Mathematical Geology, Vol. 21, No. 8, 1989

Locating a Point on a Spherical Surface Relative to a Spherical Polygon of Arbitrary Shape[1]

Michael Bevis[2] and Jean-Luc Chatelain[3]

An algorithm for determining if any given point, P, *on the surface of a sphere is located inside, outside, or along the border of an arbitrary spherical polygon,* S, *is described. The polygon is described by specifying coordinates of its vertices, and coordinates of some point* X *which is known to lie within* S. *The algorithm is based on the principle that an arc joining* X *and* P *will cross the border of* S *an odd number of times if* P *lies outside* S, *and an even number of times if* P *lies within* S. *The algorithm has been implemented as a set of FORTRAN subroutines, and a listing is provided. The algorithm and subroutine package can be used with spherical polygons containing holes, or with composited spherical polygons.*

KEY WORDS: Spherical, polygon, locate, sort, algorithm, FORTRAN, subroutine.

INTRODUCTION

Spherical polygons are polygons confined to the surface of a sphere; their sides are great circle arcs. Any shape on the surface of a sphere can be approximated (to any degree of accuracy) by a spherical polygon, provided that the polygon incorporates a sufficient number of vertices (or, equivalently, sides). This paper describes an algorithm that locates a point on the surface of a sphere relative to a spherical polygon of arbitrary shape (i.e., it determines if a given point lies inside, outside, or on the boundary of a given spherical polygon). Many authors have discussed ''point-in-polygon'' algorithms in the context of plane surface or cartesian (x, y) coordinate systems (e.g., Hall, 1975; Salomon, 1978; Davis and David, 1980; or almost any computer graphics textbook). This is, to our knowledge, the first extension of this class of algorithms to the spherical environment.

[1]Manuscript received 6 October 1988; accepted 12 January 1989.

[2]Department of Marine, Earth and Atmospheric Sciences, North Carolina State University, Raleigh, North Carolina 27695-8208.

[3]Institut Français de Recherche Scientifique pour le Développement en Coopération (ORSTOM), Nouméa, New Caledonia.

The primary application for point-in-spherical-polygon algorithms is the sorting of sphere-based data on the basis of their location. Because Earth's surface approximates a spherical surface, this includes geographical sorting of geo-based data. Point-in-spherical-polygon algorithms also facilitate use of nontrivial boundaries in numerical modeling and computational statistics. For example, a spherical polygon can be used to indicate a domain within which a geographical trend surface is adequately constrained by the data it characterizes.

The following discussion begins with a review of terminology and conventions. This is followed by a description of the point-in-spherical-polygon algorithm, and its implementation as a set of subroutines coded in FORTRAN (Appendix A). In the next section, some practical issues that arise during the application of the algorithm and subroutines are addressed. The most important of these issues is the use of multistage sorting. Finally, the persistent reader is presented with a challenge.

TERMINOLOGY AND CONVENTIONS

In the following discussion, a great circle arc joining two points on the surface of a sphere is referred to in various contexts. This requires some care because usually two great circle arcs are associated with a given pair of points. Let P and Q be distinct points on the surface of a sphere. Assume that P and Q are not antipodal (i.e., that they are not separated by exactly 180°). Then, only one great circle passes through both P and Q, but it can be divided into two arcs each of which has P and Q as its end points. The longer of these great circle arcs is called the major arc, and the shorter is the minor arc. The convention adopted here is that that any reference to *a* or *the* great circle arc joining any two points such as P and Q, refers to the minor arc unless explicitly stated otherwise. Note that this convention is rendered meaningless in the special case of antipodal points, because two such points can be joined by an infinite number of great circle arcs of equal length (180°). In the interest of brevity, a *great circle arc* often will be referred to as an *arc* in the following discussion. The word arc refers to a great circle arc unless explicity stated otherwise. Similarly, the adjective *spherical* often is dropped, and a spherical polygon is referred to as a polygon, etc.

An n-sided spherical polygon can be described completely by specifying location of its vertices. One vertex arbitrarily is called V_1 and the remaining vertices are numbered (V_2, V_3, · · · V_n) sequentially around the boundary from V_1 (Fig. 1). An important problem occurs when a spherical polygon is described in this way. A polygonal boundary on the surface of a sphere divides that surface into two domains *both* of which are spherical polygons. Which of these complementary polygons is the one under consideration? Bevis and Cambareri (1987) used the *direction* of vertex enumeration to flag which complementary polygon was being described. Here, some point X inside the polygon is speci-

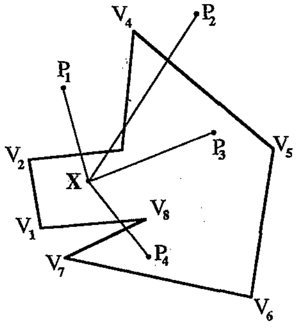

Fig. 1. The shaded area represents an eight-sided spherical polygon, S. Vertices of this polygon are labeled V_1, V_2, $\cdots$ V_8. Some point X is known to lie inside of S. Points P_1, P_2, P_3, and P_4 are located arbitrarily. An arc joining point X to any point lying outside S (e.g., P_1 and P_2) will cross the boundary of S an odd number of times, and an arc joining X and some point inside S (e.g., P_3 and P_4) will cross the boundary of S an even number of times.

fied, and this resolves any potential ambiguity about which of the complementary polygons is under study. The user of the algorithm presented below is free to number vertices in either direction.

The location of a point on the surface of a sphere is specified in terms of its latitude (λ) and longitude (ϕ). Thus, the *i*th. vertex V_i is assigned coordinates (λ_i, ϕ_i), and point X has coordinates (λ_X, ϕ_X).

By convention, the spherical polygon under consideration is produced by joining each neighboring pair of vertices by a minor great circle arc. A spherical polygon which has one or more major arcs for a side can be described by breaking each major arc into two minor arcs by introducing a pseudovertex somewhere along that major arc. Note that for purposes of description, neighboring vertices may never be antipodal (separated by exactly 180°), because the polygon would not be uniquely defined. A spherical polygon which includes one side (or more) whose length is exactly 180° is handled by introducing a pseudovertex which breaks that side into two minor arcs. (A pseudovertex has interior and exterior angles of 180°, unlike a true vertex.)

One restriction exists on the user's choice of the point X inside the polygon under consideration. Point X must not lie on any great circle that passes through two neighboring vertices. The significance of this restriction will become apparent later. Because this condition can be tested in any code that implements the algorithm, it will not burden the user unduly.

THE POINT-IN-SPHERICAL-POLYGON ALGORITHM

The basis of the algorithm is a simple extension to the spherical environment of an algorithm frequently used with plane polygons. Assume a spherical polygon S and some point X located therein (described in the manner explained above) are given. Consider any point P which is not antipodal to point X. The problem is to determine if P lies inside or outside of S, or on its border. The key to this problem is that XP (the minor arc joining X and P) will cross the boundary of S an even number of times if P is inside S, and an odd number of times if P is outside S. For example (Fig. 1), points P_1 and P_2 lay outside of polygon S, and minor arcs XP_1 and XP_2 cross the boundaries of S one time and three times, respectively; whereas points P_3 and P_4 lie inside S, and arcs XP_3 and XP_4 cross the boundaries of S not at all and twice, respectively. The kernel of the problem is to determine if any arc XP crosses any given side of the spherical polygon. This determination is made for each side in turn, and the total number of crossings is counted. The algorithm must recognize in addition the special case when P lies on the boundary of S.

The problem is illustrated in Fig. 2. Does minor arc XP cross the side whose vertices are A and B? First, determine if the strike (azimuth) of arc XP at point X is intermediate between (or equal to) that of arcs XA and XB. This is a necessary (but not sufficient) condition for arc XP to intersect arc AB (the polygon side). This test is implemented by transforming (Bevis and Cambereri, 1987) into a new coordinate system (λ', ϕ') in which point X acts as the north pole ($\lambda' = 90°$). The prime meridian in this new system passes through the north pole (N) in the original system. In this new coordinate system, it is determined if the longitude of P (i.e., ϕ'_P) lies in the range $\phi'_A \leftrightarrow \phi'_B$. This range is shaded (Fig. 2). (Two ranges of longitude have ϕ'_A and ϕ'_B as end values; the range of interest is that which spans less than 180°.) If this condition is not met, arc XP cannot possibly cross side AB, and no further consideration of this side is necessary. If this condition is met, arc XP may cross side AB. However this is not necessarily the case; the condition of necessary strike is met by point P_2 (Fig. 2), but XP_2 does not cross side AB. Therefore, another test is necessary.

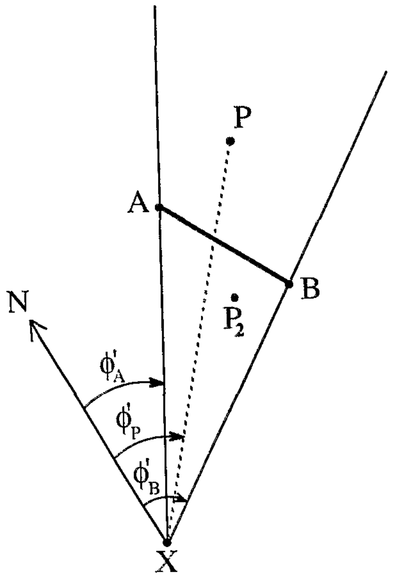

Fig. 2. A schematic illustration of the kernel problem: does arc XP cross the side with vertices at A and B? A necessary (but not sufficient) condition is that arc XP must lie within the shaded region. This is called the condition of necessary strike. (This condition is not a sufficient condition because XP_2 satisfies the condition of necessary strike, but XP_2 does not cross side AB.) To test if the condition of necessary strike is satisfied, the system is transformed to a new spherical coordinate system in which point X acts as the "north pole," and in which the prime meridian passes through the north pole, N, of the original coordinate system. In this coordinate system, arcs XA, XP, and XB fall along meridians with longitudes ϕ'_A, ϕ'_P, and ϕ'_B, respectively. The condition of necessary strike is satisfied if and only if ϕ'_P falls in the range $\phi'_A \leftrightarrow \phi'_B$.

If the condition of necessary strike is met, determine next if *XP* crosses side *AB*. This issue is resolved by determining if points *X* and *P* lie on the same side or on opposite sides of arc *AB*. This is achieved by transforming into a third coordinate system (λ'', ϕ'') in which point *A* acts as the north pole ($\lambda'' = 90°$). Then simply compute the signed angles α and β (Fig. 3) and determine whether or not they have different signs. In this case, one of the points, *P* or *X*, lies ''east'' of arc *AB* and the other point lies ''west'' of this arc; hence arc *XP* must cross side *AB*. In the special case $\beta = 0$, point *P* lies on side *AB* (on *A*, or on *B*, or on the intervening arc). For example, arc *XP* (Fig. 3) must cross polygon side *AB* because points *X* and *P* lie on different sides of arc *AB*, whereas arc XP_2 cannot cross side *AB* because points *X* and P_2 both lie on one side of *AB* (i.e., to the west).

The two tests described above resolve the problem of whether or not arc *XP* crosses any given side of the polygon, or if point *P* lies on that side. All

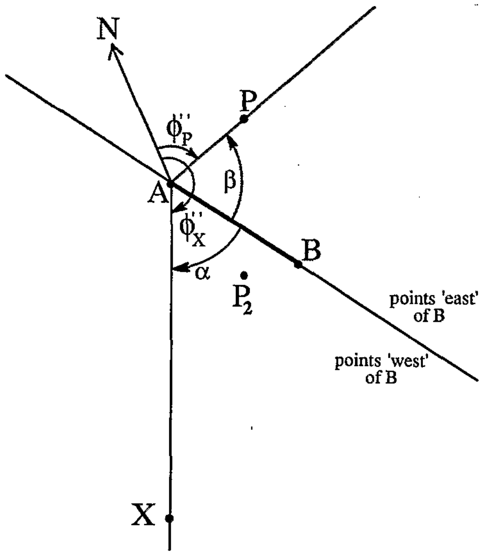

Fig. 3. Once the condition of necessary strike (Fig. 2) is known to have been satisfied, arc *XP* must cross side *AB* if points *X* and *P* lie on opposite sides of a great circle passing through *A* and *B*. This condition is tested by transforming into a coordinate system in which point *A* acts as the north pole, and in which the prime meridian passes through the north pole, *N*, of the original coordinate system. Points *X* and *P* lie one to the east of *B*, and the other to the west of *B*, if and only if arc *XP* crosses arc *AB*. In the case that point *P* lies neither east nor west of *B*, then *P* lies on arc *AB*.

that is necessary is to apply these tests (the second test is applied only if the first test for necessary strike is passed) to each side of the polygon in turn. This is achieved by identifying each neighboring pair of vertices in turn with the generic vertices A and B (i.e., let $A = V_i$ and $B = V_{i+1}$, for $i = 1, 2, \cdots, n - 1$, then let $A = V_n$ and $B = V_1$). If at any stage point P is found to lie on a vertex or a side, then the procedure can terminate immediately (without consideration of any remaining sides) and conclude that P lies on the boundary of the polygon. Otherwise, each and every side of the polygon must be considered in turn, and the number of times that arc XP crosses the boundary of the polygon must be counted.

Several subtleties must be observed when implementing this algorithm. First, the entire approach breaks down if the arbitrary point P happens to be antipodal to point X. In this special case, a unique minor arc XP does not exist. Instead, an infinite number of great circle arcs join X and P. This condition can be recognized and trapped; nevertheless, the location of P relative to S will remain undetermined. In most cases, the fact that the algorithm cannot handle one particular location for P will be of no practical importance (as long as this fact is signified). By providing a second point X_2 known to be located inside S, and reapplying the algorithm to the problem point, this shortcoming is circumvented. (P cannot be antipodal to both X and X_2.)

A second subtlety concerns the special case where arc XP passes exactly through a vertex. The algorithm must recognize that arc XP can be considered to have crossed either, but not both, of the sides sharing that vertex; otherwise, the total crossing count will be in error. This provision can be taken into account during the test for necessary strike: the test is passed if ϕ'_P equals ϕ'_A or lies in the range of longitudes between A and B, but not including ϕ'_B. Consider a case in which arc XP passes through vertex V_i. When the side between V_{i-1} and V_i is being examined, and vertex V_i is identified with B, the condition of necessary strike is not satisfied, and arc XP is found not to cross this side. However, on examining the next side, that joining vertices V_i and V_{i+1}, vertex V_i will be identified with A, and this time the necessary strike condition will be satisfied. The second test will then go into effect and a crossing will be detected. Thus, one crossing is counted when both sides have been considered.

The algorithm breaks down when side AB lies along arc XP. In this situation, the concept of arc XP crossing arc AB becomes poorly defined. This problematic configuration can occur only if point X lies on the great circle passing through both A and B. The problem is avoided easily if X is forbidden to lie on any great circle that passes through any neighboring pair of vertices. In practice, this restriction on the location of X within S rarely will inconvenience the user. Of course, any computer code implementing this algorithm must check that this restriction has been met. Detecting a violation of this restriction is simple in the coordinate system utilized for the test of necessary strike. If, in a coordinate system in which point X acts as the north pole ($\lambda' = 90°$), vertices A and B have the same longitude ($\phi'_A = \phi'_B$), then points X, A, and B must lie on a single great circle.

A FORTRAN IMPLEMENTATION OF THE ALGORITHM

The algorithm described above has been implemented as a set of subroutines coded in FORTRAN (Appendix A). The code conforms to the FORTRAN-77 standard, except that one or two common extensions to this language (such as END DO) are used. These extensions are supported by most FORTRAN-77 compilers. The subroutine package consists of four subroutines. The first pair of subroutines (DefSPolyBndry and LctPtRelBndry) are called by the user from his main or driver program. The remaining subroutines (TrnsfmLon and EastOrWest) are called by subroutines DefSPolyBndry and LctPtRelBndry, and should not be referenced by the user's main program.

Most applications that call for a point-in-spherical-polygon algorithm involve establishing a small number of polygonal boundaries (often just one), and then processing large numbers of points to find which points are inside those boundaries. Given this pattern of usage, any quantities that depend only on the position of the polygon and interior point X should be computed just once, and not repeated each time a new point P is considered. For this reason, two subroutines are provided to the user to solve the point-in-spherical-polygon problem. First, the user's program calls subroutine DefSPolyBndry to define the spherical polygonal boundary and to specify the location of the interior point X. The user's program then calls subroutine LctPtRelBndry to determine the location of any point (P) relative to the boundary. Normally, DefSPolyBndry will be called once, and subsequently LctPtRelBndry will be called many times.

Subroutine DefSPolyBndry performs several functions. It computes "longitudes" of each of the polygon's vertices in a coordinate system in which point X acts as the north pole ($\lambda' = 90°$), and stores this information, together with coordinates of the vertices and point X in the original coordinate system, in a named common block. This information is available to subroutine LctPtRelBndry which shares this named common block. DefSPolyBndry also checks for several possible error conditions. First, it ensures that sufficient storage is available to solve the problem. (The maximum allowable number of polygon sides can be adjusted by editing the value assigned to parameter *mxnv*.) It checks that all neighboring vertices are distinct (including the first and last vertices). It checks that no neighboring pair of vertices are antipodal, and that point X does not lie on the great circle projection of any polygon side. DefSPolyBndry also sets a flag in the named common block to indicate that it has been called (at least once).

The user's main program calls subroutine LctPtRelBndry to determine if some point *P*, whose coordinates are passed through the argument list, is inside, outside, or on the boundary of the polygon previously defined. LctPtRelBndry obtains any necessary information about the location of the polygon and point *X* through the common block named spolybndry. LctPtRelBndry first checks that a polygon has been defined by a previous call to DefSPolyBndry. It then checks that points *P* and *X* are not antipodal. (If they are, it flags this problem and returns control to the main program without solving the problem of *P*'s location relative to *S*.) LctPtRelBndry then processes each polygon side in turn. Each side is tested for the condition of necessary strike. In the event that this test is passed, it determines if points *X* and *P* lie on the same side of the polygon side (no crossing), on different sides (a crossing), or neither (*P* lies on the polygon side). If *P* is determined to lie on a side of *S*, the problem is solved and the subroutine terminates. Otherwise, all polygon sides are considered, and the total number of crossings is determined. The problem is solved, and the subroutine returns control to the main program.

Subroutines TrnsfmLon and EastOrWest are also listed (Appendix A). Subroutine TrnsfmLon is required by subroutines DefSPolyBndry and LctPtRelBndry to perform the coordinate transformation produced by moving the location of the north pole. This transformation is discussed in Bevis and Cambareri (1987). Subroutine EastOrWest is required by subroutine LctPtRelBndry. Given the longitudes of two points, it determines if the second point lies east, west, or neither east nor west of the first point.

USING COMPOUND POLYGONS

A polygon containing one or more holes can be defined as a single entity (Fig. 4a). For example, this situation might arise when large islands such as Sicily are excluded from a polygon that represents the Mediterranean Sea. Similarly, a suite of polygons can be defined as a single entity (Fig. 4b). For example, a chain of islands can be represented as a single polygon. Entities of this kind (Fig. 4a, b) are called compound polygons (Bevis and Cambareri, 1987). In order to describe a polygon containing a hole (Fig. 4a), the inner and outer boundaries are joined by a corridor of zero width; by treating both of these boundaries as parts of a single and continuous boundary, the shaded polygon (Fig. 4a) is described as a polygon with 14 vertices and sides. Vertices 4 and 11 are coincident, as are vertices 5 and 10. A similar device is used to represent a suite of polygons as a single polygon (Fig. 4b). In this way, a chain of islands is defined as a single geographical entity in a single call to DefSPolyBndry, and subsequently a single call to LctPtRelBndry will determine if any point *P* lies within the island chain.

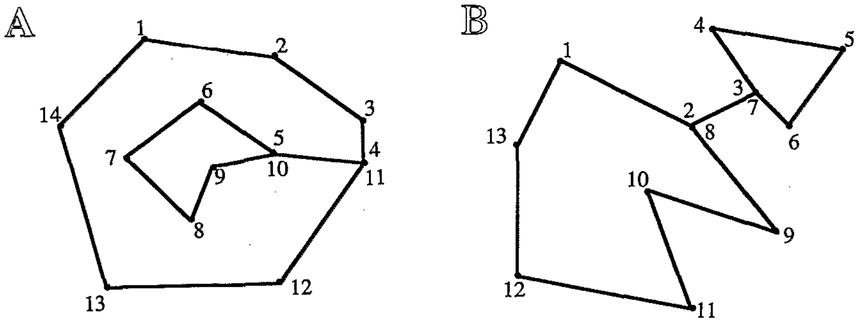

Fig. 4. Examples of compound polygons, showing how they are described for use with the point-in-spherical-polygon algorithm. (A) This polygon contains a hole. The inner and outer walls of the (shaded) polygon are joined by a corridor of zero width. One of the sides of this corridor joins vertices 4 and 5, the other joins vertices 10 and 11. Vertices 4 and 11 are coincident, as are vertices 5 and 10. Although sides 4-5 and 10-11 touch, they do not cross. Note that the compound polygon has 14 sides, none of which cross each other. (B) By a similar device, a suite of polygons may be treated as a single polygon. The composite polygon shown can be treated as a single spherical polygon with 13 sides, although in reality it consists of two separate (shaded) areas. These areas are joined by a corridor of zero width.

A PRACTICAL EXAMPLE

The application that initially prompted development of the algorithm presented here was that of sorting earthquakes in a seismicity catalog on the basis of their location. The case study described in this section was the first practical application undertaken, after initial debugging and testing of the computer codes. A network of 19 seismograph stations was established in the central New Hebrides island arc (Vanuatu) in 1978–1979, as part of a joint project of Cornell University and ORSTOM. By mid 1987 this network had been used to locate over 17,000 local earthquakes. Chatelain et al. (1986), in a study of the space and time distribution of seismicity in this area, identified four regions of large "background" seismicity (Fig. 5b). The precise significance of these boundaries, and the manner in which they were determined, need not concern the reader. In 1987, as part of an ongoing program of data analysis, all events located in the four previously identified zones were desired to be extracted. The boundaries had been digitized in the earlier study and were readily available.

These boundaries were concatenated, as described in the previous section, so as to form a single compound polygon. Corridors of zero width joining the four simple polygons are visible (Fig. 5b). This compound polygon contains 1071 vertices. The data set to be sorted consisted of 17,087 hypocenters (Fig. 5a). The application program defined the boundary by a single call to Def-

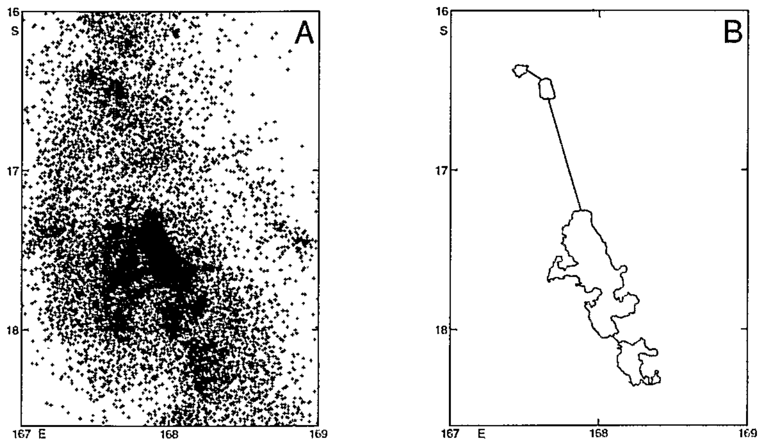

Fig. 5. (A) A map showing epicenters of 17,087 earthquakes located by the ORSTOM/Cornell network in Vanuatu. (B) A composite spherical polygon consisting of four areas of interest to seismologists managing this data set. This polygon has 1071 sides. All the earthquakes in (A) are sorted according to whether or not they fall within the boundary shown in (B).

SPolyBndry, and then called LctPtRelBndry 17,087 times to determine the location of each epicenter in turn. Epicenters that were found to lie exactly on the boundary were treated as "outside" points so as not to admit data that happened to lie along the corridors. The sorted data indicate 6,160 hypocenters are inside the boundary (Fig. 6b), whereas the remaining 10,927 hypocenters (Fig. 6a) are outside.

MULTISTAGE SORTING

Because the basic algorithm examines every side of the polygonal boundary each time some point P is located relative to the boundary, sorting large numbers of points using a polygon containing many sides is time-consuming. For example, the sort described above took just over 64 min to perform on a VAX-11/750 running VMS (with a moderate user load). Sorting times can be reduced by more than an order of magnitude in situations of this kind by implementing a multistage sort. Multistage sorting is utilized commonly in the context of spatial sorting relative to plane polygonal boundaries (Davis and David, 1980; or almost any computer graphics textbook), and this strategy is carried over easily to the spherical environment. Suppose a large number of points must be sorted relative to some polygon S that contains a large number of sides (Fig. 7). Two new polygons, I and O (Fig. 7), each containing a small number of sides compared to S, are chosen such that I lies close to but every-

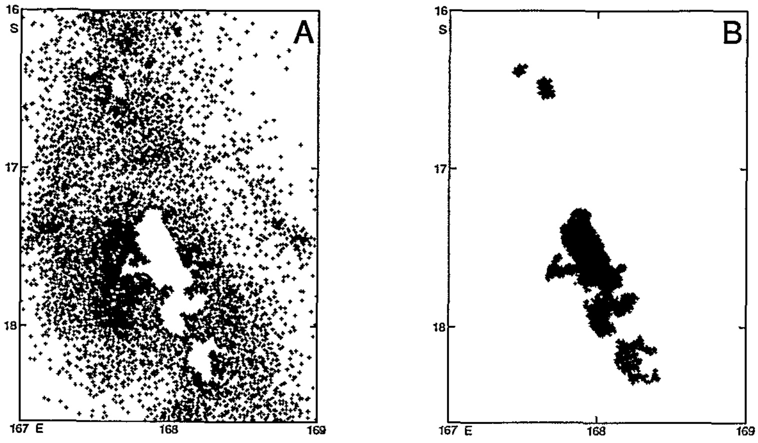

Fig. 6. Maps showing epicenters that fall (A) outside, and (B) inside, the composite spherical polygon shown in the previous figure.

where within S, and O lies close to and everywhere outside of S. The goal is to have O completely surround S and I to be completely contained by S, and to minimize the area between O and I, but keeping the number of vertices in O and I small compared to the number in S.

The principle of the multistage sort is straightforward. Given some point P, one first checks to see if it lies outside O. This is a computationally inexpensive task because O has few vertices. If P lies outside O, then clearly it must lie outside S, and the problem is solved. If P is found to lie inside O, a second test is performed to determine if P lies inside I. Again this is computationally inexpensive. If P lies inside I, then it must lie inside S, and the problem is

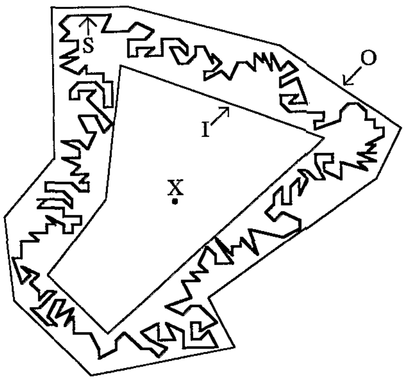

Fig. 7. The spherical polygon S has a large number of sides. Boundaries O and I have fewer sides. O lies completely outside S, and I lies completely inside S. Clearly any point lying outside O also lies outside S, and any point lying inside I also lies inside S.

solved. In a small number of cases, *P* lies inside *O* but outside *I*, and so whether or not *P* lies inside *S* has not been determined. In this case, the point-in-spherical-polygon algorithm is employed directly to solve for the location of *P* relative to *S*. An expensive computation is performed only in the event that this third test is necessary.

Implementation of the multistage sort has to be modified slightly from that described above in order to use codes provided (Appendix A). This is because the codes are structured so as to be most effective when a polygon is defined once (using DefSpolyBndry) and many points are subsequently located against that polygon (using LctPtRelBndry). A sequence of steps, such as define *O*, locate *P* relative to *O*, define *I*, locate *P* relative to *I*, read next *P*, define *O* again, locate *P* relative to *O*, *etc.*, is undesirable because *O* and *I* (and perhaps *S*) would be defined many times. In this case, the computations performed by DefSPolyBndry would be performed repetetively and redundantly. However, the multistage sort can be reorganized so as to ensure that each polygon is defined only once. The application program reads the coordinates of all points into memory, and establishes a flag for each point that can be set to one of three values, signifying (i) *P* inside *S*, (ii) *P* outside *S*, and (iii) location of *P* relative to *S* not yet determined. The polygon *O* is defined (once), and the program loops over all points and determines the location of each point relative to *O*. The results are stored in the flag array. If a point is outside of O, it is flagged as lying outside of *S*; otherwise, its location relative to *S* is undetermined. Then polygon *I* is defined and a second loop is executed in which all points as yet not located relative to *S* are located relative to *I*. If any such point is inside *I*, its flag is reset to indicate that it is inside *S*. These tasks are performed rapidly because the number of sides in *O* and *I* are small. Finally, polygon *S* is defined. The program then loops over each point, checks the flags to identify every point as yet unlocated relative to *S*, and applies the point-in-polygon-algorithm directly, explicitly determining the location of the point relative to *S*.

The procedure discussed above, in which all points are located relative to *O*, and subsequently a subset of these points are located relative to *I*, and finally a subset of these points are located relative to *S*, reflects the structure of the subroutines as listed (Appendix A). The codes could be modified so that each point could be located relative to *O*, and if necessary relative to *I*, and if necessary relative to *S*, prior to consideration of the next point. This approach would minimize the use of program storage. However, all information currently passed (from DefPolyBndry to LctPtRelBndry) through the named common block, would have to be passed through LctPtRelBndry's argument list instead, so that points could be located against polygons *O*, *I*, or *S* in any sequence, without the need to define *O*, *I*, or *S* more than once. The subroutines presented here are not structured this way because the expanded argument list is cumbersome to use, and results in a rather ugly code.

DISCUSSION

Subroutines presented here have been employed in several real-world contexts and, from a practical point of view, their performance has been satisfactory. The requirements that point *X* not lie on a great circle joining any neighboring vertices of the polygonal boundary, and that *X* not be antipodal to point *P*, have only once forced a change in the position initially assigned to point *X*. Nevertheless, from the viewpoint of the geomathematician, these limitations manifest a certain inelegance inherent to the algorithm. Indeed, nearly all awkward aspects of this algorithm largely derive from the choice of point *X* and its relationship to other points in the problem. This suggests an avenue for the future improvement of point-in-spherical-polygon algorithms.

Consider the related problem of computing the area of a spherical polygon of arbitrary shape. Algorithms can be devised that solve this problem with reference to some point *X* that lies within the boundary under consideration. However, this problem can be solved without reference to any point lying within the polygon. Bevis and Cambareri (1987) presented an algorithm for computing the area of a spherical polygon that requires as input only the coordinates of the vertices of the polygon. They adopted a convention whereby the direction of vertex enumeration flagged which of the spherical polygons enclosed by the boundary was the one under consideration. Their algorithm is leaner and more elegant than the one presented here. Undoubtedly, a point-in-spherical-polygon algorithm could be developed that requires only coordinates of the polygon's vertices (and not the location of some point *X* lying with the polygon); an algorithm in this class would eliminate the restrictions (and much of the special case handling) associated with the algorithm presented here. The reader is invited to develop this new class of algorithm.

APPENDIX A

```
c Given some spherical polygon S and some point X known to be located inside S, these routines
c will determine if an arbitrary point P lies inside S, outside S, or on its boundary.The calling
c program must first call DefSPolyBndry to define the boundary of S and the point X. Any
c subsequent call to subroutine LctPtRelBndry will determine if some point P lies inside or
c outside S; or on its boundary. (Usually DefSPolyBndry is called once, then LctPrRelBndry is
c called many times).

c REFERENCE:     Bevis, M. and Chatelain, J.-L. (1989)
c                Mathematical Geology, vol 21.

c VERSION 1.0

c*********************************************************************
      Subroutine DefSPolyBndry(vlat,vlon,nv,xlat,xlon)
c*********************************************************************
c This main entry point is used to define the spherical polygon S and the point X.
```

```
c ARGUMENTS:
c  vlat,vlon (sent) ...  vectors containing the latitude and longitude of each vertex of the spherical
c                        polygon S. The ith. vertex is located at [vlat(i),vlon(i)].
c  nv        (sent) ...  the number of vertices and sides in the spherical polygon S
c  xlat,xlon (sent) ...  latitude and longitude of some point X located inside S. X must not be
c                        located on any great circle that includes two vertices of S.

c UNITS AND SIGN CONVENTION:
c  Latitudes and longitudes are specified in degrees.
c  Latitudes are positive to the north and negative to the south.
c  Longitudes are positive to the east and negative to the west.

c VERTEX ENUMERATION:
c  The vertices of S should be numbered sequentially around the border of the spherical polygon.
c  Vertex 1 lies between vertex nv and vertex 2. Neighbouring vertices must be seperated by less
c  than 180 degrees. (In order to generate a polygon side whose arc length equals
c  or exceeds 180 degrees simply introduce an additional (pseudo)vertex ).
c  Having chosen vertex 1, the user may number the remaining vertices in either direction.
c  However if the user wishes to use the subroutine SPA to determine the area of the polygon S
c  (Bevis & Cambareri, 1987, Math. Geol., v.19, p. 335-346) then he or she must follow the
c  convention whereby in moving around the polygon border in the direction of increasing
c  vertex number clockwise bends occur at salient vertices. A  vertex is salient if the interior angle
c  is less than 180 degrees. (In the case of a convex polygon this convention implies that vertices
c  are numbered in clockwise sequence).

      implicit none
      integer mxnv,nv
c..............................................................................
c Edit next statement to increase maximum number of vertices that may be
c used to define the spherical polygon S
      parameter (mxnv=500)
c The value of parameter mxnv in subroutine LctPtRelBndry must match that
c of parameter mxnv in this subroutine, as assigned above.
c..............................................................................
      real*8 vlat(nv),vlon(nv),xlat,xlon,dellon
      real*8 tlonv(mxnv),vlat_c(mxnv),vlon_c(mxnv),xlat_c,xlon_c
      integer i,ibndry,nv_c,ip
      data ibndry /0/

      common /spolybndry/vlat_c,vlon_c,nv_c,xlat_c,xlon_c,tlonv,ibndry

      if(nv.gt.mxnv)then
        print *,'nv exceeds maximum allowed value'
        print *,'adjust parameter mxnv in subroutine DefSPolyBndry'
        stop
      end if

      ibndry=1                 ! boundary defined at least once (flag)

      nv_c=nv                  ! copy for named common
      xlat_c=xlat              !    "     "
      xlon_c=xlon              !    "     "

      do i=1,nv

        vlat_c(i)=vlat(i)      !    "     "
        vlon_c(i)=vlon(i)      !    "     "

        call TrnsfmLon(xlat,xlon,vlat(i),vlon(i),tlonv(i))
      if(i.gt.1)then
       ip=i-1
      else
       ip=nv
      end if

      if(vlat(i).eq.vlat(ip) .and. vlon(i).eq.vlon(ip))then
       print *,'DefSPolyBndry detects user error:'
        print *,'vertices ',i,' and ',ip,' are not distinct'
       stop
      end if

      if(tlonv(i).eq.tlonv(ip))then
       print *,'DefSPolyBndry detects user error:'
        print *,'vertices ',i,' & ',ip,' on same gt. circle as X'
       stop
      end if

      if(vlat(i).eq.-vlat(ip))then
       dellon=vlon(i)-vlon(ip)
       if(dellon.gt.+180.)dellon=dellon-360.
         if(dellon.lt.-180.)dellon=dellon-360.
         if(dellon.eq.+180.0 .or. dellon.eq.-180.0)then
           print *,'DefSPolyBndry detects user error:'
            print *,'vertices ',i,' and ',ip,' are antipodal'
           stop
         end if
       end if

      end do

      return
      end


c***********************************************************************
      Subroutine LctPtRelBndry(plat,plon,location)
c***********************************************************************
c This routine is used to see if some point P is located inside, outside or on the boundary of the
c spherical polygon S previously defined by a call to subroutine DefSPolyBndry. There is a
c single restriction on point P: it must not be antipodal to the point X defined in the call to
c DefSPolyBndry (ie.P and X cannot be seperated by exactly 180 degrees).

c ARGUMENTS:
c plat,plon  (sent)... the latitude and longitude of point P
c location (returned)... specifies the location of P:
c                        location=0 implies P is outside of S
c                        location=1 implies P is inside of S
c                        location=2 implies P on boundary of S
c                        location=3 implies user error  (P is antipodal to X)

c UNITS AND SIGN CONVENTION:
c  Latitudes and longitudes are specified in degrees.
c  Latitudes are positive to the north and negative to the south.
c  Longitudes are positive to the east and negative to the west.

      implicit none
      integer mxnv
```

```
c.............................................................................................
c The statement below must match that in subroutine DefSPolyBndry
      parameter (mxnv=500)
c.............................................................................................

      real*8 tlonv(mxnv),vlat_c(mxnv),vlon_c(mxnv),xlat_c,xlon_c
      real*8 plat,plon,vAlat,vAlon,vBlat,vBlon,tlonA,tlonB,tlonP
      real*8 tlon_X,tlon_P,tlon_B,dellon
      integer i,ibndry,nv_c,location,icross,ibrngAB,ibrngAP,ibrngPB
      integer ibrng_BX,ibrng_BP,istrike

      common /spolybndry/vlat_c,vlon_c,nv_c,xlat_c,xlon_c,tlonv,ibndry

      if(ibndry.eq.0)then              ! user has never defined the bndry
       print *,'Subroutine LctPtRelBndry detects user error:'
       print *,'Subroutine DefSPolyBndry must be called before'
       print *,'subroutine LctPtRelBndry can be called'
       stop
      end if

      if(plat.eq.-xlat_c)then
       dellon=plon-xlon_c
       if(dellon.lt.-180.)dellon=dellon+360.
       if(dellon.gt.+180.)dellon=dellon-360.
       if(dellon.eq.+180.0 .or. dellon.eq.-180.)then
        print *,'Warning: LctPtRelBndry detects case P antipodal to X'
        print *,'location of P relative to S is undetermined'
        location=3
        return
       end if
      end if

      location=0      ! default ( P is outside S)
      icross=0        ! initialize counter

      if(plat.eq.xlat_c .and. plon.eq.xlon_c)then
       location=1
       return
      end if

      call TrnsfmLon(xlat_c,xlon_c,plat,plon,tlonP)

      do i=1,nv_c              ! start of loop over sides of S

       vAlat=vlat_c(i)
       vAlon=vlon_c(i)
       tlonA=tlonv(i)

       if(i.lt.nv_c)then
        vBlat=vlat_c(i+1)
        vBlon=vlon_c(i+1)
        tlonB=tlonv(i+1)
       else
        vBlat=vlat_c(1)
        vBlon=vlon_c(1)
        tlonB=tlonv(1)
       end if

       istrike=0
```

```
          if(tlonP.eq.tlonA)then
            istrike=1
          else
            call EastOrWest(tlonA,tlonB,ibrngAB)
            call EastOrWest(tlonA,tlonP,ibrngAP)
            call EastOrWest(tlonP,tlonB,ibrngPB)
            if(ibrngAP.eq.ibrngAB .and. ibrngPB.eq.ibrngAB)istrike=1
          end if

          if(istrike.eq.1)then

            if(plat.eq.vAlat .and. plon.eq.vAlon)then
              location=2          ! P lies on a vertex of S
              return
            end if

            call TrnsfmLon(vAlat,vAlon,xlat_c,xlon_c,tlon_X)
            call TrnsfmLon(vAlat,vAlon,vBlat,vBlon,tlon_B)
            call TrnsfmLon(vAlat,vAlon,plat,plon,tlon_P)

            if(tlon_P.eq.tlon_B)then
              location=2          ! P lies on side of S
              return
            else
              call EastOrWest(tlon_B,tlon_X,ibrng_BX)
              call EastOrWest(tlon_B,tlon_P,ibrng_BP)
              if(ibrng_BX.eq.(-ibrng_BP))icross=icross+1
            end if

          end if

        end do                    ! end of loop over the sides of S

c if the arc XP crosses the boundary S an even number of times then P
c is in S
        if( jmod(icross,2).eq.0 )location=1

        return
        end


c------------------------------------------------------------------------------------------------
      subroutine TrnsfmLon(plat,plon,qlat,qlon,tranlon)
c This subroutine is required by subroutines DefSPolyBndry & LctPtRelBndry. It finds the
c 'longitude' of point Q in a geographic coordinate system for which point P acts as a 'north
c  pole'. SENT: plat,plon,qlat,qlon, in degrees. RETURNED: tranlon, in degrees.
        implicit none
        real*8 pi,dtr,plat,plon,qlat,qlon,tranlon,t,b
        parameter (pi=3.141592654d0,dtr=pi/180.0d0)
        if(plat.eq.90.)then
          tranlon=qlon
        else
       t=dsin((qlon-plon)*dtr)*dcos(qlat*dtr)
       b=dsin(dtr*qlat)*dcos(plat*dtr)-dcos(qlat*dtr)*dsin(plat*dtr)*
     &      dcos((qlon-plon)*dtr)
       tranlon=datan2(t,b)/dtr
        end if
      return
      end
```

```
c-----------------------------------------------------------------------------------------------------------
        subroutine EastOrWest(clon,dlon,ibrng)
c This subroutine is required by subroutine LctPtRelBndry. This routine determines if in
c travelling the shortest path from point C (at longitude clon) to point D (at longitude dlon)
c one is heading east, west or neither.
c SENT: clon,dlon; in degrees. RETURNED: ibrng (1=east,-1=west, 0=neither).
        implicit none
        real*8 clon,dlon,del
        integer ibrng
        del=dlon-clon
        if(del.gt.180.)del=del-360.
        if(del.lt.-180.)del=del+360.
        if(del.gt.0.0 .and. del.ne.180.)then
          ibrng=-1       ! (D is west of C)
        else if(del.lt.0.0 .and. del.ne.-180.)then
          ibrng=+1       ! (D is east of C)
        else
          ibrng=0        ! (D north or south of C)
        end if
        return
        end
c-----------------------------------------------------------------------------------------------------------
```

REFERENCES

Bevis, M., and Cambareri, G., 1987, Computing the Area of a Spherical Polygon of Arbitrary Shape: Math. Geol., v. 19, p. 335–346.

Chatelain, J. L., Isacks, B. L., Cardwell, R. K., Prévot, R., and Bevis, M., 1986, Patterns of Seismicity Associated with Asperities in the Central New Hebrides Island Arc: J. Geophys. Res., v. 91, p. 12497–12519.

Davis, M. W., and David, M., 1980, An Algorithm for Finding the Position of a Point Relative to a Fixed Polygonal Boundary: Math. Geol., v. 12, p. 61–68.

Hall, J. K., 1975, PTLOC: A FORTRAN Subroutine for Determining the Position of a Point Relative to a Closed Boundary: Math. Geol., v. 7, p. 75–79.

Salomon, K. B., 1978, An Efficient Point-in-polygon Algorithm: Comput. Geosci., v. 4, p. 173–178.