An Exact Method for Finding Shortest Routes on a Sphere, Avoiding Obstacles

Alan Washburn, Gerald G. Brown

Operations Research Department, Naval Postgraduate School, Monterey, California 93943

Received 2 April 2016; revised 28 July 2016; accepted 1 August 2016
DOI 10.1002/nav.21702
Published online 22 August 2016 in Wiley Online Library (wileyonlinelibrary.com).

Abstract: On the surface of a sphere, we take as inputs two points, neither of them contained in any of a number of spherical polygon obstacles, and quickly find the shortest route connecting these two points while avoiding any obstacle. The WetRoute method presented here has been adopted by the US Navy for several applications. Naval Research Logistics 63: 374–385, 2016

Keywords: WetRoute; spherical mathematics; finding shortest routes

1. INTRODUCTION

Finding a shortest path (or route) is one of the most commonly solved network problems. Arguably, this history starts with the introduction of graph theory by Euler in 1758 [6] and continues today, with perhaps the most widely known intermediate result being that of Dijkstra in 1959 [5] for finding shortest total length paths in a network (i.e., a graph with numerical attributes, here non-negative edge lengths). The transitive significance of Dijkstra's work, and that of others in that era, is the formality of specifying an algorithm in terms of an input and an output, proving its correctness, and implementing the algorithm on a digital computer.

Our subject here is finding shortest routes, but not on a network. We are given two arbitrary "wet points" X and Y on a spherical Earth that is cluttered with dry obstacles, and asked to find the shortest route that does not run over an obstacle. The wet-dry terminology is due to the motivating application where a ship at maritime point X wants to get to some other maritime point Y, traveling a minimum distance without running aground. Neither X nor Y can be confined to any predetermined finite set, and therefore cannot be embedded in any finite network. In spite of this difficulty, these shortest routes must be found quickly. This is because the actual computational problem given to WetRoute is to find the shortest distance between every pair of destinations taken from a list of on the order of 100 of them. WetRoute is fast enough on a modern computer to find all of these shortest routes in a few seconds.

Correspondence to: Alan Washburn (awashburn@nps.edu)

One could, of course, superimpose a mesh of some kind on the wet part of Earth, compute and store the apparent shortest route from each node of the mesh to other nodes, and then move X and Y to their closest nodes in order to quickly retrieve the approximately shortest route. We do not argue that doing so would be unreasonable; in fact, the mesh method has important advantages over the method that we are about to describe. Chief among these is that the "distance" measure could actually depend on distance, fuel, time, and risk, all of which can have different minima in the face of currents, winds, politics, and storms. We have been tempted by that method, but ultimately rejected it. The approximations required in the mesh method would be problematic. Earth's oceans cover about 362,000,000 km^2. If one were to partition this region with equilateral triangles with 5 km sides, the number of triangles required would be about 29,000,000, and the number of "distances" to be computed and stored would be the square of that number. Each of those computations would be a significant task in itself, and the implied storage requirement exceeds the capability of most computers. One would be tempted to make the triangles larger, thus making the approximation worse. The advantage of WetRoute is that it does not need to begin by artificially moving X and Y to a node in a pre-established mesh, and therefore avoids the associated need for approximation.

The motivating application for WetRoute is the US Navy's Replenishment at Sea Planner (RASP) [15] that recommends when and where supply ships should load fuel and stores and when and where to rendezvous with underway US and coalition partner combatants (customers) for at-sea replenishment. The track and supply status of each customer is given, with

RASP advising the activities of the supply ships. RASP's problem is to minimize the cost of supplying the customers over a given time horizon, essentially a multiple traveling salesman problem with time windows, moving customers and multiple depots. The customers can be at many locations over the time horizon, and supply ships may need to travel from one customer location to another. WetRoute must therefore calculate thousands of shortest routes prior to RASP doing its minimization, hence the emphasis on computational speed.

Fuel consumption is a dominant concern. In early development of RASP, an approximating route planner introduced a host of complications, and these were exacerbated when unanticipated changes to existing schedules were needed. Planners needed to quickly find a minimal number of events to reschedule, while still meeting as many prior scheduled events as possible. To do this, the only degree of freedom was to speed up the supply ships, and ship fuel consumption is a strongly increasing function of speed. Quickly dealing with these revisions based on approximate, tabulated distances, or on the visual thumb rules of a Navy Lieutenant scheduler, demonstrated the need for a better distance estimation tool like WetRoute.

2. INITIAL OBSERVATIONS

2.1. Notation and Terminology

Our abstract Earth is a perfect sphere, has unit radius, and is cluttered with "obstacles", each of which is a simple polygon described by at least three distinct vertex points in clockwise order, with an interior that is not empty. A model of Earth that is reasonably accurate for navigational purposes will have about 900 vertices (see Fig. 1). For vertex *i*, we will use the notation *i.next* and *i.previous* for the next and previous vertex as the obstacle boundary is traversed clockwise.

Each pair of adjacent vertices (*i, i.next*) is connected by a segment of a great circle, the length of which is the shortest distance between the two vertices. Such great circle segments always have a length in the open interval $(0, \pi)$, and will be referred to simply as "segments" in the following. The vertices of the obstacles are all assumed to be distinct, and we also assume that the obstacles do not intersect or even touch each other. All points in the interior of an obstacle or on one of its boundary segments are "dry", whereas points that are not part of any obstacle are "wet". The interiors of these obstacles must be avoided in the process of getting from one wet point to another, but the boundaries can be included in the route. Wet points like *X* and *Y* that are the subject of shortest route calculations will be referred to as "origins" and "destinations". Fagerholt, Heimdal and Lotku [7] describes a similar problem in the plane where the destination is necessarily one of the obstacle vertices. Here *Y*, like *X*, is an arbitrary wet point—our problem is a "two-point query".

Figure 1. Showing in Google Earth the level of detail one can achieve with about 900 world-wide vertices. Note that most small Caribbean islands are not counted as obstacles. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

Earth has an important property that we will assume true of our spherical abstraction. The largest obstacle on Earth (Eurasia with Africa still attached, which we have named EurAfrica) can still be contained in a hemisphere; i.e., all obstacles are "small" in that sense. One can imagine obstacles that are not small, an example being one of the two pieces of leather that cover a baseball. Such obstacles are not contained in any hemisphere, and therefore cannot be seen in their entirety from any viewpoint outside the sphere. Nor do they have convex hulls, unless the entire sphere is taken to be "the smallest convex region that contains the obstacle" [9]. Conveniently, there are no such obstacles on Earth. We will therefore assume that all obstacles are small, and the word "clockwise" used above should be understood to be from the point of view of an observer outside the sphere who is looking at the entire obstacle. For any such obstacle, take a rubber band and stretch it along the edge of a containing hemisphere. Start the rubber band shrinking toward the obstacle, but keep it out of the obstacle by marking the obstacle's vertices with pins. Just as in the plane, the rubber band will settle at the convex hull of the obstacle. The convex hull and the obstacle will share some segments, and these segments are by definition clockwise in the obstacle if and only if they are clockwise in the convex hull. For this reason, a clockwise tour of EurAfrica might appear to be going counterclockwise after it encounters the Mediterranean at the southern end of the Straits of Gibraltar.

WetRoute locates points on Earth such as *X* by giving their latitude and longitude, symbolically *X.lat* and *X.lon*. To avoid repeated evaluations of sines and cosines, properties of a point *X* also include *X.slat, X.clat, X.slon*, and *X.clon*, those four numbers being sin(*X.lat*), cos(*X.lat*), sin(*X.lon*), and cos($X.lon$), respectively. We say that destination Y is "visible" from origin X (or vice versa) if the segment connecting X to Y does not include an interior point from any obstacle. That connecting segment is unique unless X and Y are antipodes. Should it happen that X and Y are antipodes, we say that Y is visible from X if *any* connecting segment avoids all obstacle interiors.

2.2. Spherical Mathematics

The need for spherical mathematics is unfortunate, since visualization, and computation are easier in the plane. However, because the origin and destination of a ship routing problem can be thousands of miles apart, the subject cannot be avoided. Visualization is indeed a difficulty, but the calculations need not be hindered by the incessant need for trigonometric calculations as long as points have the properties specified above. For example, the expression used in WetRoute for the cosine cd of the distance between X and Y is

$$cd = (X.slat)(Y.slat) + (X.clat)(Y.clat) \times ((X.clon)(Y.clon) + (X.slon)(Y.slon)),$$

an expression that involves only multiplication and addition. This formula is the Law of Cosines for Sides (e.g., [10]). The arc cosine function is required if one needs to know the actual distance, but even this can be avoided if one is merely comparing distances, since the arc cosine function is monotonic on the interval $[-1, 1]$.

The direction from X to Y depends on how angles are measured. We measure angles in radians clockwise from the North Pole (NP), and by "direction from X to Y" mean the angle from the (X, NP) segment to the (X, Y) segment. This is the initial course a ship would take in moving along the (X, Y) segment. In nautical terms the angle from X to Y is the "bearing" from X to Y. Like distances, directions can be computed without extensive use of trigonometric functions. Appendix B includes the details of this and certain other common calculations.

Most spherical problems have topologically similar planar counterparts, especially when obstacles are small, which makes it possible to illustrate them. An exception to this is any problem that involves antipodes (antipodes are the only problem—absent the antipode of X, the sphere is topologically homeomorphic to the plane). A ray from a point does not extend to infinity on the sphere, but rather to the point's antipode on the other side of the sphere; in fact, *all* rays from a point go through the point's antipode. The direction from a point to its antipode is ill-defined, and there are also some other inconvenient features that will be addressed later. One advantage of dealing with obstacles that are small is that the antipode of any point in a given obstacle cannot be in the same obstacle. The antipode of a wet point, however, can be wet.

2.3. Basic Algorithm

If Y is visible from X, then the shortest route is simply the segment that connects them. If Y is not visible from X, then the shortest route will go from X to some visible vertex i, then from vertex to vertex until some vertex j is encountered from which Y is visible, and finally from j to Y. Vertices i and j may or may not be the same vertex. A proof of these claims for the plane can be found in [16]. Since the triangle inequality holds on the sphere, just as it does in the plane, a similar proof (omitted here) holds on the sphere. Let S_X and S_Y be the sets of "candidate" vertices for X and Y, respectively. For the moment these can be thought of as sets of vertices that are visible from each of those wet points, although we will later show that they can be substantially reduced. Neither of these sets can be empty unless Y is visible from X. Let $d(X, Y)$ be the length of the (X, Y) segment, and let $D(X, Y)$ be the shortest distance between X and Y when obstacles are considered. Then

$$D(X, Y) = \begin{cases} d(X, Y) \text{ if } Y \text{ is visible from } X, \text{ or otherwise} \\ \min_{i \in S_X, j \in S_Y} d(X, i) + D(i, j) + d(j, Y) \end{cases} . \tag{1}$$

In WetRoute, the tasks necessary for the determination of $D(X, Y)$ are partitioned into "static" and "dynamic". Static tasks are those that do not require knowledge of X or Y, while dynamic tasks cannot be carried out until X and Y are specified. The principal static task is the determination of the inter-vertex distances $D(i, j)$ for all vertex pairs. This approximately 900×900 matrix is computed, stored, and then recalled whenever (1) is needed to compute $D(X, Y)$. The shortest path from i to j can be recovered from a separate, identically dimensioned matrix that stores the first vertex encountered in the optimal path from one vertex to another. The dynamic task is to determine whether wet point Y is visible from wet point X, and, if not, to determine the candidate sets S_X and S_Y required in (1). The size of these sets is crucial, since (1) is basically just an exhaustive examination. The Cursor algorithm described below in Section 3 quickly determines X-to-Y visibility and the candidate sets S_X and S_Y, after which (1) is employed to determine the optimal route from X to Y.

Although the Cursor algorithm is as applicable to finding the shortest distances between vertices as it is to general wet points, there is no need to determine the inter-vertex distances $D(i, j)$ quickly. That matrix can even involve a generalized notion of "distance", a fact that we will take advantage of in Section 4 when considering passage through the Panama and Suez canals.

3. VISIBILITY AND THE CURSOR ALGORITHM

Determining visibility is in most cases trivial for a human looking at a globe, but is nonetheless a significant computational task. We are given a point X on the surface of the sphere that is not on the boundary of any obstacle. We wish to determine whether X is wet, and, if so, then which vertices and other wet points are visible from X.

3.1. The Border Function

The WetRoute method for determining visibility involves a "Border" function $B(\phi, X)$ about X, for $0 \leq \phi < 2\pi$. The meaning of $B(\phi, X)$ is "the largest distance at which a point with bearing angle ϕ is visible from X". Given the border function, visibility to an arbitrary point Y is easily determined by calculating the distance R and bearing ϕ of Y from X; Y is visible from X if and only if $R \leq B(\phi, X)$. In practice, WetRoute computes an intermediate function $Seg(\phi, X)$ that identifies the visible obstacle segment at angle ϕ, the attractive feature of this function being that it is constant in angular intervals. Distance $B(\phi, X)$ is then the distance from X to $Seg(\phi, X)$. It typically takes on the order of 20 angular intervals to store $Seg(\phi, X)$ over a complete circle. Once $Seg(\phi, X)$ is known, the visibility from X to any other point on the sphere, whether destination or vertex, is easily determined. The initial computation and storage of $Seg(\phi, X)$ amounts to paying a computational setup cost that simplifies subsequent visibility calculations. The heart of the procedure for determining $Seg(\phi, X)$ is the Cursor algorithm, which determines the wetness of X in the process. The Cursor algorithm resembles the planar VisibleVertices algorithm of [4] in being a circular sweep of all the angles surrounding X, but has different computational goals.

3.2. Obstacle Reduction into Chains

Suppressing X in the notation for brevity, define an angle $A(i)$ from X to every vertex i in some obstacle as follows: Arbitrarily select a non-pole starting vertex i_0 and let its angle $A(i_0)$ be the bearing angle of i_0 from X. Then set i to i_0 and go through the vertices of the obstacle in clockwise order. Let θ be the clockwise angle from (X, i) to $(X, i.next)$, a number in the interval $(-\pi, \pi)$, and let $A(i.next) = A(i) + \theta$. Applied iteratively, this relation defines the angles for all vertices in the obstacle, including finally i_0. The meaning of $A(i)$ is in all cases the bearing from X to i, except that angles are not confined to any specific interval by modular arithmetic. This lack of confinement means that the terminal calculation of the angle at i_0 can differ from the initial value. There are three possibilities:

1) $A(i_0)$ may have increased by 2π from its initial value. If so, X is inside the obstacle.
2) $A(i_0)$ may be unchanged from its original value. If so, then neither X nor its antipode is inside the obstacle.
3) $A(i_0)$ may have decreased by 2π. If so, the antipode of X is inside the obstacle.

Three cases suffice because it is not possible for a point and its antipode to both be inside an obstacle when obstacles are small, as we are assuming. Cases 1 and 2 are anticipated by the study of winding angles in the plane [12]. In case 1 the Cursor algorithm terminates because X is not wet. We assume that case 2 holds for the moment. We will return to case 3 in Section 3.5.

An obstacle may have vertices i where $A(i.previous) < A(i) \geq A(i.next)$. Such vertices are by definition "maxima". Similarly "minima" are vertices where $A(i.previous) \geq A(i) < A(i.next)$. As one proceeds clockwise around the obstacle, a maximum can only be followed by a minimum (possibly with several intermediate vertices that are not extremes), and vice versa. The sequence of vertices *clockwise* from a maximum to a minimum is a "chain" that includes its start at the maximum and its end at the minimum. For chain c, we will use the notation *c.start* for the starting vertex and *c.end* for the ending vertex. Because a vertex cannot be both a maximum and a minimum, *c.start* and *c.end* cannot be the same vertex. The angles along a chain are always nonincreasing as one proceeds clockwise through the vertices, and always $A(c.start) > A(c.end)$. The obstacle shown in Fig. 2 has only a single chain of five vertices that extends clockwise from max to min along the inner boundary of the obstacle (recall that the clockwise direction for an obstacle is the same as the clockwise direction for its convex hull, so the outer boundary determines direction).

Chains can never intersect with one another because obstacles never intersect with themselves or one another. Maxima, minima, and chains exist in the same number. All vertices that are not on chains are invisible to X, so the visible vertices are a subset of those belonging to chains (Theorem 2 in Appendix A). The set of obstacles thus spawns a set of chains. The total number of chains depends on X, but is usually a few hundred in WetRoute when all obstacles are considered. The collection of obstacles has now been reduced to a collection of monotonic, non-intersecting chains. The obstacles themselves will play no further role in the Cursor algorithm.

3.3. The Cursor Algorithm Specified

3.3.1. *Definitions and Preliminary Considerations*

Chain c will be said to "cover" angle ϕ if $A(c.end) < \phi + 2k\pi \leq A(c.start)$ for at least one integer k. The addition of k circles to ϕ is necessary because there are no absolute limits to the angles of the vertices in a chain. A chain covers an angular interval if it covers every angle within that interval.

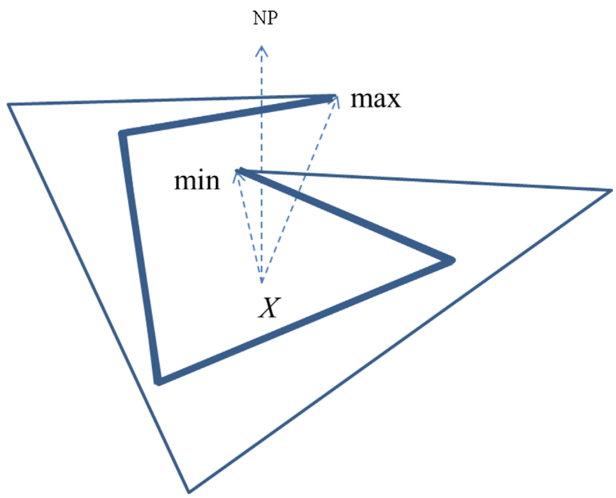

Figure 2. Illustrating an obstacle with eight vertices where A(max) and A(min) differ by more than 2π. A(max) is about 0.5 radians, with angles decreasing along the only chain (four heavy segments) to about -6.4 radians at the min vertex. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

If c covers ϕ, let $L(c, \phi)$ be the length of the segment from X to the closest intersection point with c at angle ϕ. Among all of the chains that cover ϕ, the *dominant* chain at that angle is the one for which $L(c, \phi)$ is the minimum, and the minimum value is the border function $B(\phi, X)$. It is not possible to have ties for dominance because chains never intersect, but it is possible that no chain will cover ϕ, in which case we say that the dominant chain at ϕ is the "White" chain that does not limit visibility.

As a ray from X moves through the vertices of a chain from start to end, its bearing angle will move counterclockwise—as the vertices of a chain are examined in clockwise order (from the viewpoint of the host obstacle), the angle from a wet point to those vertices moves counterclockwise. All chains have a counterclockwise successor. Imagine sitting at X shining a laser at the end of chain c. If you move the laser slightly counterclockwise, the beam will no longer illuminate the end of c. The chain that it does illuminate is the successor of c. Mathematically, the successor of c is the eligible chain x that has the smallest score $L(x, \phi)$ at angle $\phi = A(c.end)$. All chains that do not cover ϕ are ineligible. With one exception, all chains x for which $(A(x.end) - \phi) \bmod 2\pi = 0$ are also ineligible; these are the chains that end at the same angle as c. The exception is that c itself can be eligible if its angular length exceeds 2π—this would be the case in Fig. 2 if there were no other obstacles, since the first segment of the only chain is dominant for angles just counterclockwise of the ray from X to min. In addition, all chains that cover ϕ are eligible as long as they do not end at the same angle as c. If and only if there are no eligible chains, the successor is White—the laser beam will go all the way to the antipode of X. White chains also have a successor, as will be made clear in the next section.

3.3.2. *The Algorithm*

The object of the Cursor algorithm is to find the dominant chain at every angle in the interval $[0, 2\pi)$, and in the process to quantify the border function. Angles for which c is dominant are described as "marked" with c. If there are no chains except for White, then all angles are marked White and the algorithm terminates with the border function being π everywhere (the whole sphere is visible). Otherwise the algorithm begins by selecting an arbitrary non-White "clock" chain C_0 and subtracting $A(C_0.start)$ from all angles in all chains, thus rotating the angular frame of reference so that the starting angle of C_0 is 0. For each chain, the starting angle is then adjusted to be in the interval $(0, 2\pi]$ by adding some integer multiple of 2π, and the same adjustment is made to all other angles in the chain, thereby preserving angular differences within each chain. The notation *c.next* will be used for the chain following c in counterclockwise order of their starting angles (if multiple chains happen to have the same starting angle, the ordering among them can be arbitrary). The chain C designated as the clock chain will change as the algorithm proceeds, with the starting angle of C being called "*cursor*". *Cursor* decreases monotonically from 2π to 0 as the algorithm proceeds. The initial dominant chain D_0 cannot be White because C_0 itself covers 2π, but D_0 could be C_0. Like the clock chain, the dominant chain D will vary its identity as the algorithm proceeds.

The Cursor algorithm also involves an angle *whisker* that starts at 2π and follows *cursor*. Basically, *cursor* jumps counterclockwise from one clock chain start angle to the next, creating an angular gap between *whisker* and *cursor* on each occasion, and then *whisker* reduces the gap to 0 by catching up to *cursor*, marking angles as it moves. When *whisker* catches up, *cursor* moves again until *cursor* has decreased to 0 and *whisker* has caught up to it, at which point the algorithm terminates.

Figure 3 is a flowchart of the Cursor algorithm. The flowchart uses brief titles, so some notes are in order about exactly what is meant:

1. The algorithm starts at the topmost box, where *whisker* and *cursor* are both set to 2π.
2. The $\leftarrow$ symbol means that the left-hand side is updated by the right-hand side.
3. All tests are binary, with "Y" or "N" marking the exit where the answer is true or false.
4. *Cursor* jumps in the box labeled "$C \leftarrow$ *C.next*", where it is set to the start of the next clock chain in counterclockwise order.

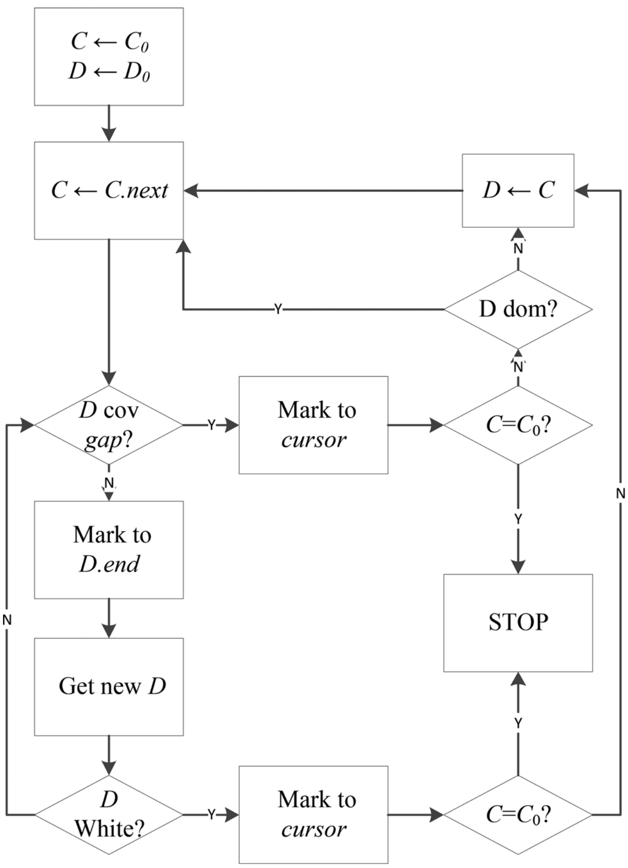

Figure 3. Flow diagram for the heart of the Cursor algorithm.

5. All three “Mark” boxes move *whisker* counterclockwise to the stated limit, marking all angles with D and reducing the gap.
6. “Mark to cursor” moves *whisker* to *cursor* and reduces the gap to a single point, where it will remain until the algorithm stops or the clock chain is again updated.
7. The test labeled “D cov *gap*?” asks whether chain D covers the gap.
8. “Get new D” replaces dominant chain D with its successor.
9. The test labeled “D dom?” asks whether chain D dominates chain C at *whisker* (or at *cursor*, since both are equal at this point).

Figure 4 shows one stage in the application of the flowchart where *cursor* has advanced after D has been reset in the box labeled “$D \leftarrow C$”. *Whisker* will move counterclockwise to *cursor* in two steps to close the pictured gap. The next marking will be in the “Mark to *D.end*” box, after which the dominant chain will be White.

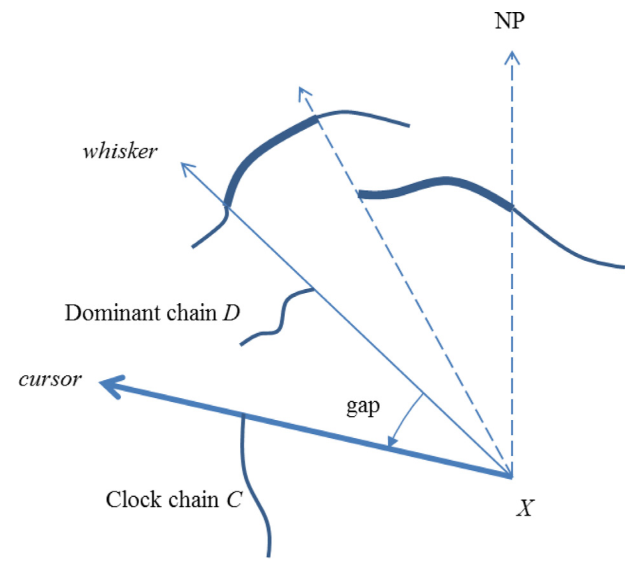

Figure 4. *Cursor* has just advanced from the position now occupied by *whisker*. Chain D will be marked to its end, thus partially filling the gap. Next the White chain will be found dominant up to *cursor*, after which the gap will be closed and *cursor* will move again. Previously marked parts of chains are shown heavy. The initial chain is not shown, but starts far out on the ray from X to NP. The chain pictured crossing that ray is the initial dominant chain D_0, which is partially marked. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

3.3.3. Convergence, Completeness and Correctness

All “Mark” boxes in Fig. 3 result in marking a positive part of the gap by definition of dominance, so no infinite loop can involve “D cov gap”. Clock chains never repeat until C_0 is encountered again, which terminates the algorithm. Thus convergence is assured.

All gaps are completely marked as they occur, and the gaps that occur when *cursor* moves cover a complete circle, so every angle in $(0, 2\pi]$ is marked. The Border function is therefore defined for all angles in that interval. For completeness take $B(0, X) = B(2\pi, X)$.

If chain c is dominant at angle θ and if $\phi > 0$, then c will also be dominant in $(\theta - \phi, \theta]$ unless it ends in that angular interval, as long as no other chain starts in that interval. This is because chains cannot cross each other. In the algorithm, *cursor* jumps counterclockwise from one chain start to the next in decreasing order of chain starts, so there are no chain starts between jumps. Between jumps, any chain that becomes dominant will therefore remain dominant until it either ends or covers *cursor*. This is exactly what happens during *whisker* movements, including movements that correspond to the White chain. Therefore all angles are marked correctly with the dominant chain.

3.3.4. Storing and Using the Border Function

Given X, the interval $[0, 2\pi]$ can be partitioned using the Cursor algorithm into $K+1$ intervals, in each of which a single chain is dominant. Call this partition P. The intervals of P can be described by pairs (D_k, ψ_k), with angle $\psi_k.next$ being the next angle in a cyclic counterclockwise ordering. Chain D_k is dominant in the angular interval $(\psi_k.next, \psi_k]; k = 0, \ldots, K$. The closed end of each interval is determined by the *whisker* position when chain D_k is discovered to be dominant in the "Get new *D*" box, and the open end by the *whisker* position when D_k either ends in "Mark to *D.end*" or loses to another chain in box " $D \leftarrow C$". Angle $\psi_0.next$ is the first angle defined (the angle at which the dominance of D_0 ends), and ψ_0 is the last and smallest angle defined. The first interval should be understood to include all angles $\phi \in [0, 2\pi]$ for which either $\phi > \psi_0.next$ or $\phi \leq \psi_0$; otherwise the open end of each interval is smaller than the closed end.

Calculating the distance to a chain will invariably be reduced to calculating the distance to one of its segments (the last computation in Appendix B). This is accomplished by further subdividing the intervals of P into subintervals where obstacle segments are dominant, rather than chains, thus obtaining the function $Seg(X, \phi)$ mentioned above in Section 2.3.

3.4. Qualification of Vertices

Once the Border function from either X or Y is determined, it can be easily determined whether X can see Y. If so, then the shortest route is direct. If not, then the shortest route will involve first going directly to some "transit point" i, a vertex that is visible from X. On account of the Interior theorem (Theorem 1 in Appendix A), there exists an optimal route where i is either a maximum or a minimum of some chain. The only vertices that qualify as transit points are thus the start and end of each chain. All other vertices on the chain, even if visible, can be eliminated from the candidate sets. This vertex elimination is in fact one of the main motivations for obstacle reduction in the first place. The same observation about optimal paths in the plane is made by [16].

3.5. The Antipode Issue

We now return to case 3 of Section 3.1 where the antipode of X is in the interior of the obstacle, a circumstance that does not have a counterpart in the plane. For example, imagine that the only obstacle is a convex polar cap surrounding the North Pole, and that X is the South Pole. The cap will spawn no chains because there are no maxima or minima—the angle function $A()$ simply decreases as one moves clockwise around the cap. Therefore none of the cap's vertices will be found visible in the cursor algorithm. In fact *all* of them are visible, but finding them invisible is harmless because they can all be eliminated by the corollary to the Interior theorem (Appendix A). In the example above, the polar cap cannot interfere with the shortest path between X and any other wet point. Therefore the Cursor algorithm as described above does not need to be modified to deal with the possibility that an obstacle spawns no chains, even though that possibility is a real one. If there are no maxima or minima, then all vertices on the obstacle are eliminated as transit points by the Interior theorem, and this happens automatically when there are no chains.

The antipode issue also potentially arises in the process of determining visibility to vertices, since the antipode of a wet point might be a vertex. However, all antipode vertices can be safely disqualified as transit points. Although it is possible that an optimal route from X might begin by going directly to its antipode j, in that case there will always be some vertex i that can see both X and j. The total length of the segments (X, i) and (i, j) will be π, so the initial segment can as well be (X, i).

3.6. Computational Experience and Potential for Improvement

All of the run times in this section are for an implementation in Microsoft Excel ™ using VBA macros on a Lenovo ThinkPad Carbon X1 laptop.

The static computation of the vertex routing matrix $D(i, j)$ must be completed and saved before the dynamic computation of routes between wet points can be undertaken. The static computations are made in two stages. In the first stage the Cursor algorithm (slightly modified to deal with "wet points" that are vertices) is used to determine inter-vertex visibility. This takes about one second when there are $n = 900$ vertices. The second stage uses a label-correcting dequeue network shortest path algorithm [1] to compute $D(i, j)$, and takes about 2 seconds. We have not experimented with larger values of n because 900 vertices suffice for our purpose, but would expect the time for the first stage to increase quadratically with n and soon overtake the time for the second stage. This overtaking is because the number of *qualified* vertices in the second stage (Section 3.4) will be bounded above if the increasing number of vertices is simply used to define a fixed number of obstacles more precisely. Readers who are disappointed with the large-scale performance of the Cursor algorithm for the first stage should consult the computational geometry literature, where algorithms that theoretically scale better with n will be discovered. The "visibility graph" of computational geometry is computationally equivalent to the output of the Cursor algorithm, for example, and has been much studied in the plane [8].

In the process of applying the Cursor algorithm to calculate the static routing matrix, a border for each vertex is also computed. Since a vertex border suffices to determine visibility to wet points, as well as to other vertices, one might suppose that saving and later recalling those borders would obviate the need to later compute wet point borders in the dynamic computations. The flaw in that argument is that vertex borders cannot be used to determine visibility between wet points, so a border for each wet point (except for one, since visibility is symmetric) will still need to be computed. Since the Cursor algorithm must still be applied anyway at almost all wet points to determine inter-wet-point visibility, there is little utility in storing vertex borders once the static calculations are complete.

The motivating RASP application provides a list of m possible destinations to WetRoute, requiring the shortest route between all possible pairs as output. The number of destinations on RASP's list is on the order of 100, but varies significantly between instances. Our main concern has been with WetRoute's performance on these dynamic computations. WetRoute begins the dynamic task by recalling the previously computed static vertex routing matrix $D(i, j)$, so the time required to *compute* that matrix is not included in the times quoted in the rest of this section.

The total time for a dynamic RASP call to WetRoute can be roughly partitioned into (a) the constant time required to load the static vertex routing matrix, (b) the time required to run the Cursor algorithm m times and make the ensuing visibility determinations, and (c) the quadratic time required to accomplish Eq. (1) for every pair of wet points. Time c is quadratic in m because the number of combinations of m things taken two at a time is quadratic in m. Time a is about 0.3 seconds when there are 900 vertices. This dominates the other two times when m is small. When 200 destinations are chosen randomly on the globe, the three times are approximately $(a, b, c) = (0.3, 0.3, 0.5)$ seconds, depending on exactly which destinations are chosen, and when $m = 1000$ the three times are $(0.3, 3.0, 12.1)$, again for randomly chosen destinations. The quadratic time c becomes dominant as m increases, but is nonetheless minimized because the candidate sets S_X and S_Y have been made as small as possible in the process of running the Cursor algorithm. For lists of the size provided by RASP, the total time never exceeds a few seconds. The times encountered in practice are usually smaller than those quoted above because the typical destination list includes many that can see each other in what the Navy calls an "area of operations," whereas that is unlikely when destinations are chosen randomly.

As mentioned above, the time complexity of the dynamic problem is $O(m^2)$. For problems where m is very large, the coefficient of m^2 might be made smaller than it is when (1) is repetitively employed as in WetRoute. That coefficient depends on the algorithm in use as well as the number of vertices. There are efficient ($O(n \log n)$) methods for calculating a "shortest path map" (SPM) that determines the shortest route from X to any other wet point. Implementations of these methods have been planar, but might be adapted to the sphere. Given the SPM for X, the shortest route from X to any specific wet point can be retrieved in a time that is $O(\log n)$ [11]. One way of dealing with our two-point query problem would be to compute an SPM for every wet point (except for one), thus making the coefficient of m^2 be $O(\log n)$. There are other promising possibilities. See [2] for a description of one of them, or [8] for a survey.

4. EXPERIENCE IN ROUTING NAVY SHIPS

4.1. Canals

The Panama and Suez canals are problematic because the effective speed of a ship decreases dramatically upon entrance. Most ships are more concerned about time (and fuel consumption) than distance, and a route that goes through a canal in order to minimize distance may not minimize time. There are two tempting approaches. Using the Panama Canal as an illustration, approach A is to separate America into its north and south parts with a narrow wet gap between the two obstacles. Method B keeps America as one obstacle, but introduces Atlantic and Pacific canal entrances as intervisible vertices v_1 and v_2. In making the static calculations, the "distance" between v_1 and v_2 is initially set to the time required to navigate the canal (an input) multiplied by the open-ocean speed of the ship (another input), thus artificially increasing the geographic distance between the two canal entrances. For example, a ship whose open-ocean speed is 15 knots and expects to take 9 hours to transit the canal would set the "distance" between v_1 and v_2 to be 135 nautical miles (the actual distance is about a third of that). The Suez Canal would be handled similarly. Method B is more complicated than method A, but nonetheless is employed in the software supporting RASP.

4.2. Oceanic Routing Service

The US Navy has an official way to approve routing a ship from X to Y. This involves sending a naval message to one of two Fleet Weather Centers, which will reply after several hours with a route that considers currents, winds, fuel consumption and other important things that are ignored in WetRoute [13]. Such routes are surely better than those provided by WetRoute, but the associated time delay can be onerous, especially if the ship is testing multiple destinations. The desire for a fast routing (and rerouting) method, even at the cost of some accuracy, has led to an implementation of WetRoute in an application called the Oceanic Routing Service (ORS), a component of the Navy's Maritime Tactical Command and Control (MTC2) system [14]. That software is not publicly available, but the interested

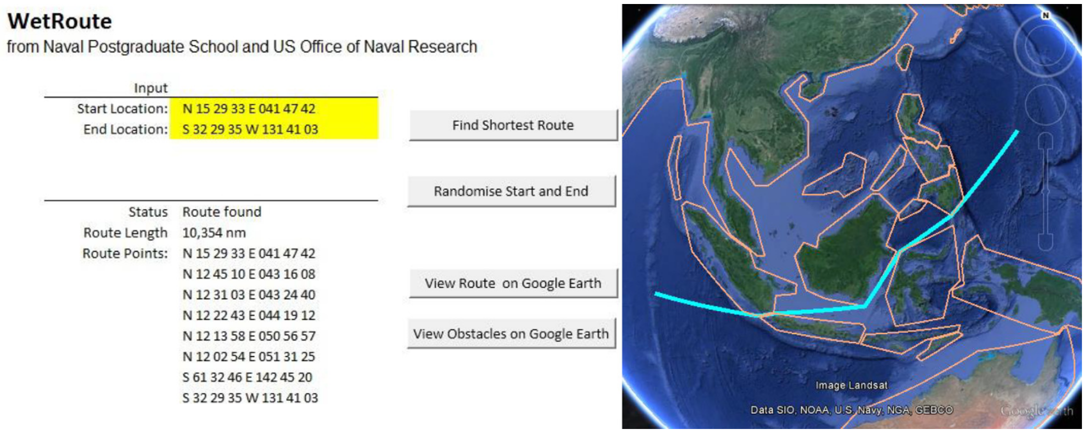

Figure 5. Showing the WetRoute digital interface (left) and map (right). [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

reader can download a similar spreadsheet *WetRoute.xlsm* from the Excel Spreadsheets section of the Downloads link at http://faculty.nps.edu/awashburn. Figure 5 shows the user interface from *WetRoute.xlsm*, together with a map of an optimal route created by exporting it to Google Earth™. Both applications use method A for dealing with canals.

4.3. Dry Points

Any attempt to approximate actual obstacles with simple polygons will include some wet points inside the polygons that are actually dry, as well as exclude some dry points that are actually wet. The latter mistake risks having ships run aground while following "optimal" routes, and the former mistake risks sarcastic comments from ships that are certain that they are afloat. As an inspection of Fig. 1 will reveal, the WetRoute obstacle database is biased to avoid running aground. Some provision must therefore be made to deal with destinations that are apparently dry. The provision in ORS or *WetRoute.xlsm* is null—the user will simply be informed that the point is dry and invited to try again. We have experimented with two other methods for use in the software supporting RASP. Method 1 simply moves a "dry" ship location to the nearest wet point. This is simple to implement, but has the drawback that the truly wet ship will see an initial arbitrary leg to the supposedly "optimal" route. Method 2 is to temporarily modify the offending obstacle so that the dry destination becomes wet, "temporarily" because other destinations will still see the original obstacle. Imagine dragging an obstacle vertex over to the dry destination. Method 2 is harder to implement, but is more in accord with the idea that reported ship positions are always truly wet. We use method 2 in the software supporting RASP, but method 1 also works. What doesn't work is to inform the captain of a US Navy ship that he is aground.

4.4. Submarines, etc.

Submarines also face shortest route calculations, albeit with an obstacle database that includes larger and more numerous obstacles because of the requirement to stay submerged. Work on developing an analog of ORS for submarines is ongoing, but the only essential difference is that the obstacle database has to change. One can also imagine applications for aircraft who are forbidden to fly over certain regions.

APPENDIX A: THEOREMS

THEOREM 1 (Interior Theorem): Assume that X and Y are both points that are neither inside nor on the border of any obstacle (wet points). In an optimal path from wet point X to wet point Y, if any segment has an endpoint on the boundary of an obstacle, then the extension of that segment beyond the endpoint will not immediately enter the interior of the obstacle.

PROOF: Figure 6 shows an obstacle and part of an optimal path that includes sequential segments (S,O) and (O,T), with point O being on the boundary of the obstacle. Suppose that the forward extension of segment (S,O) enters the interior of the obstacle, as shown by the dashed extension from point O (a similar proof applies if the backward extension of (O,T) enters the interior of the obstacle). If O is at a corner of the obstacle, as illustrated, let CW and CCW be the obstacle segments clockwise and counterclockwise from point O. Otherwise let CW and CCW be the clockwise and counterclockwise parts of the obstacle segment that includes point O. In either case CW and CCW must lie on opposite sides of the unique great circle that includes (S,O), and (O,T) must lie entirely on one side of the other, except for endpoint O. Suppose that (O,T) lies on the CW side, as illustrated (a similar proof applies if (O,T) lies on the CCW side). The angle SOT (shown as A in Fig. 6) must be smaller than π because segment (O,T), which cannot contact the interior of the obstacle, must lie angularly between CW and (S,O). Since $SOT < \pi$, a spherical triangle with sides a, b, and c can be formed by introducing a small shortcut from (S,O) to (O,T) (the dashed arrow shown in Fig. 6). Because obstacles do not touch each other, this shortcut will not contact any other obstacle if it is small enough. Since

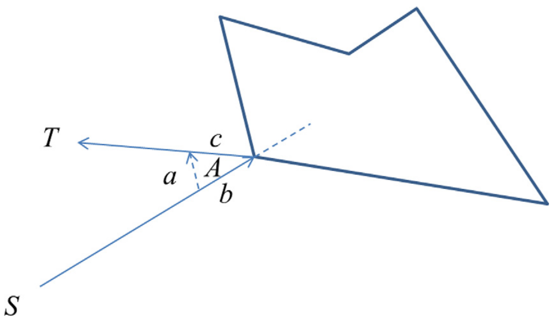

Figure 6. A path from S to T contacting an obstacle bounded by five segments. Point O (not illustrated to avoid clutter) is at the apex of angle A.

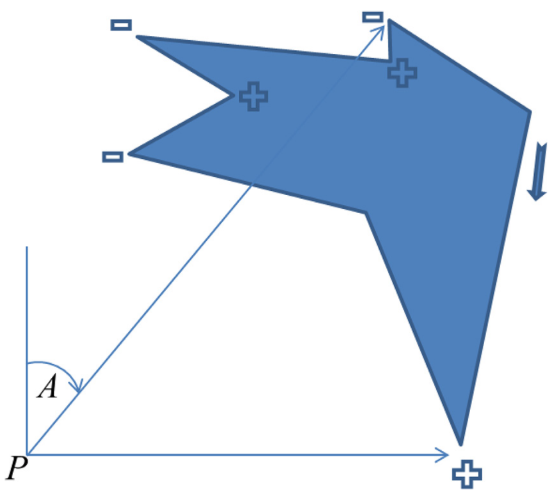

Figure 7. Illustrating a polygon with the interior shaded, three local minima (−), three local maxima(+), and the clockwise direction indicated. The two cursor positions (straight arrows from point P) bound a pair of CI segments that go from a minimum to the next clockwise maximum.

$a < b + c$ in a spherical triangle, a feasible path that follows the shortcut will be shorter than the path that includes O, contradicting the assumption that the latter path is optimal. □

COROLLARY: Suppose that an obstacle produces no chains. Then, regardless of Y, the optimal path from X to Y will not begin by contacting the obstacle.

PROOF: An obstacle can produce no chains only if the antipode of X is in the interior of the obstacle, and if obstacle reduction produces no maxima nor minima. Any segment from X to such an obstacle, if extended, will go through the antipode (in fact all great circles through X will go through the antipode). Since the antipode is in the interior of the obstacle and there are no maxima or minima, the extension of the segment from X to the obstacle must immediately enter the interior of the obstacle. According to the theorem, this cannot happen if the path from X to Y is optimal. QED

We first prove the next theorem on the two-dimensional plane, rather than on a sphere. Consider a clockwise planar polygon with boundary S, and an exterior point P. A ray from P to any point on S (the "cursor") has an angle A that is measured clockwise from some fixed direction (see Fig. 7). A segment

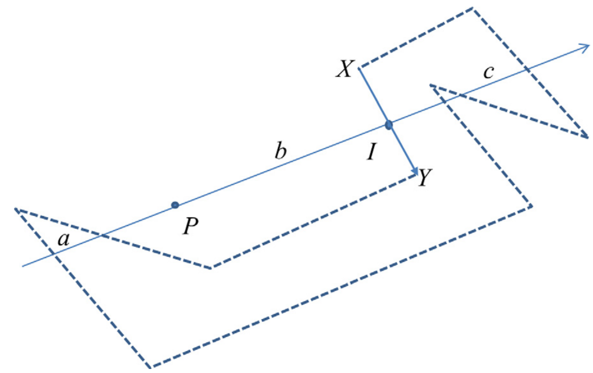

Figure 8. Illustrating a CI segment from X to Y. Try to complete the polygon in a clockwise manner without enclosing point P or crossing part b of the line. The dashed line shows a counterclockwise way of doing it where $A = 2$, $B = 1$, and $C = 3$. According to Theorem 2, there is no clockwise way.

(X, Y) of S is by definition "clockwise increasing" (CI) if the angle of a ray from P to X does not exceed the angle of a ray from P to Y. □

THEOREM 2: All CI segments (X, Y) of S are "invisible" in the sense that the cursor to any point I in the interior of such a segment will include some other point on S that is closer than I to the exterior point P.

PROOF: It is possible that the cursor to I will include an endpoint of the segment containing I. In that case the endpoint is closer to P than is I, which concludes the proof. The only other possibility is that the angle strictly increases as the cursor moves from X to Y, which we assume in the following. Figure 8 shows a CI segment from X to Y, together with a ray from P to an arbitrary point I in the interior of the segment. The line L containing the ray is partitioned into three parts separated by points P and I (the two heavy dots in Fig. 8). Part a extends from P to infinity without including point I, part c extends from I to infinity without including point P. Part b is the rest of the line, a segment whose endpoints are P and I. Consider an arbitrary clockwise completion of the (X, Y) segment into a polygon for which P is exterior. Let the boundary of the polygon be S, and let A, B, and C be the number of times that S crosses parts a, b, and c, respectively. Since I is in b, $B \geq 1$. If any segment of S lies entirely in L, then it must also lie entirely in either a, b, or c, and the number of crossings of that part is counted as 0 if the preceding and succeeding segments are both on the same side of L, or otherwise 1. None of these crossings can be at P because P is exterior, and exactly one will be at I because S is non self-intersecting. All interior points of b are closer to P than is I. We will show that B cannot be 1; that is, we will show that I cannot be the closest point of S to P. It will suffice to show that B must be an even number.

The total number of line crossings $A + B + C$ must be an even number because L is a complete line and S is a simple polygon. Furthermore, since a is an unbounded ray from exterior point P, A must be an even number because any ray from an exterior point will intersect a polygon an even number of times [3]. We will show that C must also be even. The direction from X to Y is clockwise if and only if points immediately to the left of I are exterior (unlike the counterclockwise S of the illustration, where points immediately left of I as one moves from X to Y are interior instead of exterior). Since the direction is clockwise by assumption, we can introduce an exterior test point I' within c that is so close to I that there are no line crossings between I and I'. Let C' be the number of times S crosses the ray from I' away from P to infinity. Then C' must be an even number because I' is exterior, and

Table 1. Computation times of various operations in nanoseconds

Addition or subtraction	2.7
Multiplication	4.1
Division	5.1
Square root	6.2
Sine or cosine	13.2
Arcsine or arccosine	14.0
Atn2	19.2

$C' = C$. Since A and C must both be even, B also has to be even, and the theorem follows. □

COROLLARY: As one moves clockwise around *S*, there will be an alternating sequence of maxima and minima of the cursor angle. All segments of the curve from any minimum to the next clockwise maximum are invisible to *P*.

Spherical Generalization

Theorem 2 remains true if *P* and the polygon are on a sphere, rather than the plane, as long as the boundary *S* does not include P', the antipode of *P* (the angle *A* is not well defined at P'). A sketch of the proof follows. The "line" *L* becomes the unique great circle that includes both *P* and *I*. Part *a* is the half of *L* between *P* and P' that does not include *I*, part *c* is the part of *L* between *I* and P' that does not include *P*, and part *b* is the rest of *L*. Great circle *L* must be crossed an even number of times by *S*, as in the plane. The relevant topological fact here is that the sphere, absent the antipode of *P*, is homeomorphic to the plane.

APPENDIX B: SOME COMPUTATIONAL OBSERVATIONS

All of these observations are based on the idea that every spherical point of interest has properties that include the sines and cosines of latitude and longitude, the time for computing of which will be ignored. Otherwise, the following table will be used to calculate the total time in nanoseconds on a Lenovo W530 laptop using one 64-bit processor. These are merely examples that influence how we compute the numerical results we seek. Graphical processing units and special computer enhancements will yield different results.

The two-argument arctangent function *Atn2*() is required for some of the computations described below. This function is sometimes defined so that $Atn2(1, 0) = 0$, and sometimes so that $Atn2(0, 1) = 0$. We use the former definition, and always $-\pi \leq Atn2(x, y) < \pi$.

Distance and Direction

Our formula for the cosine of the distance *cd* between *X* and *Y* is recorded in Section 2.2. According to that formula and the times in Table 1, computation of *cd* requires 39.9 nanoseconds. The final use of the arc-cosine function to find $d(X, Y)$ would require an additional 14 nanoseconds. Using double-precision arithmetic, we have found the given expression for *cd* to be accurate to within 10^{-9}, which corresponds to distances of less than a centimeter on the physical, spherical Earth. We therefore make no use of other formulas that are known to be more accurate for small distances [17].

The direction $C(X, Y)$ from *X* to *Y* is defined as in Section 2.2—the initial course in radians clockwise from the North Pole (NP). To compute $C(X, Y)$, define

$$\begin{aligned} den &= (X.clat)(Y.slat) - (X.slat)(Y.clat)((X.clon)(Y.clon) \\ &\quad + (X.slon)(Y.slon)) \\ num &= (Y.clat)((Y.slon)(X.clon) - (Y.clon)(X.slon)), \text{ and then} \\ C(X, Y) &= Atn2(den, num)\,. \end{aligned}$$

The derivation involves the observation that the points *X, Y*, and the North Pole form a spherical triangle. Variables *den*(*num*) are proportional to the *cosine*(*sine*) of $C(X, Y)$, the proportionality constant in each case being the sine of the distance between *X* and *Y*. After counting operations and using Table 1, the time required to calculate $C(X, Y)$ is 60.1 nanoseconds $(32.8 + 8.1 + 19.2 = 60.1)$. If only the sine and cosine of $C(X, Y)$ are needed, as is always the case in WetRoute, then the call to *Atn2*() can be replaced by a square root and two divisions.

If *X* is the South Pole, the angle returned will be $Y.lon - X.lon$. One might argue that the correct answer in this case is 0, since all courses from the South Pole go north, with a similar observation holding at the North Pole. [18], for example, makes exceptions for the poles. We do not make those exceptions. Our primary justification is that using our formula without the polar exceptions gives the correct answer in problems that involve turning at a pole, as in the next problem considered below (the irrelevant longitude of the pole cancels in that calculation). A secondary justification is that saying "go north" to someone at the South Pole is not sufficient guidance for motion, since it leaves open the question of which meridian to follow.

Determining Whether Two Great-Circle Segments Intersect

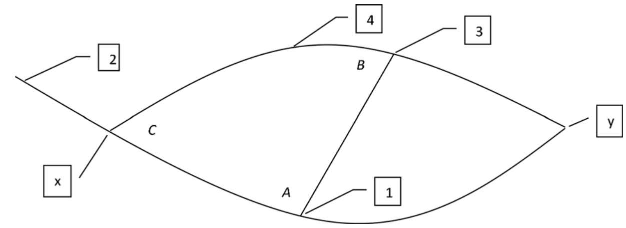

Figure 9. Showing the lune formed by two intersecting great circles, one determined by points 1 and 2, the other by points 3 and 4. The segment formed by points 1 and 3 completes a spherical triangle with interior angles A, B, and C. As pictured, segments (1, 2) and (3, 4) do not intersect.

The two segments connect point pairs (1, 2) and (3, 4). "Intersect" means that there is a unique point that is interior and common to both segments; that is, the two segments cross in the manner of an "X". Two segments do not thus intersect if any pair of the four endpoints are identical, so we will assume all four points are distinct in the sequel. Let d_{ij} be the length of the segment connecting point *i* to point *j*. If either $d_{12} = \pi$ or $d_{34} = \pi$, the corresponding segment is not unique because it connects antipodes, and can always be chosen so that there is no intersection. We therefore assume $0 < d_{12} < \pi$ and $0 < d_{34} < \pi$. First compute $cd_{12} \equiv \cos(d_{12})$ and $cd_{34} \equiv \cos(d_{34})$. Given the above assumptions, each pair of points uniquely determines a great circle. If the great circles determined by (1,2) and (3,4) are identical, then the segments do not have a unique interior intersection. Otherwise the two great circles will intersect at two points *x* and *y* that are antipodes of each other and the apexes of a spherical lune, as shown in Fig. 9.

Let θ_{ijk} be the angle from segment (*j,i*) to segment (*j,k*), measured clockwise. As long as $\sin(\theta_{134}) > 0$, angle *B* is θ_{134} as pictured in Fig. 9. Similarly angle *A* is θ_{213} as long as $\sin(\theta_{213}) > 0$. No *Atn2*() calls are required to find the sines and cosines of *A* and *B*. For the moment, assume that both of the sines are positive, as in the figure. In that case we can use the law of cosines for the spherical triangle x-1-3 to obtain

$$\cos(C) = \sin(A)\sin(B)cd_{13} - \cos(A)\cos(B).$$

The sine of C is necessarily nonnegative, and can be obtained with another square root. We can then use the law of cosines again to obtain

$$cd_{1x} = \frac{\cos(B) + \cos(A)\cos(C)}{\sin(A)\sin(C)} \text{ and } cd_{3x} = \frac{\cos(A) + \cos(B)\cos(C)}{\sin(B)\sin(C)}.$$

An intersection occurs if and only if $d_{1x} < d_{12}$ and $d_{3x} < d_{34}$. Since the inverse cosine function is monotonic, this criterion is equivalent to $cd_{1x} > cd_{12}$ and $cd_{3x} > cd_{34}$. This completes the computation in the case where $\sin(\theta_{134})$ and $\sin(\theta_{213})$ are both positive.

Should it happen that $\sin(\theta_{134})$ and $\sin(\theta_{213})$ differ in sign, the meaning is that points 2 and 4 lie on opposite sides of the great circle determined by points 1 and 3. In that case intersection is impossible. An "X" intersection is also impossible if one of the sines is 0. If both are negative, then points 2 and 4 lie towards *y*, rather than towards *x*. The analysis in that case is similar. The important point here is that expensive calls to trigonometric functions are not required as long as the sines and cosines of the input points are already available.

Intersection of a Ray with a Segment

At what distance d does a ray from point *A* intersect the great circle determined by segment (*B,C*)? The three points form a spherical triangle. Let *a*, *b*, and *c* be the cosines of the opposite side lengths, all assumed smaller than 1 in absolute value, let ϕ be the clockwise angular distance in radians from (*A,C*) to the ray, with $0 < |\phi| < \pi$, and let s_1 and c_1 be the sine and cosine of ϕ. The following will calculate $cd \equiv \cos(d)$ efficiently:

$$\begin{aligned} c_2 &\equiv (c - ab)/\sqrt{(1-a^2)(1-b^2)} \\ s_2 &\equiv \sqrt{(1-c_2{}^2)} \\ c_3 &\equiv bs_2s_1 - c_2c_1 \\ s_3 &\equiv \sqrt{(1-c_3{}^2)} \\ cd &\equiv (c_2 + c_3c_1)/(s_3s_1)\,. \end{aligned}$$

Parameters c_2 and c_3 are less than 1 when squared because they are the sine and cosine of an angle in a spherical triangle. Given that the initial sines and cosines are already available, the total time required is not quite 100 nanoseconds.

ACKNOWLEDGMENTS

This paper has benefitted considerably from suggestions made by referees. Professor Wilson Price toured the world for us with Google Earth, dropping pins on obstacles to navigation. Research Associate Professor Anton Rowe tends our various implementations and graphical interfaces. Professor Matt Carlyle helped us with shortest path algorithms. This work has been supported by the Office of Naval Research.

REFERENCES

[1] R. Ahuja, T. Magnanti, and J. Orlin, Network flows, Prentice Hall, Saddle River, NJ, 1997. chapter 5.

[2] Y. Chiang and J. Mitchell, "Two-point {Euclidean} shortest path queries in the plane," in: Proc. Tenth Annual ACM-SIAM Symposium on Discrete algorithms, Baltimore MD, January 16–19, 1999, pp. 215–224.

[3] O. Cismasu, 1997, The Jordan curve theorem for polygons, http://www-cgrl.cs.mcgill.ca/~godfried/teaching/cg-projects/97/Octavian/compgeom.html, accessed July, 2016.

[4] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, Computational geometry: Algorithms and applications, 3rd Ed., Springer-Verlag, Berlin, 2008. chapter 15.

[5] E.W. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik 1 (1959), 269–271.

[6] L. Euler, Solutio problematis ad geometriam situs pertinentis, Novi Commentarii Academiae Scientarium Imperialis Petropolitanque 7 (1758), 9–28 (English translation available from N. Biggs, E. Lloyd, and R. Wilson. Graph theory: 1736–1936, Clarendon Press, Oxford, UK, 1976, pp. 3–8.).

[7] K. Fagerholt, S. Heimdal, and A. Lotku, Shortest path in the presence of obstacles: An application to ocean shipping, J Oper Res Soc 51 (2000), 683–688.

[8] J. Goodman and J. O'Rourke (Editors), CRC handbook of discrete and computational geometry, 2nd Ed., CRC Press LLC, Boca Ration, FL, 2004, pp. 607–641.

[9] C. Grima and A. Marquez, Computational geometry on surfaces, Springer, Berlin, 2001. chapter 3.

[10] MathWorld, 2015, Spherical polygon, http://mathworld.wolfram.com/SphericalPolygon.html, accessed July, 2016.

[11] J. Mitchell, A new algorithm for shortest paths among obstacles in the plane, Ann Math Artif Intell 3 (1991), 83–106.

[12] D. Sunday, 2013. Inclusion of a point in a polygon, http://geomalgorithms.com/a03-_inclusion.html, accessed July, 2016.

[13] United States Navy, 2007, Navy's optimum track ship routing going to automation, http://www.navy.mil/submit/display.asp?story_id=27544, accessed July 2016.

[14] United States Navy, 2015a, Program Executive Office Command, Control, Communications, Computers and Intelligence (PEO C4I) PMW 150 Command and Control Systems Program Office, http://www.public.navy.mil/spawar/PEOC4I/Documents/Tear%20Sheets/PMW%20150_Tear%20Sheet_JAN2015-approved.pdf, accessed July, 2016.

[15] United States Navy, 2015b, Military Sealift Command Replenishment at Sea Planner (RASP), http://mscsealift.dodlive.mil/2015/08/04/replenishment-at-sea-planner-program/, 4 August, accessed July, 2016.

[16] J. Viegas and P. Hansen, Finding shortest paths in the plane in the presence of barriers to travel for any l_p-norm, Eur J Oper Res 20 (1985), 373–381.

[17] T. Vincenty, Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations, Surv Rev XXIII (1975), 88–93.

[18] E. Williams, 2015. Aviation formulary V1.46, http://williams.best.vwh.net/avform.htm, accessed July, 2016.