Paper 933-2017

Jumping and Cutting:
Using the Hash Object to Implement a Polygon Clipping Algorithm

Seth Hoffman, GEICO

ABSTRACT

Data with a location component is naturally displayed on a general reference map. Base SAS® 9.4 provides libraries of map data sets to assist in creating these images. Sometimes, a particular sub-region is all that needs to be displayed. The SAS/GRAPH GPROJECT procedure can create a new subset of the map using the minimum and maximum latitude and longitude options. However, this method is only capable of cutting out a "rectangular" area.

This paper implements the Greiner Horman Polygon Clipping Algorithm[1] with the help of the DATA Step Hash Object to create arbitrarily shaped custom map regions.

INTRODUCTION

A picture may be worth a thousand words. Sometimes, that might a few hundred too many. Just as a text document could need editing for clarity, so too can a picture. This is especially true in business and research papers where artistic merit is not a relevant concern. Any bit of unneeded color or an extraneous line can create ambiguity in the message a graphic is trying to deliver.

Pictures taken with a camera are raster images. The color value of each pixel is stored in an array. Features in such imagery are created in the mind of the viewer. Professionally editing these requires complicated software tools, talent, and a lot of manual effort.

Vector images only draw the desired features. Rather than having values for every pixel, these files only store important points. These points can be drawn singly or combined to form lines or polygons. Examples of these features found on maps could be landmarks, roads, and state boundaries, respectively. Editing these simply involves moving, adding, or removing the listed points. Requiring only some basic mathematical knowledge, this work can be done programmatically.

The GMAP Procedure reads SAS datasets that hold information about the features it will draw. Each row in the input data contains information about one point. Successive points are connected to form polygons. A new shape will begin when the value of the polygon ID changes. Maintaining the proper row order is of utmost importance for preserving the proper shape.

Knowing where rows are to be added and deleted requires careful indexing and bookkeeping. While it is possible to do so using the ARRAY statement and %MACRO loops, the code can become quite unwieldy. Fortunately the Hash Object works via random access, allowing any point to be accessed via its key value. The clipping algorithm relies on this access, as the interactions between the polygons' points do not happen sequentially.

MAKING USE OF SAS

Algorithms can be coded in any language. SAS provides the tools needed to programmatically make and edit maps. Besides all the map data available in MAPS, and now MAPSGFK, creating custom regions is as simple as creating any other dataset. Viewing installed or custom maps is done with PROC GMAP. Complex decision making that requires jumping around a dataset can now be done in a single DATA step with the Hash Object.

THE GMAP PROCEDURE

With all this talk about shapes, it would be helpful to know how to display them. SAS/GRAPH® includes PROC GMAP which can be used to visualize shape data. The procedure uses ordered sets of coordinates to create an image. In addition to a required "x" and "y" value for every point, each polygon in the dataset must be given a unique ID. The following example defines and draws two shapes:

```
DATA foo;
   poly_id=1; x=0; y=0; OUTPUT;
   poly_id=1; x=1; y=0; OUTPUT;
   poly_id=1; x=1; y=1; OUTPUT;
   poly_id=1; x=0; y=1; OUTPUT;
   poly_id=2; x=2; y=0; OUTPUT;
   poly_id=2; x=4; y=0; OUTPUT;
   poly_id=2; x=3; y=2; OUTPUT;
RUN;

PROC GMAP MAP=foo;
   ID poly_id;
   CHORO poly_id;
RUN;
QUIT;
```

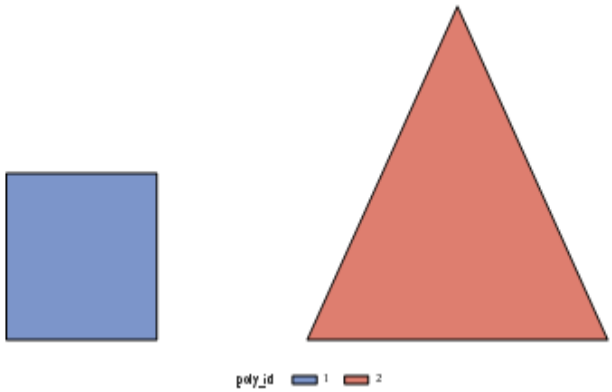

Figure 1. Output from PROC GMAP Example

THE DATA STEP HASH OBJECT

Once in a DATA step, there are limited options for adding and deleting rows. The OUTPUT statement may be used to add more rows, but they can only be added from the current line of the dataset. This is problematic if the program is on the 100th row and then decides a new one has to be inserted after the 3rd row. The DATA step would have to be wrapped in a %MACRO loop to handle this problem.

Feeling our pain, the good folks at SAS introduced a better solution: the Hash Object[2]. While the DATA step can only access one row at a time from the Program Data Vector, Hash Object methods can access any part of a dataset at any point during the execution of the DATA step.

```
DATA foo;
   Id_field=1; Onevar=a; TwoVar=23; RedVar="Cat";  BlueVar="Hat"; OUTPUT;
   Id_field=2; Onevar=b; TwoVar=8;  RedVar="Eggs"; BlueVar="Ham"; OUTPUT;
   Id_field=3; Onevar=c; TwoVar=14; RedVar="Blue"; BlueVar="Red"; OUTPUT;
RUN;
DATA bar;
  IF _N_ EQ 1 THEN DO;
    DECLARE HASH hash_object_name(DATASET:"foo", ORDERED:"a");
    hash_object_name.DEFINEKEY("id_field");
    hash_object_name.DEFINEDATA("OneVar", "BlueVar");
    hash_object_name.DEFINEDONE();
    CALL MISSING(id_field, OneVar, BlueVar);
    %END;
  rc = hash_object_name.find(key:2);
  x = OneVar;
  y = BlueVar;
RUN;
```

In the above example, a data set is defined. It is then loaded as a Hash object in the next DATA step. Any field with unique values for every row can be used in the *definekey* method. This key value is then used to call the values of other variables listed in the *definedata* method. When the *find* method is called, the given key value will access the values from its "row" so that they can be used in further lines of the DATA step. The key value of "2" means that X will then have the value "b" and Y will have the value "Ham".

POLYGON MANAGEMENT

Regions of the Earth are divided up in many different shapes and sizes. Often, the only step required for producing clear visual output is deciding which areas to print. Even if some regions need to be divided or otherwise altered, any paint program can do the job for making the image publication ready.

However, there are several advantages to changing the map data. Doing so allows the use of built-in functions to calculate properties like area or the GINSIDE procedure. Another advantage is reduction in runtime by making the edits as part of the preprocessing stage rather than applying some math every time an analysis runs. Finally, changing the map data increases the portability as any mathematical transformations in code could require subtle changes if moved to a different system.

"Polygon clipping" is one such method. This operation takes two polygons and returns their union. The subject polygon is the original shape to be cut down. The clipping polygon is an overlay, or mask, used to guide the cutting of the subject area. One limitation is that neither of the polygons be self-intersecting. For map areas representing real spatial areas, this is never an issue. A second limitation is that the polygons do not have holes. In the world of mapping, holes just represent some other area fully bounded by one of the polygons. On the subject polygon, a hole would be something like a small body of water. A hole in the clipping polygon would require first producing output for the outer clipping polygon, and then running a second time to remove any results that were in the negative mask area of the hole.

POLYGON CLIPPING, AN EXAMPLE

Before beginning, define both the clipping and the subject polygons. In the example below (Figure 2.), the black rectangle is the clip and the green "G" shape is the subject. Both polygons have their vertices numbered, with 4 for the clip and 12 for the subject. Making sure that two polygons are both clockwise or both counter-clockwise is an additional pre-processing step that will be important for the second phase.

Phase one calculates intersection points between line segments of the clip polygon and any other segment from the subject. These new points are then inserted back into each polygon and the point indices are updated. Had there been overlapping line segments, one point would be added at the start of the overlap, and another at the end. In the example below, there were 6 line intersections, raising the number of vertices in the clipping polygon to 10 and the subject to 18.

Phase two involves tracing the polygons and determining which line segments to keep. Starting with point number one of the subject polygon, the algorithm follows along till a line segment exits (Out) a boundary defined by a clipping line segment. When such a crossing occurs, the program will switch to follow the clipping polygon. It will follow he clip till it comes to another intersection that enters (In) the clip. Because both polygons are traversed in the same direction, the algorithm will follow a consistent direction when encountering a clip-subject intersection.

Finally, phase three takes all the bookkeeping done in the previous phase and creates the output polygons. Subject points 1, 2, 3, and 18 form a complete polygon. After turning from point 18, the algorithm next finds point 1. Because that point was already traversed, it will start again at the next lowest untraversed point in the "to keep" list. Points 4 and 5 on the subject polygon are outside of the clip, so the second output polygon will begin with point 6.

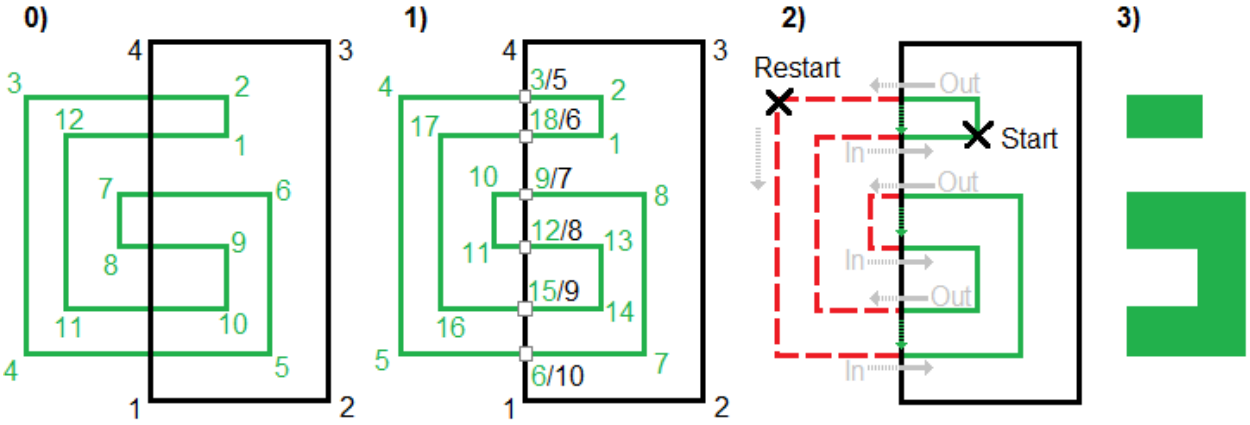

Figure 2. Phases of the Polygon Clipping Algorithm

POLYGON CLIPPING, THE CODE

The following implementation of the Greiner Horman Polygon Clipping Algorithm is much longer than necessary. The main consideration for writing the code in this manner was to show every functional part as an individual step. The Hash Object is only used for the final phase. Using the *add*, *remove*, and *replace* methods to make shorter work of phase 2 is left as an exercise for the enthusiastic reader.

```
%MACRO clip_polygon();

/*** Build line segments ***/
DATA subject2 (KEEP=x1 x2 y1 y2 sega alpha inside);
  SET subject END=eof;
  RETAIN ox oy x1 y1 sega;
  alpha = 0;   inside = 0;
  IF _N_ EQ 1 THEN DO;     /*Save to paste to end*/
     ox = x; oy = y; x1 = x; y1 = y; sega = 0; DELETE;
     END;
  ELSE DO;
     x2 = x; y2 = y; sega = sega + 1; OUTPUT subject2;
     x1 = x; y1 = y;
     END;
 IF eof THEN DO;
     x2 = ox; y2 = oy; sega = sega + 1; OUTPUT subject2;
     END;
RUN;

DATA clip2 (KEEP=x3 x4 y3 y4 segb beta inside);
  SET clip END=eof;
  RETAIN ox oy x3 y3 segb;
  beta = 0; inside = 0;
  IF _N_ EQ 1 THEN DO;     /*Save to paste to end*/
     ox = x; oy = y; x3 = x; y3 = y; segb = 0;     DELETE;
     END;
  ELSE DO;
     x4 = x; y4 = y; segb = segb + 1;  OUTPUT clip2;
     x3 = x; y3 = y;
     END;
  IF eof THEN DO;
     x4 = ox; y4 = oy; segb = segb + 1;      OUTPUT clip2;
     END;
RUN;

/*Determine if polygons are clockwise (>0) or counter-clockwise(<0)*/
PROC SQL;
   SELECT DISTINCT SUM((x2 - x1) * (y2 + y1)) INTO: subj_dir
   FROM subject2;
QUIT;

PROC SQL;
   SELECT DISTINCT SUM((x4 - x3) * (y4 + y3)) INTO: clip_dir
   FROM clip2;
QUIT;
%PUT &subj_dir;
%PUT &clip_dir;

/*Make sure both polygons go counterclockwise*/
%IF (&subj_dir > 0) %THEN %DO;
   PROC SORT DATA=subject2; BY DESCENDING sega; RUN;
   DATA subject2(DROP=tx ty);
```

```
      SET subject2;
      sega = _N_; /*reorder segment numbers*/
      tx = x1; ty = y1;
      x1 = x2; y1 = y2;
      x2 = tx; y2 = ty;
      RUN; /*%PUT "subject is now counter-clockwise";*/
%END;

%IF (&clip_dir > 0) %THEN %DO;
   PROC SORT DATA=clip2; BY DESCENDING segb; RUN;
   DATA clip2(DROP=tx ty);
      SET clip2;
      segb = _N_;
      tx = x3; ty = y3;
      x3 = x4; y3 = y4;
      x4 = tx; y4 = ty;
      RUN; /*%PUT "clip is now counter-clockwise";*/
%END;

/*********************************************************
 *** POLYGON CLIPPING PHASE I: Find The Intersections ***
 *********************************************************/
PROC SQL;
   CREATE TABLE intersections AS
   SELECT a.*, b.*
   FROM subject2 AS a,
        clip2 AS b
   WHERE MAX(x1,x2) GE MIN(x3,x4) AND
         MIN(x1,x2) LE MAX(x3,x4) AND
         MAX(y1,y2) GE MIN(y3,y4) AND
         MIN(y1,y2) LE MAX(y3,y4)
   ORDER BY a.sega, b.segb;
QUIT;

/*** Create and append intersection points ***/
DATA intersections2(KEEP=sega segb alpha beta xint yint inside);
   SET intersections;
   alpha_den = ((y2 - y1) * (x3 - x4)) - ((x2 - x1) * (y3 - y4));
   beta_den = ((y2 - y1) * (x3 - x4)) - ((x2 - x1) * (y3 - y4));
   IF (alpha_den NE 0) AND (beta_den NE 0); /*Prevent infinity or NAN*/
   alpha = (((y3 - y4) * (x1 - x3)) -
            ((x3 - x4) * (y1 - y3))) / alpha_den;
   beta = (((x2 - x1) * (y1 - y3)) - ((y2 - y1) * (x1 - x3))) / beta_den;
   IF ((alpha GE 0) AND (alpha LE 1)) AND ((beta GE 0) AND (beta LE 1));
   /*Don't intersect out of bounds*/
   xint = x1 + alpha * (x2 - x1);
   yint = y1 + alpha * (y2 - y1);
   inside = 1;
RUN;

PROC SQL; /*only keep non-co-incident intersections*/
   CREATE TABLE subj_intersect AS
   SELECT d.*, c.drop_ind
   FROM (SELECT a.xint, a.yint, 1 AS drop_ind
         FROM intersections2 AS a INNER JOIN
              subject2 AS b
         ON a.xint = b.x1 AND
```

```
            a.yint = b.y1) AS c RIGHT JOIN
         intersections2 AS d
    ON c.xint = d.xint AND
       c.yint = d.yint;
 QUIT;

PROC SQL;
  CREATE TABLE clip_intersect AS
  SELECT d.*, c.drop_ind
  FROM (SELECT a.xint, a.yint, 1 AS drop_ind
           FROM intersections2 AS a INNER JOIN
                  clip2 AS b
           ON a.xint = b.x3 AND
         a.yint = b.y3) AS c RIGHT JOIN
         intersections2 AS d
  ON c.xint = d.xint AND
     c.yint = d.yint;
QUIT;

 /*Add intercept points back into each polygon*/
 DATA subject3(KEEP=sega alpha x1 y1 inside);
    SET subject2 subj_intersect(RENAME=(xint=x1 yint=y1));
    IF (drop_ind EQ .);
 RUN;

 DATA clip3(KEEP=segb beta x3 y3 inside);
    SET clip2 clip_intersect(RENAME=(xint=x3 yint=y3));
    IF (drop_ind EQ .);
 RUN;

 PROC SORT DATA=subject3 OUT=subject3(KEEP=x1 y1 inside);
    BY sega alpha;
 RUN;
 PROC SORT DATA=clip3 OUT=clip3(KEEP=x3 y3 inside);
    BY segb beta;
 RUN;

 /*** Find number of points in each polygon and index each point***/
 DATA _NULL_;
    SET subject3 end=eof;
    IF eof THEN CALL SYMPUTX('subject_len', _N_);
 RUN;
 DATA _NULL_;
    SET clip3 end=eof;
    IF eof THEN CALL SYMPUTX('clip_len', _N_);
 RUN;
 /*data sets will be concatenated, each point id needs to be unique*/
 DATA subject3; SET subject3; point = _N_; RUN;
 DATA clip3; SET clip3; point = _N_ + &subject_len; RUN;

/*********************************************************
 *** POLYGON CLIPPING PHASE II: Find The Intersections ***
 *********************************************************/
 PROC GINSIDE DATA=subject MAP=clip OUT=inside INSIDEONLY; ID inside; RUN;
 PROC CONTENTS DATA=inside OUT=content_inside(KEEP=nobs) NOPRINT; RUN;
 PROC CONTENTS DATA=intersections OUT=content_intersect(KEEP=nobs) NOPRINT;
 RUN;
```

```
DATA _NULL_;
   SET content_inside (OBS=1);
   CALL SYMPUTX('count_inside',nobs);
RUN;

DATA _NULL_;
   SET content_intersect (OBS=1);
   CALL SYMPUTX('count_intersect',nobs);
RUN;

%IF (&count_inside = &subject_len) %THEN %DO;
/*subject is entirely inside clip, return subject*/
   DATA out;
      SET subject;
      new_poly = 1;
      new_point = _N_;
   RUN;
   %RETURN;    /*nothing left to do*/
   %END;
%ELSE %IF (%EVAL(&count_inside + &count_intersect) = 0) %THEN %DO;
/*if clip is entirely in subject, return clip*/
   DATA clipx; SET clip(DROP=inside); RUN;
   DATA subjectx; SET subject; inside = 1; RUN;
   PROC GINSIDE DATA=clipx MAP=subjectx OUT=insidex INSIDEONLY; ID inside;
   RUN;
   PROC CONTENTS DATA=insidex OUT=content_inside2(KEEP=nobs) NOPRINT; RUN;
   DATA _NULL_;
      SET content_inside2 (OBS=1);
      CALL SYMPUTX('count_inside2',nobs);
   RUN;
   %IF (&count_inside2 = &clip_len) %THEN %DO;
      DATA out;
         SET clipx;
         new_poly = 1;
         new_point = _N_;
      RUN;
      %RETURN;
      %END;
   %ELSE %DO;
      %RETURN; /*No part of subject is within clip*/
      %END;
   %END;
%ELSE %DO;     /*continue as normal*/

/*** link intersection points ***/
   PROC SQL;
      CREATE TABLE subject4 AS
      SELECT c.point, c.x1 AS x, c.y1 AS y,
             (c.inside + (NOT (NOT e.inside))) AS inside, d.cross_id
      FROM subject3 AS c LEFT JOIN
           (SELECT DISTINCT a.point, b.point AS cross_id
            FROM subject3 AS a INNER JOIN
                 clip3 AS b
            ON a.x1 = b.x3 AND
               a.y1 = b.y3) AS d
      ON c.point = d.point LEFT JOIN
```

```
      inside AS e
   ON c.x1 = e.x AND
      c.y1 = e.y
   ORDER BY c.point;
QUIT;

PROC SQL;
   CREATE TABLE clip4 AS
   SELECT c.point, c.x3 AS x, c.y3 AS y, d.cross_id, . AS inside
   FROM clip3 AS c LEFT JOIN
        (SELECT DISTINCT a.point, b.point AS cross_id
         FROM clip3 AS a INNER JOIN
              subject3 AS b
         ON a.x3 = b.x1 AND
            a.y3 = b.y1) AS d
   ON c.point = d.point
   ORDER BY c.point;
QUIT;

DATA final; SET subject4 clip4; RUN;

/*************************************************************
 *** POLYGON CLIPPING PHASE III: Create The Output Polygons ***
 *************************************************************/
DATA out(KEEP=new_poly new_point x y);
   IF _N_ EQ 1 THEN DO;
      DECLARE HASH s(DATASET:"final", ORDERED:"a");
      s.DEFINEKEY("point");
      s.DEFINEDATA("x", "y", "cross_id", "inside");
      s.DEFINEDONE();
      CALL MISSING(point, x, y, cross_id, inside);
      END;
   LENGTH used_subject $&subject_len.;
   /*polygon length dynamic, so must initialize during run*/
   DO i = 1 TO &subject_len;
      SUBSTR(used_subject, i, 1) = '0';
      END;
   new_poly = 0; /*each new polygon should get a unique ID*/
   DO i = 1 TO &subject_len;
      rc = s.find(key:i);
      IF ((inside EQ 1) AND (SUBSTR(used_subject,i,1) EQ 0)) THEN DO;
         /*Starting new polygon*/
         new_point = 1;
         new_poly = new_poly + 1;
         j = 0; /*set default so loop executes*/
         /*new polygon closes when comes back to starting place*/
         DO UNTIL (MIN(j,cross_id) EQ i);
            IF (j EQ 0) THEN j = i; /*reset j after 1st test passed*/
            IF (MIN(j, cross_id) LE &subject_len) THEN
               SUBSTR(used_subject,MIN(j, cross_id),1) = 1;
               /*Only need to mark off subject points*/
            OUTPUT;
            IF (j LE &subject_len) THEN DO;
            /*If on subject, look ahead for inside status of next pt*/
               lookahead = MOD(j, &subject_len) + 1;
            /*next subject. Other path would be current "cross_id", so
              save current as "other_branch"*/
```

```
                    other_branch = cross_id;
                    rc = s.find(key:lookahead);
                    IF (inside EQ 1) THEN j = lookahead; /*-> subj*/
                    ELSE DO;
                       j = MOD(other_branch - &subject_len, &clip_len) +
                           &subject_len + 1; /*follow clip*/
                       rc = s.find(key:j); /*subject NOT inside, -> clip*/
                       END;
                    END;
                 ELSE IF (cross_id NE .) THEN DO;
                    /*on clip and intersects with subject*/
                    lookahead = MOD(cross_id, &subject_len) + 1;
                    /*preserves current "j", so don't need "other_branch"*/
                    rc = s.find(key:lookahead);
                    IF (inside EQ 1) THEN j = lookahead; /*follow subject*/
                    ELSE DO;
                       j = MOD(j - &subject_len, &clip_len) +
                           &subject_len + 1;
                       rc = s.find(key:j); /*subject NOT inside, -> clip";*/
                       END;
                    END;
                 ELSE DO;    /*on clip and no intersection*/
                    j = MOD(j - &subject_len, &clip_len) + &subject_len + 1;
                    rc = s.find(key:j);
                    END;
                 new_point = new_point + 1;
                 END; /*loop back to DO UNTIL*/
              END; /*ends IF which starts new polygon*/
           SUBSTR(used_subject,i,1) = 1; /*point checked. Won't be reused*/
           END; /*ends DO that loops through subject*/
        RUN;
     %END;
%MEND clip_polygon;
```

CONCLUSION

While the algorithm itself is simple, there is a fair amount of bookkeeping involved in deciding which point to move to next. Without the Hash Object's ability to jump around the index list, the code for the final phase would be much longer and more complex.

My most common usage of this code is when answering questions of the form "Tell me … within an X kilometer radius". Rather than just including any zip-code or county that has some portion of itself within 150 kilometers of the center of study, those edge regions are trimmed down. Study areas are then consistently πR^2.

Applications of clipping tend to involve polygons that are small relative to Earth's circumference. As such, arcs between two geographic points can be treated as Cartesian line segments with very little error. If the clipping region extends into the 100's of kilometers or the mapping regions are on a small moon, using spherical geometry will be required to produce accurate results.

Most arcs are not actually part of great circles, meaning that their planes do not pass through the center of the sphere on which they are drawn. Most arcs are actually Rhumb lines. These arcs follow a constant compass direction, making straight lines when drawn on a standard Mercator projection map. The following code[3] can be swapped in for the intersection calculation code that creates the dataset "intersections2".

```
   /* Here, 1 and 2 refer to the starting and ending point of a rhumb line*/
   Theta13 = 2 * ARSIN(MIN(COS(lat1) * SIN(lat2) -
                           SIN(lat1) * COS(lat2) * COS(lon2 - lon1),
```

```
                        SIN(lon2 - lon1) * COS(lat2)));
/* Repeat for Theta23 using point 2*/

/* Here 1 and 2 refer to the starting points of the 1st and 2nd rhumb line*/
a1 = SIN((lat1 - lat2) / 2)**2 +
     COS(lat1) * COS(lat2) * SIN((lon1 - lon2) / 2)**2;
Delta12 = 2 * ARSIN(MIN(1, SQRT(a1)));
Theta_a = ACOS((SIN(lat2) - SIN(lat1) * COS(Delta12) /
                (SIN(Delta12) * COS(lat1));
Theta_b = ACOS((SIN(lat1) - SIN(lat2) * COS(Delta12) /
                (SIN(Delta12) * COS(lat2));

IF SIN(lon2 - Lon1) > 0 THEN DO;
   Theta12 = Theta_a;
   Theta21 = 2 * CONSTANT('pi') - Theta_b;
   END;
ELSE DO;
  Theta12 = 2 * CONSTANT('pi') - Theta_a;
  Theta21 = Theta_b;
  END;

Alpha1 = Theta13 - Theta12;
Alpha2 = Theta21 - Theta23;
Alpha3 = ACOS(-COS(Alpha1) * COS(Alpha2) +
              SIN(Alpha1) * SIN(Alpha2) * COS(Delta12));
Delta13 = 2 * ARSIN(MIN(SIN(Delta12) * SIN(Alpha1) * SIN(Alpha2),
                        COS(Alpha2) + COS(Alpha1) * COS(alpha3)));
Lat_intersect = ARSIN(SIN(lat1) * COS(Delta13) +
                 COS(lat1) * SIN(Delta13) * COS(Theta13));
DeltaLon13 = 2 * ARSIN(MIN(SIN(Theta13) * SIN(Delta13) * COS(lat1),
                           COS(Delta13) - SIN(lat1) * SIN(Lat_intersect));
Lon_intersection = lon1 - DeltaLon13;
```

REFERENCES

1) Greiner, Günther; Kai Hormann. 1998. “Efficient Clipping of Arbitrary Polygons.” *ACM Transactions on Graphics*, 17 2: 71 - 83.

2) SAS Support. “SAS 9 – Hash Object Tip Sheet”. Accessed November 30, 2016. https://support.sas.com/rnd/base/datastep/dot/hash-tip-sheet.pdf

3) Movable Type Scripts. "Calculate distance, bearing and more between Latitude/Longitude points." Accessed January 10, 2017. http://www.movable-type.co.uk/scripts/latlong.html

ACKNOWLEDGMENTS

I would like to thank SAS and this year’s conference chair, John Amrhein for organizing and supporting another year of Global Forum. I would also like to thank Professor Ranga Myneni of Boston University, without whose generosity and support, I would still think geography was mostly about knowing state capitols.

CONTACT INFORMATION

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

Seth W. Hoffman
301-986-3072
Seth.W.Hoffman@gmail.com