projections, but currently only three are official options: albers, lambert and gnomonic (gnomonic). Fortunately, in the above example, albers is well-suited for large east-west areas away from the equator such as North America. A comparison of the spherical distances and the Euclidean distances based on the projected co-ordinates above of North American cities reveals that they are generally quite close. The following code compares the spherical distances to the projected two-dimensional distances:

```plaintext
title 'Compare spherical with projected distances';
%let unit = 3959 ; * radius in miles ;
proc sql number;
   create table compare as
      select o.city as origin_city, d.city as destination_city ,
         d.latitude_decimal as destination_lat ,
         d.longitude_decimal as destination_long ,
         d.x as destination_x ,
         d.y  as destination_y ,
         o.latitude_decimal as origin_lat ,
         o.longitude_decimal as origin_long ,
         o.x as origin_x , o.y  as origin_y
```
In all cases, the two distance measures differ by less than five miles. However, it should be stressed that the above approach works only if the appropriate projection is selected for a restricted geographic area.

**CUSTOM QUERIES AND DERIVED VARIABLES BASED ON DISTANCE**

So far, only the basic mechanics of computing distances have been examined. But within the larger context of data analysis, a summary measure based on the distances is often desired so that it may be used in conjunction with other variables. The next few examples present a variety of queries and measures based on the cities data created earlier. As demonstrated earlier, PROC SQL is particularly flexible in these types of problems, as it conveniently accommodates self-joins, summary measures and creation of new variables.

In this first example, the closest destination city to each of the origin locations is desired. The PROC SQL step completes most of the work, calculating the distance between each origin and destination, and sorting them by distance within each origin. The DATA step selects the first (closest) city for each by-group, along with the corresponding distance.

*** Selected Queries ***;
*** Example 1. Closest destination to each origin (min distance) ***;
proc sql;
  create view _distances as
  select o.city as origin_city, d.city as destination_city
    , %spherical_distance
      ( lat1 = origin_lat, long1=origin_long
      , lat2 = destination_lat, long2 = destination_long, unit = MI)
    as distance_lat_long format = 8.2
    , %distance2d
      ( x1 = origin_x*unit, y1 = origin_y*unit
      , x2 = destination_x*unit, y2 = destination_y*unit, k = 2)
    as distance_2d format = 8.2
    , calculated distance_lat_long - calculated distance_2d as difference
  from cities NA_p (where = (OD eq 'D')) as d
  , cities NA_p (where = (OD eq 'O')) as o
  order by o.city, d.city
;
quit;
%symdel unit;

*** Selected Queries ***;
*** Example 1. Closest destination to each origin (kilometres) ***;
proc sql;
  create view _distances as
    select o.city as origin_city, d.city as destination_city
      , %spherical_distance
        ( lat1 = o.latitude_decimal
        , long1 = o.longitude_decimal
        , lat2 = d.latitude_decimal
        , long2 = d.longitude_decimal
        , unit = KM)
      as distance
    from cities (where = (OD eq 'O')) as o
    , cities (where = (OD eq 'D')) as d
  order by origin_city, distance
;
quit;

data min_distance / view = min_distance;
  set _distances;
  by origin_city;
  if first.origin_city;
run;

title 'Example 1: Closest destination to each origin (kilometres)';
proc print data = min_distance;
run;
This next example simply lists destinations less than 6,000 miles from each of the origins. In this instance, there can be a variable number of rows for each origin city.

```
title "Example 2: Destinations within 6,000 miles from each origin" ;
proc sql ;
  select o.city as origin_city, d.city as destination_city ,
    %spherical_distance
    (lat1  = o.latitude_decimal,
     long1 = o.longitude_decimal,
     lat2  = d.latitude_decimal,
     long2 = d.longitude_decimal,
     unit = MI
     ) as distance format = 8.1
  from cities (where = (OD eq 'O')) as o
    , cities (where = (OD eq 'D')) as d
  where calculated distance <= 6000
  order by o.city, distance
;
quit ;
```

The last example is a variation of the preceding query. Rather than list the destinations less than 6,000 miles away from each of the origins, the following code counts the number of destinations that are less than 6,000 miles away.

```
title 'Example 3: N destinations within 6,000 miles from each origin' ;
proc sql ;
select city ,
  sum(distance_flag) as destination_lt6000mi
from (select o.city,
    %spherical_distance
    (lat1  = o.latitude_decimal,
     long1 = o.longitude_decimal,
     lat2  = d.latitude_decimal,
     long2 = d.longitude_decimal,
     unit = MI
    ) as distance ,
  case when calculated distance <= 6000 then 1
     else 0
  end as distance_flag
from cities (where = (OD eq 'O')) as o
  , cities (where = (OD eq 'D')) as d
)
group by city
;
quit ;
```

**CONCLUSIONS**

Although there may be challenges to calculating the exact distance between two points, acceptable results can usually be obtained by using the methods outlined in this paper. These methods are not complicated, but they do require that you understand your data and the goals of your analysis. This paper reviewed several measures of distance in two dimensions, as well as spherical distance. Proper handling of latitude and longitude and methods of projection were also briefly reviewed. Given the constraints of this paper, the primary focus was on using SAS to accomplish these various functions. For more detail on the spatial concepts, you are encouraged to consult some of the references at the end of this paper.
In addition to the basic calculations, this paper also provided examples of how to create queries and new aggregate measures based on the original distance calculations. This, perhaps, is the greatest advantage of using SAS to calculate distance. All stages of data preparation and analysis can be performed with SAS, starting with the initial storage and transformation of the co-ordinates, calculation of distances, and creation of specific variables. These variables can then be used with a wide variety of statistical and reporting procedures that are also available in SAS.

REFERENCES


ACKNOWLEDGEMENTS

SAS is a Registered Trademark of the SAS Institute, Inc. of Cary, North Carolina.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Frank Ivis
Canadian Institute for Health Information
90 Eglinton Avenue East, Suite 300
Toronto, Ontario, M4P 2Y3
CANADA
Phone: (416) 544-5454
Fax: (416) 481-2950
fivis@cihi.ca