3

|  |
| --- |
| Simulator Module |
| SAROPS Version 2.2 |
|  |
| This document describes the basic approach and algorithms of the Simulator Module in SAROPS 2.2. |
| This document contains data that is delivered to the Government with Unlimited Rights in accordance with contract HSCG44-16-F-001007 and FAR 52.227-17. |
| Thomas M Kratzke, Strategic Data Systems, 03 October, 2018 |

Table of Contents

[I. Introduction 0](#_Toc526327521)

[A. Non-Sim and Non-Planner Modules of SAROPS Sim 0](#_Toc526327522)

[B. Overview of Sim 1](#_Toc526327523)

[C. Sim Input Files 1](#_Toc526327524)

[1. Sim.xml 1](#_Toc526327525)

[2. Sim.properties 2](#_Toc526327526)

[3. Environmental Files 2](#_Toc526327527)

[D. Main Sim Output File 2](#_Toc526327528)

[E. Overview of the Document 3](#_Toc526327529)

[II. Scenarios 3](#_Toc526327530)

[A. and Scenarios 3](#_Toc526327531)

[B. LOB Scenario 4](#_Toc526327532)

[C. Flare Scenario 7](#_Toc526327533)

[D. Voyage 7](#_Toc526327534)

[E. LKP Plus Dead-Reckon 10](#_Toc526327535)

[F. Sailboat Voyage 11](#_Toc526327536)

[1. Basic Pre-Distress Movement of a S/V; Using Polars 11](#_Toc526327537)

[2. Sail Segments; Motoring and Hove-To 14](#_Toc526327538)

[3. Stopping a Sail Segment 15](#_Toc526327539)

[4. Avoiding Land 16](#_Toc526327540)

[5. Particle State, SVT, and SVTs for S/V particles 17](#_Toc526327541)

[III. Hazards and Distress Incident 18](#_Toc526327542)

[IV. Reverse Drift Scenario 20](#_Toc526327543)

[V. Movement of Distress Particles; Using Environmental Data 20](#_Toc526327544)

[A. Non-Riverine Interpolation within a Time Layer 21](#_Toc526327545)

[B. Riverine Currents Interpolation within a Time Layer 22](#_Toc526327546)

[C. Perturb the Data 23](#_Toc526327547)

[D. Computing the Vector 23](#_Toc526327548)

[E. Creating a Leeway Calculator 24](#_Toc526327549)

[F. Using a Leeway Calculator 25](#_Toc526327550)

[VI. Land Processing 25](#_Toc526327551)

[A. Shoreline Data Characteristics 26](#_Toc526327552)

[B. Land Storage and Truncation of Polygons 27](#_Toc526327553)

[C. Storing the Edges; GeoMtx 29](#_Toc526327554)

[1. The Dateline is a Nuisance 30](#_Toc526327555)

[D. Slippery Shore, Stop-and-Block 31](#_Toc526327556)

[E. SimLand and BuildSimLand 32](#_Toc526327557)

[VII. Bathymetry and Anchoring 32](#_Toc526327558)

[A. ETOPO 32](#_Toc526327559)

[B. Great Lakes 33](#_Toc526327560)

[VIII. Completed Searches 34](#_Toc526327561)

[A. Lateral Range Curves and 35](#_Toc526327562)

[B. ESS 35](#_Toc526327563)

[C. ESS+NVG 37](#_Toc526327564)

[D. 5-Minute Rule 37](#_Toc526327565)

[E. Common Roots with Planner 38](#_Toc526327566)

[IX. Particle File 38](#_Toc526327567)

[A. Dimensions and Variables of Particle File 38](#_Toc526327568)

[B. Strings of Particle Files 41](#_Toc526327569)

[REFERENCES 42](#_Toc526327570)

List of Tables

[Table 1 Main Components of Sim.xml 2](#_Toc526327571)

[Table 2: Data of Bearing Call 5](#_Toc526327572)

[Table 3: Requirements for an LOB Fix 5](#_Toc526327573)

[Table 4: States for Pre-Distress Sailboat Voyage Particles 17](#_Toc526327574)

[Table 5: Using the Environment to Move a Particle 20](#_Toc526327575)

[Table 6: Leeway Data 24](#_Toc526327576)

[Table 7: Leeway Calculator 24](#_Toc526327577)

[Table 8: Characteristics of Land Edges 26](#_Toc526327578)

[Table 9: Polygons from Sections of a Polygon 28](#_Toc526327579)

[Table 10: 4 Levels and their Colors 29](#_Toc526327580)

[Table 11: Steps for Creating a NetCDF File 38](#_Toc526327581)

[Table 12: Dimensions in a Particle File 39](#_Toc526327582)

[Table 13: Basic Variables or Tables of a Particle File 40](#_Toc526327583)

[Table 14: Variables Averaging out Particle 40](#_Toc526327584)

[Table 15: OcTables of Particle File 41](#_Toc526327585)

List of Figures

[Figure 1: Non-convex Polygon 4](#_Toc526327586)

[Figure 2: Area of an LOB 6](#_Toc526327587)

[Figure 3: Snowcone diagram: Center (green dot) and Vertex (orange dot) 7](#_Toc526327588)

[Figure 4: "Correlation” between Dwell Zones 8](#_Toc526327589)

[Figure 5: Connecting Centers of Mass, and Perfect Correlation 9](#_Toc526327590)

[Figure 6: Perfect Anti-Correlation and No Correlation 10](#_Toc526327591)

[Figure 7: Correlation = 0.75, used in Sim 10](#_Toc526327592)

[Figure 8: Intervals of a Hazard for an Example Particle 19](#_Toc526327593)

[Figure 9: High School Equation of a Line 25](#_Toc526327594)

[Figure 10: Truncation Produces Multiple Polygons 28](#_Toc526327595)

[Figure 11: Truncated Island inside of Truncated Lake inside of Truncated Continent 29](#_Toc526327596)

[Figure 12: Lake Basins for Superior, Michigan-Huron, Erie, Ontario, and Champlain 34](#_Toc526327597)

[Figure 13: Non-zero Detection Region for ESS 36](#_Toc526327598)

[Figure 14: Practical Detection Region 37](#_Toc526327599)

Simulator Module

SAROPS Version 2.2

# Introduction

There are four major components of the SAROPS Sim module. Two of them are highly visible; the Simulator which produces a dynamic probability distribution, and the Planner that uses this probability distribution to assign boxes to SRUs. In this document, we discuss Simulator, and leave a discussion of Planner to other documents. We will refer to the Simulator module of SAROPS 2.2 as “Simulator 2.2” or simply “Simulator,” or even simply “Sim.” Similarly, we will refer to the Simulator module of SAROPS 2.0.3 as Simulator 2.0.3, and the Simulator module of SAROPS in general as simply Simulator or Sim.

The two other components of the SAROPS Sim module are the SimWebServer and the BuildSimLand modules. Here we give very brief overviews of these two modules.

## Non-Sim and Non-Planner Modules of SAROPS Sim

SimWebServer is a class in the “control” package of the code. This class has a “main” method which is the entry point for running the program SimWebServer. This main simply creates a single SimWebServer object, which creates a org.eclipse.jetty.server.Server and a SimCaseManager. This latter object is used to collect and run Sim and Planner cases in parallel. A SimCaseManager does not deal with handling http requests or port numbers or any of the other intricacies of being a web server; it simply queues up the Sim and Planner requests, and assigns them to its own set of “engines” as these engines become available. The commands that a SimWebServer’s Server handles are “Run a Sim or Planner Case with the following xml file,” and “Give the status of a particular case or all of the cases.”

BuildSimLand is another class/main combination that builds the “land files” from a collection of input files. The resulting “land files” are called SimLand. In 2.2, this run takes about 90 minutes, which is down from 36 hours. If additional corrections are needed (e.g. missing islands or errant coast lines), we must produce files that specify these corrections, and rerun BuildSimLand to produce a new version of SimLand.

In 2.2, we are relying heavily on Google Earth (GE) to provide data for BuildSimLand, and hence we are using kml files. kml files are xml files with special tags. The author did not find a package for reading and writing kml files, and translating those special tags into java objects, so he wrote code, based on the existing xml facilities within the current code, to read and write only those tags.

## Overview of Sim

Sim is the module that produces the paths of the particles. When a Sim run is completed, it leaves behind a file of particles, called the . The particles can be grouped together in cells and these cells are displayed as a probability distribution. The Planner, a separate program, reads the and computes rectangles for Search and Rescue Units (SRUs) based on this file. Hence, there is no direct communication between Sim and Planner. Indeed, Sim isn’t even running when Planner runs, and Sim’s creation, the , is read for every Planner run. These Planner runs include the “GetInitial,” the “Optimize,” and the “Eval” runs; these are all Planner runs, and each of these starts by reading the same that Sim produced.

## Sim Input Files

### Sim.xml

The main input to a Sim run is an xml that we call the “Sim.xml file” or simply “Sim.xml.” The key data within Sim.xml are a set of “Scenarios” and a set of “Object Types.” Each scenario is of a “scenario type,” and provides distributions of particles before or at distress. For example, the simplest scenario type is an “LKP” (last known position) scenario, which means that we have a central Lat/Lng and time of the distress incident, and there is no pre-distress motion. This is modelled by a circular bivariate normal in position and a truncated normal in time. Other scenario types include “LKP+Dead-Reckon,” flare sighting, Line-of-bearing, and voyages. In Sim 2.1, there is pre-distress motion only in the Voyage and LKP+Dead-Reckon. In Sim 2.2, there is a new type of scenario, called the Sailboat Scenario.

The object types represent different the types of things that we are looking for. Each object type is either the originating craft or a distress type. The “originating craft” is relevant only for pre-distress motion, and distress types are relevant only after the distress. Examples of distress object types are a “person in the water,” a “raft,” a “capsized boat,” etc.

Scenarios and Object Types are the main components of the data, but the largest set of data is the environmental data. Sim pushes around the distress particles by using wind and current data. The wind affects different object types in different ways, and each object type has “leeway data” that specifies this effect.

In addition, there may be completed (unsuccessful) searches and this has an impact on the probability distribution as well; particles that were well searched are less likely to represent the missing object and their probabilities or weights are reduced. Therefore, each object type needs a lateral range curve for Sim to update the particles’ weights from completed searches.

Summarizing, the main components of Sim.xml are:

1. Scenarios
   1. Type and Data
2. Object Types
   1. Type and Leeway Data
   2. Lateral Range Curve for each Completed Search
3. Specification of NetCdf files providing wind and current information
4. Area-of Interest specification for collecting land and bathymetric data

Table 1 Main Components of Sim.xml

### Sim.properties

The data in Sim.xml are very simple xml: tags and attributes. A resource-file Sim.properties within the main jar file provides many defaults for the data in Sim.xml. The “fully qualified” name of a datum is its path of tags followed by its attribute name. For example, there is a slope attribute in the DWL tag, which is a sub-tag of the LEEWAY tag, which in turn is a sub-tag of the SEARCH\_OBJECT\_TYPE tag. Therefore, the fully qualified name of this datum is SEARCH\_OBJECT\_TYPE.LEEWAY.DWL.slope. The names of the properties are the fully qualified datum name, appended by either .DEFAULT or .OVERRIDE. The former indicates that the property’s datum should be used only if the datum is missing from Sim.xml and the latter indicates that the datum should always be used. Put another way, if the property is “only” “.DEFAULT,” it will be ignored if it is in the xml, and if the property is “.OVERRIDE,” then the xml will be ignored.

In addition to the jar-file Sim.properties, there *can be* a Sim.properties file in the data subdirectory of the install directory. The properties in this file overwrite the ones in the jar-file’s Sim.properties. If a property within the Sim.properties ends with “.OVERRIDE,” then the corresponding property that ends with “.DEFAULT” (if any) is removed.

As of Sim 2.2, another source of Sim.properties is available. The xml itself can specify a 3rd place to find the properties. On the main tag (SIM for Simulator, PLAN for Planner), the attribute, if it exists “XmlSimPropertiesFilePath” specifies another file and the properties in that file will override the previously defined properties as in the preceding paragraph.

### Environmental Files

A Sim run uses wind and current information that are stored in NetCdf files. The rest of SAROPS produces these files and tells Sim where they are in the Sim.xml file. In addition, there are static files containing the shoreline and bathymetric data. These are large files, and the Sim.xml file tells Sim which parts of these files should be collected for the Sim run.

## Main Sim Output File

There are several output files that Sim produces, but the main one is the . In fact, the rest of the output files could be derived from the . The stores all of the information about the particles and their paths in a NetCDF file. We give a detailed description of the Particle File in §IX.

## Overview of the Document

In §II, we will discuss the different types of Scenarios. In §III, we will explain the data and method that a distress time is chosen. There are new features here. We give a brief description of reverse drift in §IV.

In §V, §VI, and §VII, we describe how the environment affects the particles, and we finish with a discussion of Completed Searches, comparing them to Planner’s reports, in §VIII.

# Scenarios

Scenarios are the basic way we describe what we know about the case. For example, we might know that the lost person intended to follow a certain route, or we might know only where his last known position was. Furthermore, we use multiple scenarios to indicate that there are several possibilities for what might have happened.

## and Scenarios

These are very similar. stands for “Last known position.” A Lat/Lng is given together with parameters specifying a bivariate normal distribution. Because a bivariate normal is a concept on a 2-dimensional plane, a 2-dimensional approximation to the earth is created and draws are made for the initial position of each particle, from a bivariate normal distribution.

In addition, a time uncertainty is part of this data, and the initial time of the particles are also randomly drawn. This will be a repeated theme for all the scenarios, and it will have consequences described in §III.

The scenario is very similar, except that the initial distribution is a uniform distribution over a polygon. This time, we choose a planar approximation out of convenience; as opposed to bivariate normal distributions, a spherical version of a uniform polygonal distribution *does* exist, but we use the planar version because it’s easier to work with.

We implemented the distribution of a uniform polygon so that it would be efficient for all polygons, not just convex simple ones. Consider the blue polygon of Figure 1.

Some Island

Figure 1: Non-convex Polygon

One common approach is to form a “bounding rectangle” around the polygon and make draws from it. Once such a point was drawn, one could test for containment in the blue polygon, and re-draw if the point were not in the blue polygon.

In the case illustrated in Figure 1, there would be many re-draws; most points within the bounding rectangle are not in the polygon itself. Moreover, the containment test can be quite expensive for complicated polygons.

Instead, a one-time setup of a data structure is computed and this allows 2 draws to always produce a single point within the polygon itself. The first uniform draw represents the and this is translated to the -coordinate. The data structure that allows this to happen efficiently is quite complicated and we will not go into that here. The second uniform draw represents the , where is already known, and this is translated to the -coordinate. Again, the data structure and associated algorithm for this is tricky. We will say only that both algorithms involve two binary lookups and hence it is fast.

We will use the ideas of the above paragraph in the Voyage scenario of II.D, but only after a change of coordinates. The Voyage scenario has pre-distress motion and is a sequence of bivariate normal and uniform polygonal distributions. Draws for the positions within these areas are not drawn randomly and the draws that we make there depend heavily on the s of the above paragraph.

## LOB Scenario

The LOB Scenario is similar to the LKP and Area scenarios in that there is no pre-distress motion; particles drift according to the winds and currents and their leeway data. However, the similarity ends there.

A collection of Lines-of-bearing calls (LOBs) are received and a single distribution must be made from them. This distribution plays the role of the bivariate normal or uniform polygon distribution of the previous section and, other than that, an LOB scenario is the same as these two.

If the LOBs are “consistent” (to be made precise shortly), we create a single elliptical distribution and draws are made from that. If they are not consistent, the ‘worst’ (again, to be made precise shortly) LOB is removed and we try again. If we get down to two LOBs and these are still not consistent, then the LOBs are all used, but individually.

The data of a single LOB is:

1. Center Point (Lat/Lng) of Origin of Call, and x-error of center point
2. Called Bearing
3. Min and max range
4. Standard Deviation for the bearing

Table 2: Data of Bearing Call

We follow [WANGSNESS], and we call the algorithm of this paper, the “Wangsness’ Algorithm.” This method produces an elliptical distribution for the position. We modified the technique so we could consider a circular uncertainty in the origin of each LOB.

The ellipse must pass the requirements listed in Table 3 below. In this table, the center of the ellipse is called the “fix” and the ellipse itself is the 95% containment ellipse.

1. The area of the ellipse must be at most 3000 sq nmi
2. The semi-major axis of the ellipse must be at most 500 nmi
3. The ratio of the two ellipse’s axes must be at most 64
4. When one projects the fix to each bearing call’s great circle, the fix must be in the direction of the called bearing, as opposed to 180 degrees off
5. If there are exactly two bearing calls, their bearing calls’ intersections must be at least 5 degrees
6. The fix must be between the min and max ranges for each bearing call

Table 3: Requirements for an LOB Fix

Although the constants of Table 3 have never been given in Sim.xml, they could be. In practice, the defaults specified in Sim.properties are used.

If any of these conditions fails, we consider the set of LOBs to be a “inconsistent” and an LOB is discarded. The one that is discarded is the one that is farthest from the fix, after dividing by its maximum distance.

If we reach a situation where there are only two bearings left, and they still fail a test in Table 3, then we use a uniform distribution of all the lines of bearing as follows. Each line of bearing has an area; it is the section of the annulus defined by the minimum and maximum ranges, with angular bounds given by ± the standard deviation of the bearing. This is illustrated in orange in Figure 2.

Max Range

Min Range

Figure 2: Area of an LOB

The particles are divided (nearly) equally among these lines of bearings’ areas, and the set of particles assigned to a line of bearing are distributed with a uniform distribution within the LOB’s area. Note that such a distribution has a closed form and is easy to generate.

It is possible to directly give an ellipse to an LOB scenario. In this case, all other inputs (LOBs and other ellipses) are ignored.

Wangsness’ Algorithm is quite elegant and straightforward. However, it does involve eigenvalues and linear regression, and we will not reproduce it here. Moreover, it does not allow for position uncertainty of the origin. We describe our modification for that here.

We will call the origin of the bearing call the “center,” and will define another point that we call the “vertex.” In Figure 3, the green dot is the center and the orange dot is the vertex. This vertex will be on the same great circle as the line of bearing, but behind the center. It is based purely on geometry and not probability, and depends on the circle with containment of 0.5 in a bivariate normal. We call this circle the uncertainty circle, or COU. The radius of the COU is precisely the data that is provided for the LOB.

After the COU is given, we use the standard deviation of the bearing call to project backward to the vertex, as illustrated in Figure 3 and the following discussion.

Call the bearing uncertainty . In Figure 3, we have drawn part of the great circle that is perpendicular to the line of bearing at the center. We move along the COU by , away from the LOB, to find the tangent points on the circle; these are the blue dots in Figure 3 We took the great circles that are tangent to the COU at these tangent points, and found their intersection, and that is the (orange) vertex. We call the diagram in Figure 3 the “snowcone” diagram, where the COU is the “snow” and the projection back to the vertex is the “cone.”

From the vertex, we compute the ½-range of angles between the vertex and the tangent points and use that as the standard deviation of the bearing call. Note that if the bearing uncertainty is 0, the vertex is the antipodal point of the center. If the position uncertainty is 0, we simply define the vertex to be the center, and use the given standard deviation.

LOB: 090

COU

Figure 3: Snowcone diagram: Center (green dot) and Vertex (orange dot)

## Flare Scenario

A flare scenario is like an LOB scenario. In fact, the code represents a flare scenario as a special type of an LOB scenario with one LOB. Both are represented by the class com.tmk.sarops.model.LobScenario. The main difference is that a flare scenario behaves much more like an area scenario, where the area is a polygonal approximation to the orange area of Figure 2.

In theory, there could be more than one flare sighting for a scenario. In practice, SAROPS forbids this. If it were allowed, the intersection of the orange areas would be used. Additional sightings would be allowed until the intersection becomes empty.

## Voyage

The voyage scenario is the first of two scenarios with pre-distress motion. The voyage scenario represents a situation in which the subject scheduled one or more of a sequence of stops or Dwell Zones. These Dwell Zones do not have to have a positive dwell-*time* and without positive dwell-times, they act merely as routes.

The Dwell Zone are either polygons or circles, like the LKP and area scenarios. For each particle, a sequence of positions is chosen, one from each Dwell Zone Dwell Zone. A loiter time is also picked for each Dwell Zone. For each particle and leg (transit between Dwell Zones), a speed is selected. The result of all this is a complete trajectory or path for each particle.

For each particle, the position within each Dwell Zone is not independent of the previous or next Dwell Zone. Rather, the draws are correlated. In the code (not even in Sim.properties), there is a constant , called the correlation value, and it is currently set to 0. The name of the constant is paying homage to the correlation coefficient, but we make no claims that this is the correlation coefficient. For that matter, we’re not entirely sure what the correlation coefficient between two 2-d random variables is anyway. No matter how we define 2-d correlation, we would get perfect correlation if we set to one, independence if we set it to 0, and perfect anti-correlation if we set it to -1.

Dwell Zone A

Dwell Zone B

Figure 4: "Correlation” between Dwell Zones

We illustrate our technique in Figure 4. We suppose we are going from Dwell Zone A to Dwell Zone B, the stars indicate the centers of mass of the two zones, the red axis is the great circle connecting the two stars, and the green axis is perpendicular to the red axis at Dwell Zone A’s center of mass. Suppose also we wish to “correlate” the brown triangle to some point in Dwell Zone B.

With respect to the green axis, let’s suppose that the brown triangle has a value of 0.8. If we did “perfect correlation,” we would pick a point in Dwell Zone B that had a value of 0.8 with respect to the purple axis. That axis was chosen as the great circle perpendicular at Dwell Zone B’s center of mass to the red great circle. To get the value within Dwell Zone B, we use something similar, but base it on the conditional of . Let’s say that the result is the purple square that is connected to the brown triangle by the dashed line. We call this point the “partner” of the brown triangle.

We do not use perfect correlation, but we do base our draws on the partner. Now note that, for any continuous random variable , is a -(0,0) random variable. We convert the value to a Gaussian one by using the inverse of a Gaussian, and make a (truly) correlated draw for a new Gaussian by using the formula to get a new Gaussian. We convert that to a by finding its value, and then using that uniform and our method of drawing a uniform polygonal draw, to get the draw in Dwell Zone B.

This correlation was requested long ago because it really doesn’t make sense for the two draws to be independent. What this is capturing is simply the notion that “if someone starts to the left, he generally stays more left than right.”

In the following figures, we show the same simulation case with 100 particles, but change the correlation value. The dark green segment is the section of the great circle that connects the two areas’ centers of mass. This is most visible in the first picture of Figure 5. The other pictures are for “perfect correlation,” perfect anti-correlation,” “no correlation,” and “correlation = 0.75,” which is what is use in Sim.

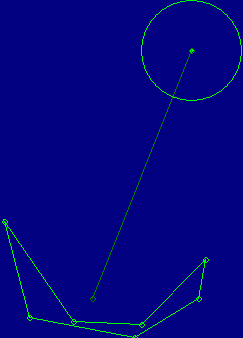
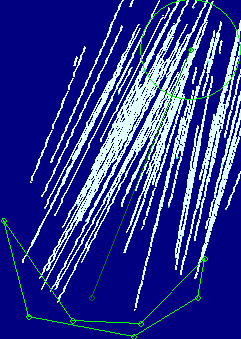
 

Figure 5: Connecting Centers of Mass, and Perfect Correlation

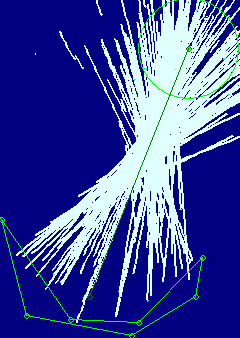
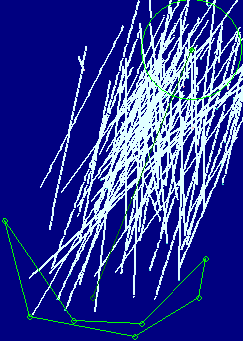
 

Figure 6: Perfect Anti-Correlation and No Correlation

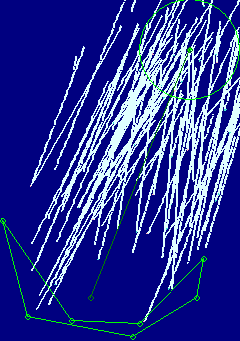


Figure 7: Correlation = 0.75, used in Sim

## LKP Plus Dead-Reckon

This one is conceptually simpler. For each particle, we draw a starting point and time as with LKP’s, and a direction. The particle moves in that direction until distress; the last random characteristic for the pre-distress is the direction.

## Sailboat Voyage

In Sim 2.2, we have introduced a special type of voyage, called a “sailboat voyage,” or S/V. The S/V behaves in the end very much like the regular voyage of §II.D, but the operator does not provide all of the legs. The operator does provide legs just as with a regular voyage, and these legs are specified by waypoints in the xml, just as with a regular voyage. Furthermore, for each particle, points are drawn from the waypoints’ spatial distributions and thus, are created for a S/V particle.

However, these do not provide the entire route of the S/V particle. For each , Sim must create that are feasible for a S/V to travel. For example, if a is straight up-wind, Sim will have to compute to get from the beginning of the to its end. We use the term instead of , and discuss how Sim creates these in this section.

Note that the completely disappear because each one is replaced by a sequence of . In contrast with §II.D, each particle can have a different number of legs. However, once these legs (i.e., the ) are computed, the simulation proceeds the same, with hazards, randomly chosen distress times, etc. Again, we emphasize that:

Sim produces a set of sail segments for each particle, and these sail segments become the legs for that particle, and behave and are treated just as the legs for a particle in § II.D

Equation 1: Converting S/V to §II.D

To do this, Sim needs to use the environmental data *during the pre-distress* part of the simulator, for each particle. Moreover, it needs an externally supplied function called to translate the environmental data into the set of possible velocities for the S/V.

### Basic Pre-Distress Movement of a S/V; Using Polars

We start by discussing how a S/V particle moves and for this, we start with the assumption that there is no current. Here, we introduce the *Polars* function.

This function is given by input data and provides a map between direction relative to up-wind and speed of wind, to speed of boat in that direction. Put another way, *Polars* is a function that is provided to Sim, its domain is the Cartesian product , and its range is . The input in the xml that specifies this function is the function evaluated at the cross product of a set of directions and a set of wind-speeds. The directions given typically range from 30º to 180º. They do not start at 0º because a sailboat cannot go directly up-wind. In fact, the first number in the set corresponds to the closest to up-wind that a sailboat can move, and the last number in this set corresponds to the direction closest to downwind that the sailboat can move. It is possible that a sailboat can go directly downwind, in which case, the final direction will be 180º. Sailboats never go directly upwind so the first direction will always be positive.

The directions given are usually some arithmetic sequence, so that a typical set of directions would be . The rest of the domain of the *Polars* function is the wind‑speed and the input data gives the function’s values for an arithmetic sequence of wind-speeds. The typical wind‑speeds given are . Using these two example sets, the function would be given by specifying the function’s values at separate combinations of direction-from-upwind and wind-speed, and we simply use interpolation is make the domain continuous.

A perhaps easier way to think of the input to Sim is that it is a sequence of tables. The tables in this sequence correspond to different wind-speeds, and each table has entries for different relative angles to up-wind. The values in these tables give the boat speed if the boat is travelling in that relative angle.

One can think of as either a function (with an implicit understanding that interpolations will be used for the evaluation of the function), or as a sequence of tables. We will be slightly imprecise and let context determine the appropriate interpretation.

Sim makes heavy use of interpolation. In fact, it starts by using interpolation within each table to make sure that all of the tables have the same set of relative angles. Then when asked to find the boat speed for a particular angle and wind-speed, it interpolates within each table, basing this 1st interpolation on the angle, and interpolates the two results, basing this 2nd interpolation on the wind-speed.

#### Sailing Port or Sailing Starboard; forbidden wedges

The function is given only for a set of relative angles; for concreteness, let’s say that these angles go from 30º relative to up-wind to 180º relative to up-wind. We assume that if a S/V is travelling (say) 330º relative to up-wind, then its speed would be what the function would give for 30º degrees. In other words, is given only for one side and we assume symmetry.

We say that a S/V is if the wind is coming across its port side, and it is if the wind is coming over its starboard side. For example, suppose the wind is out of the north so that up-wind is 000º. Also assume that a S/V’s target course is 000º, i.e., straight into the wind. If its course is 030º, which is as close to its target course as possible on the port side, then it is , and veering to the right of its target course. Hence, if a S/V is to make progress towards an up-wind course, then it will be veering to the right of its target course. If a S/V is towards a down-wind course, then it will be veering to the left of its target course. Similar statements hold for .

We use the term to mean the angles near up-wind that a S/V cannot go. If we wish to be explicit, and the S/V cannot go directly down-wind, we will use the term and , but for now, assume that we are only interested in the up-wind wedge. For example, if the wind is from 010º, and the function’s relative angles go from 30º to 180º, the will be from 340º to 40º. If the relative angles go from 30º to 175º, then the will be from 340º to 40º, and the will be from 185º to 195º.

#### Sailing with No Current

In this section, we assume that there is no current. With no current, we say that if a S/V’s target course is not in a f, then it can and our model is that it will do so. Otherwise, the target course lies within a , and we must decide whether to towards the target or . With no current, this is fairly simple; we simply take the endpoint of the forbidden wedge that is closer to the target course.

#### Using Polars with Currents

Until now, we have assumed that we have used with no regard to the currents. The function is only for the apparent wind against the water. If the water is moving (i.e., there is a current), then the apparent (up-)wind is the sum of the up-wind and (down-)current vectors, and the apparent wind’s speed is what we must use in the function. Moreover, the result of the function gives only the boat speed and direction against the water. To get the boat’s velocity against land, we must add in the current. If the wind is light and the current strong, this difference can be substantial, and the code dealing with it is quite involved.

When there is a current, Sim must compute the . The are the tables derived from the tables, after adjusting for the currents. This involves two steps; compute and use the , and add the current vector to each vector from the function.

For example, suppose the wind is 8 kts from 000º and the current is flowing 3 kts towards 090º. In the 1st step, Sim adds the two vectors (0,8) to (3,0) to get the . The ’s speed is and it is from . Hence we would “read” (using interpolation between the tables for wind-speed=8.0 and wind-speed=10.0) the table for . Furthermore, we consider up-wind to be 20.56º. Sim needs to get the vectors for both the port and starboard sides, because it will add the constant vector (3,0) to all of the vectors to get the true velocities against land.

In the 2nd step, Sim adds the vector (3,0) to each vector of the table that Sim got in the 1st step; symmetry between the port and starboard sides is lost in this step. The port and starboard sides will each still form a contiguous swath of courses, but these sections are not symmetric and indeed, are not even necessarily disjoint. Hence Sim’s strategy in the presence of currents changes.

Sim’s strategy is to “do its best ,” “do its best ,” and choose the better of these two possibilities. “Better” here is measured by progress along its target course. What remains to discuss is how Sim chooses a vector when the side (port or starboard) is known. If the current is very strong, this is quite complicated and for this discussion, we will omit those details.

When Sim is assuming a , Sim will choose a direct heading if it is within the ’s relative angles. Otherwise, it will choose the course that is closest to the target course as possible. Similar statements hold for .

### Sail Segments; Motoring and Hove-To

Henceforth, we assume that there is a current and we will use “wind‑speed” to mean “apparent wind‑speed.”

When the wind‑speeds given by the environment fall outside of the wind‑speeds given, we cannot always simply extrapolate the tables to get boat-speed. We do extrapolate when the winds are on the low side, but even there, there is a floor. There are properties in Sim.properties, together with additional data given in the xml, that tell us how low the wind can be before Sim quits the extrapolation for boat-speed. When the winds are on the high side, Sim never extrapolates from the tables, but simply uses the highest value wind-speed table. Again, there is a limit (ceiling instead of floor) to what the S/V can handle and this is given again by a combination of the xml and Sim.properties.

The four thresholds on wind-speed that the xml and Sim.properties supply are “enter motor wind-speed,” “exit motor wind-speed,” “enter hove-to wind-speed,” and   
“exit hove-to wind-speed.” These, together with the wind-speed determined by the environment, determine if the S/V is “motoring,” “sailing,” or “in hove-to.” If the wind-speed drops below the “enter motor wind-speed” value, the S/V will start motoring and continue to motor until the wind-speed rises above the “exit motor wind-speed” value. Similarly, if the wind-speed rises above the “enter hove-to wind-speed” value, the S/V will enter hove-to and continue in hove-to until the wind drops below the “exit hove-to wind-speed” value.

When motoring, Sim considers the effect of the currents, but not the winds. This appears different than the regular voyage of §II.D, because then, we assume the boat overpowers both the currents and the winds. However, this step of the processing is simply creating the legs that we will use as in §II.D and so our comments concerning the similarity of S/V and regular voyage still hold.

If the currents are *very* strong, the S/V can be overwhelmed by the currents even while motoring. When in hove-to, the S/V drifts roughly perpendicular to the apparent wind. When motoring, the S/V can avoid land. When in hove-to, landing means that the S/V’s voyage is over. When motoring, the S/V continues to try to make progress towards its target course, and if it is blocked by land, it will try courses in 5º increments until it finds a good one. While motoring, Sim will only choose a course that is more than 90º from its target course if it cannot find one that is within 90º of its target course. When in hove-to, there is no choice for a course; a S/V in hove-to behaves very much like a distress object.

### Stopping a Sail Segment

#### Stopping a Sail Segment; CheckIntervalTime and Time-Step

Sim assumes that the environment does not change for a fixed period of time, called the , and currently set in Sim.properties to one hour. The 1st way of stopping a is simply to use up the . Clearly, no can last longer than .

A 2nd way to end a is for the simulation time-step to run out. Currently, this is usually 20 minutes. Because of this, no Sail Segment intersects multiple time-steps.

#### Stopping a Sail Segment; CloseEnoughToAbandonOffset

Recall some terminology: each is broken into a sequence of . In the xml and Sim.properties, there is a specification of a distribution for the offset from the , that each particle should start with, and how long it should stay on that offset. For example, if a particle ’s is straight north, but ’s draw from the distribution mentioned above is -20º, then Sim will act (almost) as if the desired course is 340º instead of 000º. In addition, the xml and Sim.properties will specify a distance that must maintain 340º as its . When that distance has been achieved, Sim will then turn towards its original target , the end of the . Hence, another way for a to end, is if a particle gets “close enough to abandon its offset course.”

The reason for the “almost” is that Sim will never go more than 90º from its true target course. In this example, it will go from -70º to 110º away from the course it is trying to achieve. This issue comes into play when avoiding land and when there are extreme currents.

Sim does not require that a particle reaches the exact of its ’s termination point. Again, in the xml and Sim.properties are used to specify a threshold distance; let’s assume that it is 5 NM. Therefore, if gets within 5 NM of its target point (the end of its ), Sim will consider as having arrived, and the will end. If this is the last , ’s entire set of is complete.

#### Stopping a Sail Segment: the Projection Point

Sim never pushes a particle along a course that is more than 90º from its target course. It is possible during a , for a particle to reach a point, called its , such that if it were to continue in the same direction, it would be moving away from its target point; again, we emphasize that the target is the endpoint of the major leg. If reaches its , the ends.

#### Tacking Strategy

Sim currently has no tacking strategy. When there is a strategy for changing tacks, this will be another way for a to end.

### Avoiding Land

Sim relies on the human operator to “route” the sailboat by setting up the . For example, in Washington State, there is a long fjord called Hood Canal that separates the Olympic and Kitsap peninsulas. To follow this discussion’s geography, the reader can follow the link https://www.google.com/maps/place/Hood+Canal/@47.6,-123.1.

Hood Canal starts in the north at the Hood Canal bridge, extends roughly SSW for about 45 miles to Annas Bay, and bends back roughly ENE for about 15 miles to Lynch Cove. If a connected Lynch Cove directly to the Hood Canal bridge, Sim would fail; Sim would not be able to route particles to Lynch Cove and take the sharp right necessary to get to the Hood Canal bridge. The user must help in this case by setting up at least two s; one from Lynch Cove to Annas Bay and one from Annas Bay to the Hood Canal bridge. Even then, there are slight bends in Hood Canal that could prove troublesome. In particular, Sim would benefit from adding points that keep the particles from going into Dabob Bay.

Sim’s automatic land avoidance is not the “routing” type that would be necessary for getting from Lynch Cove to the Hood Canal bridge. Rather, the idea behind Sim’s land avoidance is that it can (sometimes) go around small islands. Given a (which is either directly towards its or the offset course discussed in §II.F.3.b), Sim will apply the logic to come up with a course. If that course runs into land during the , Sim will try alternate courses by adding and subtracting multiples of 5º until it can sail the entire . If it cannot find such a course that is within 90º of the course towards its , it chooses the course that closes as much as possible towards the , and then it backs off from the first crossing of the land. It will try again, but if it continues to be blocked, it will wait for the environment to change. After 3 attempts, the particle is considered blocked, and it stays there for the rest of the duration.

Note that, unlike the regular voyage of §II.D, particles may never arrive at their destination. In §II.D, particles are always scheduled to arrive at their destination, but they all go into distress during the simulation. In a S/V voyage, particles may get hopelessly blocked by land or even unfavorable winds and currents. Moreover, being blocked would become part of the particle’s pre-distress “itinerary.” For example, if a particle is blocked 1 hour into a 100-hour simulation then, when the time of distress is drawn, the chances that its distress point will be where it got blocked will be .

Although this logic is primarily used for sailing part of a S/V, even the motoring part of a S/V case is vulnerable to being blocked. This is because motoring progress is affected by the currents, which could conceivably (although it is very unlikely) be strong enough compared to the S/V’s motor, to cause problems. Finally, (as with Lynch Cove directly to the Hood Canal bridge above), a S/V might not find any direction that would close on its .

If after 3 attempts, the particle is still blocked, Sim gives up on this particle and its trajectory stops. The exception to this rule is if a particle is in hove-to and is driven into land, the particle stops immediately.

### Particle State, SVT, and SVTs for S/V particles

Every particle, at every time, is in a particular . Here, we use the word as in the expression “state of the union.” For non-sailboat particles, there are only two ; and . For an LKP, there is only . For an S/V case, there are several pre-distress s, but just the single distress . The S/V pre-distress s are:

1. Port Direct: Port side sailing, not on the edge of non-forbidden courses
2. Port Tack: Port side sailing, on up-wind edge of non-forbidden courses
3. Port Gibe: Port side sailing, on down-wind edge of non-forbidden courses
4. Starboard Direct: Starboard Sailing, analogous to Port Direct
5. Starboard Tack: Starboard side sailing, analogous to Port Tack
6. Starboard Gibe: Starboard side sailing, analogous to Port Gibe
7. Projection: Stopped because it reached the projection point
8. Close enough to abandon offset: stopped because it’s close enough to abandon the offset
9. Deflected: Sailing, deflected by land
10. Blocked: Sailing, but stopped by land
11. Overwhelmed: Sailing, could not close on target point
12. Overwhelmed and Blocked: Not only could it not make progress, but it got blocked by land in the direction it was going
13. Motor: Wind is too low, motoring
14. Motor Projection: Stopped motoring because it reached the projection point
15. Motor Deflected: While motoring, deflected by land
16. Motor Overwhelmed: While motoring, could not make progress because of the current
17. Motor Blocked: While motoring, blocked by land
18. Motor, Overwhelmed, and Blocked: Not only could it not make progress, but it got blocked by land in the direction it was going
19. HoveTo: Wind is too high, in HoveTo
20. HoveTo, Blocked: While in HoveTo, ran into land. This stops the particle permanently
21. Arrive: Either at end of Major Leg or Target

Table 4: States for Pre-Distress Sailboat Voyage Particles

The records the of a particle at each time-step. When a particle changes s multiple times during a time-step, only the last is recorded. This means, in particular, that a or a state will rarely if ever appear in the ; once the Projection is reached, the particle will immediately begin a new , most likely some sailing or motoring . These transitory s are useful only when stepping through the algorithms in the code. In §IX, we discuss these sailboat voyage pre-distress s further.

Because there is quite a bit of data associated with a particle at a given time (e.g., latitude, longitude, …), this data is collectively referred to as a . The is only one part of a , and hence it is usually referred to as a , or .

# Hazards and Distress Incident

For the pre-distress motion of a particle from a Voyage or LKP-Dead-Reckon scenario, a distress time and place must be chosen. The simplest way of doing this is to lay out a route for the particle and ignore distress considerations. Compute the beginning and ending time of the route and make a uniform draw for the time of distress. This is essentially what is done, but it is complicated by “hazards,” prescribed distress times, and the requirement that a particle go into distress over water.

A hazard is an area, a time-interval, and an intensity. The area is fixed, the time-interval tells us when the hazard is active, and the intensity tells us how much more likely particle was to get into trouble while in the hazard. Hazards do not affect the route of the particle; only when it gets into distress. To illustrate, suppose we have a particle whose planned route (ignoring that it will get into distress at some point) is shown in Figure 8. For example, this particle is 3 times as likely per unit time, to go into distress during times to . His must vulnerable period is between and ; there his “hazard score” is .

Let’s suppose the particle is travelling between and . If there were no hazards, a uniform draw between 0 and 20 could be used to assign the particle a distress time. But with hazards, we compute an “expanded time.” Note that the intensity of a hazard-free section of the trip is the same as a hazard with intensity 1, so our expanded time is 5-2). A uniform draw is made between 0 and 177, and is translated back to a time between 0 and 20. The code is somewhat complex since it must deal with overlapping hazard intervals, but the underlying idea is no more complicated than this example. It’s slightly more complicated going from a draw between 0 and 177 to a time, but the formulae are more cumbersome to write out and read than difficult to understand.

3

5

2

10

Figure 8: Intervals of a Hazard for an Example Particle

An additional complication is that, for hazards, there are Dwell Zones, and the hazards may turn on or off while “dwelling.” Still, the main idea of using hazards to determine a distress time, is captured in the above example.

Whether there are hazards, dwell times, both, or neither, the distress time for a particle can be computed with the aid of a single uniform draw. There is an additional wrinkle though; the distress *interval* can be given within the xml. Sim simply computes a distress time that is within this interval by repeatedly making draws until it gets a distress time in the interval.

The same strategy is used if Sim.xml states that the particle should go into distress in water. For example, if the pre-distress motion is modelling a plane flying over an island, then the human planner might not want particles to start on that island. If a particle does go into distress over land, again a draw is made.

This strategy of repeating draws until something acceptable occurs could result in too long a runtime. Sim has code in place that prevents this, and it is summarized in the following rule:

If at least 1000 draws have been made, and Sim has been successful less than one percent of the time, the next draw is accepted. Such a draw is not counted as a success. The numbers of draws and successes accumulate across the particles.

Equation 2: Formula for Ensuring Particles get Drawn

Finally, we mention a rule that is implemented in the code that calls Sim.

If a distress interval is given, and the maximum start time of the particles is after the beginning of the distress interval, the distress interval is ignored. Similarly, if we apply the maximum speeds between the means of the Dwell Zones, and the earliest possible start time for a particle, and the result is an arrival time that is less than the end of the distress interval, the distress interval is ignored.

Equation 3: Ignoring a Distress Interval

What this enforces is that the operator should not specify a distress interval that includes times for which some of the particles are not in distress. This is easy to enforce on the early side; we know the latest time that the particles can go into distress. In other words, it’s easy to compute the “latest possible pre-distress time” for a particle.

The “earliest possible post-distress time” seems like a strange concept, but we use that expression only to be analogous to “latest possible pre-distress time,” and we mean the earliest possible ‘arrival’ time.” Here, ‘arrival’ means that the subject would have completed its scheduled itinerary. This is not nearly as easy to compute, and so we use the fastest speeds, earliest start time, and the *means* of the Dwell Zones. It’s not hard to create pathological cases where the earliest arrival time is considerably earlier, but the technique we use will work for practical cases.

# Reverse Drift Scenario

Sim runs “reverse drifts.” The situation being modelled is that debris is spotted, and a reverse drift is used to find the distribution for distress point. A subsequent “forward drift,” (i.e. normal case) would be run.

The only scenario types that are allowed in reverse drift cases are those with no pre-distress motion. Furthermore, we cannot process completed searches in reverse-drift cases. A typical reverse drift is just one or more of LKP and Area scenarios. The ’s times must be ascending, but in a reverse drift, the natural way of computing the time-steps is descending. Furthermore, the environmental data must be turned around to drift back in time.

The strategy within the code is to run a forward drift, and use the concepts “RefTime” and “SimTime” to distinguish between the times in this forward drift, the real times, and the times used in the environmental look-ups.

# Movement of Distress Particles; Using Environmental Data

A particle’s path is given by its pre-distress motion and its distress motion. In §II.D and §II.E, we discussed the only two scenario types that *have* pre-distress motion, and in §III, we discussed how we determine when pre-distress motion stops. Distress motion starts immediately after pre-distress motion (or at the start when there is no pre-distress motion), and we discuss distress motion in this section.

Distress motion is determined by the environmental data and a particle’s object type. We compute two vectors, the vector and the vector. We simply use vector addition and add these two vectors together. We do this for each particle and each time-step.

There are 3 steps to determine each of the two vectors for a given particle at its LatLng for a given time:

1. Determine the mean and of the winds or currents
2. Add a random effect to these means to form the
3. Compute the vector from the .

Table 5: Using the Environment to Move a Particle

The first would be easy if we had the mean and speeds for every point in the ocean at every time, and of course we do not. What we do have is these mean speeds for a set of LatLngs and a set of times. The set of LatLngs do not necessarily form a grid, and the times are not necessarily evenly spaced.

For any LatLng that appears in the data, we make sure that we have data for any time that appears in the data by simply filling in the values from the closest times. Hence we may assume that we have and speeds for any LatLng/time combination of the data.

But a particle will almost certainly not be at a LatLng (and quite probably not at a time) that is in the data, and we will use interpolation to get the mean and speeds for the LatLng/time combination of interest. We use the LatLng to interpolate within each of the surrounding time layers, and then we use a linear interpolation of these results based on time. So for example, if we are interested in finding some mean speed for some LatLng at time , and we have time layers corresponding to and , we will use to get the answer within the layer, again use to get the answer within the layer, and the time interpolation will be .

## Non-Riverine Interpolation within a Time Layer

Note that the above does not address how to get the mean speed at within the or time layers. This is different for the riverine currents than from the ocean currents or winds. Here we discuss the winds and ocean currents. Our interpolation here is not linear, but based on the 3 closest points within the time layer for which we do have data. The weights applied to the data are simply the inverses of the distances. Suppose we have a LatLng and the 3 closest reference points are . Suppose furthermore that is from , and the mean speed at is , for , the result is

Equation 4: 3 Closest Point Interpolation

The most challenging part of this problem is to arrange the data so that finding the 3 closest points is quick. Since this computation is done for every time-step and every particle and for both winds and currents, a fast algorithm for finding the closest 3 points is necessary for Sim to run. We provide a very brief sketch of the data structure at the heart of this algorithm. This structure is implemented in the Java class MassFinder.

A MassFinder is essentially a binary tree of cells, with each cell being either a MassFinder or a list of up to 20 LatLngs. The two cells of a MassFinder are called its and their LatLngs are separated from each other by a constant latitude or a constant longitude. The cells that are lists of LatLngs are called . Given a LatLng , it is easy to find the containing , and there is only one. Here we are taking advantage of having *points* rather than *edges*. The situation is far more complicated in §V.B and §VI and §V.B, where the reference objects of interest are *edges* and not points.

But given the containing , we can at least find the 3 closest points in ’s list of points and use these to initialize the search for the closest points. We then traverse the tree from the root, skipping cells that we know simply by their borders, cannot have points close enough to affect our result.

## Riverine Currents Interpolation within a Time Layer

Computing the mean and speeds for the currents in a riverine case is far more complicated. The data itself must have more information. Except for riverine currents data, each data point must have lat, lng, time, , and . For riverine currents data, we need the additional data , , and . We use an example to describe what these are and how we use them.

Consider a case near the junction of the Willamette and Columbia Rivers. Each data point must identify whether it is from the Willamette or the Columbia and it does this with a . In addition, our model is that there is a current in the center of the river, a current near the river’s left bank, and a current near its right bank. Each data point must identify which of these “strips” it is in; , , or , and that is the data-point’s . Finally, within a strip, the data is sorted where upstream is considered to be before downstream. The sorting is done by the , which identifies where in its strip it occurs.

We sort the data points within a river/strip combination, and this forms a sequence of points and then a sequence of edges. Rather than finding the 3 closest points as we did in §V.A, we base our interpolation on these edges. There are two possibilities; Center-Dominated and Not Center Dominated. Suppose we are interested in finding the mean for some LatLng .

In Center-Dominated interpolation, we find the nearest edge within some Center Strip and consider the point in that edge that is closest to , and assume that it is from , and the course from to is . If we check to see if we hit a side-strip edge when going in the direction , we will use that edge instead of the center edge.

If we are not using Center-Dominated, we simply find the closest edge. The default is Center-Dominated.

Regardless, we will have an edge and we will not interpolate between its upstream point’s data and its downstream point’s data. Rather, we simply use its upstream point’s data.

## Perturb the Data

In the previous two sections, we discussed how to get the mean and speeds for a given Lat/Lng/time. This Lat/Lng/time is where the particle is that we are trying to move with environmental data.

We consider winds and non-riverine currents first. We could perturb the and means that we found in the introductory part of this section and §V.A each time by making independent draws for each of these. Let’s suppose we are making a draw to perturb the component of the winds. Suppose we have a particle , we are at time-step 1, and the most recent time-step was time-step 0, and the perturbation at time-step 0 was . We need a perturbation for this time-step. This will be correlated to the perturbation at time-step 0.

The correlation will be stronger if the two time-steps are close together. A constant called the is stored in Sim.properties, and it, together with the difference in the times, determines the correlation . This constant represents an exponential decay in the correlation so that if , , and if , . Given , and an independent draw , we have

Equation 5: Correlated Draw for a Variable

It is interesting to note that and do indeed have correlation , and that is a variable. This formula works only with Gaussian variables.

For riverine currents data, the situation is more complicated. Up to this point, we were treating the and components separately. Furthermore, we were storing the draws, but they were scaled by some standard deviation stored in Sim.properties.

For riverine currents data, we convert the mean and speeds into a downstream vector. Each particle has a constant downstream perturbation for its speed, but there is also a cross-stream component. Saying “there is a constant downstream perturbation” is the same as saying for the downstream component. In contrast, for the cross-stream component, . Moreover, the standard deviation that we multiply these draws by is not given directly. For riverine currents, the standard deviation is a percentage set in Sim.properties of the mean speed; this percentage is current set to . The standard deviation for the cross-stream draws is a fraction of the standard deviation of the downstream standard deviation. This fraction is also stored in Sim.properties, and is currently set to .

## Computing the Vector

In the previous sections of §V, we have found the and speeds of the environment (wind or currents) that we will assume are acting on a particle at time t. We directly use this vector as our vector. For winds, the situation is vastly more complicated and it is here that we have dependence on the Object Type.

This section, and the resulting code, is based on [LEEWAY].

Each Object Type has a Leeway Data, and its components are given in Table 6.

1. Slope and
2. Spread
3. Slope and
4. Spread
5. Slope and
6. Spread
7. Gybing Frequency per second
8. Nominal Speed
9. Use Rayleigh (Boolean)

Table 6: Leeway Data

For each particle, we make 3 random draws and convert a Leeway Data into that particle’s Leeway Calculator. The data of a Leeway Calculator is given in Table 7:

1. Slope and
2. Slope and
3. Slope and
4. Gybing Frequency (Copied over from Leeway Data)
5. Using plus (Boolean)
6. Next gybing ‘flip’ (point-in-time seconds)

Table 7: Leeway Calculator

For an individual particle, the first 4 entries of this table are constant; the last two are used to determine when to switch from crosswind-plus to crosswind-minus, and these change during the simulation.

## Creating a Leeway Calculator

A Leeway Calculator requires a Leeway Data and 3 truncated draws; one for each of . We will describe how the Slope and of a Leeway Data is combined with a random draw, to get the Slope and of a Leeway Calculator; the fields of a Leeway Calculator corresponding to and are created similarly.

A Leeway Data has a Slope and . These are illustrated in Figure 9. There, is the slope and is the . This is the standard equation of a line from high school math. Note that there is a Downwind spread value also in the Leeway Data, and our goal is to pick random “lines” so that the spread of the s at the nominal speed is the prescribed spread from the Leeway Data.

There are three ways of achieving this spread. We use one way if , another if , and a third way if the user has requested that we use a Rayleigh distribution. We skip the details here.

Particle Speed

Figure 9: High School Equation of a Line

## Using a Leeway Calculator

Once again, we emphasize that each particle has its own unchanging line similar to the one in Figure 9, and that the of these lines at the nominal speed, will have a spread that is specified in Sim.properties. It actually has one of these for each of , . We call these the *affine* functions of , .

When the assumed value for and are acquired from §V.C, they are converted to speed and direction, and the speed is used for the in the particle’s Leeway Calculator’s affine function to get the particle’s downwind speed.

We use the same downwind speed in the or affine function to get the particle’s crosswind speed. The crosswind direction is from . We stay with one of the choices until the next gybing flip, flip to the other one and, at the same time, make a draw to determine how long to stay on that one.

Adding the and vectors together gives us the vector.

# Land Processing

Distress particles are driven by the environment as in §V. For each time-step, this results in a small edge. It’s small because the distress particles do not travel very far in a given time-step, which is typically 20 minutes for a non-riverine case, and 2 minutes for a riverine case. Still, that can be enough time to cross some land boundary. Sim supports two modes of simulation; sticky and slippery. With a sticky mode, the particle will “stick” to the first land crossing it encounters. With the far more complicated slippery case, Sim will attempt to keep the particle moving. Either way, it’s worth taking a minute to outline how Sim *finds* the first land-edge that it crosses. To do that, we must describe the shoreline data that we use.

## Shoreline Data Characteristics

We use some fairly intuitive graph theory terminology here. Because all of our edges are “directed,” (i.e., they have a start and an end), we are working with a directed graph, and we will call the points , but we’ll continue to call the edges and not use the more common term within directed graphs “arcs.”

Our shoreline data is a set of nested polygons. Each polygon is a sequence of LatLngs (i.e., vertices), and these induce a sequence of edges. Furthermore, we will assume that there are no edges of zero length, so each edge has both left and right sides. Because each edge is a member of a polygon, it has exactly one predecessor and one successor. Except for an edge’s predecessor and successor, no edge so much as touches another one, much less crosses it. Every edge is a member of a closed polygon. We summarize these considerations, plus one more in Table 8.

1. For each edge, its its
2. Each vertex is the of exactly one edge and the of another
   1. If this were an undirected graph, graph theorists would call this a graph
3. Each edge is a member of a polygon
   1. In other words, there are no “spurs.”
4. Polygons that represent lakes or inland seas, are clockwise and those that represent continents or island are counter-clockwise
5. Equivalent to #4 is “Land is always to an edge’s left.”
6. Except for an edge’s predecessor or successor, no edge touches another edge

Table 8: Characteristics of Land Edges

It’s interesting to note that a directed graph is a collection of cycles if each vertex has exactly one and one . While debugging, this simple fact is often used; if ever there is a vertex that does *not* have exactly one and one , something has gone wrong.

The statement “set of nested polygons” is vital. It means that no polygon crosses itself and no polygon crosses another polygon. This rather long way of stating that no edge can touch an edge other than its predecessor and successor, is reminiscent of the two cardinal rules of marriage and the card game bridge: never play with your spouse as opponent and never play with your spouse as partner. Both the way of stating that there are no crossings in our data, and the advice about playing bridge and being married could be shortened. However, in both cases, the long form provides much-needed emphasis.

There are other characteristics of the polygons we insist on to avoid pathological or degenerate cases. Not only do we have no degenerate edges, but we also insist on every polygon having at least 3 edges. This, together with the other requirements, guarantees that every polygon has a well-defined interior and a well-defined exterior.[[1]](#endnote-1)

It is likely that all of these conditions are met in our data, because Sim uses the land that is produced by the BuildSimLand module of Simulator, and it is responsible for guaranteeing that. These very rigid requirements are not met in most data that we’ve seen. By far, the best one is the backbone of our data, GSHHS ([GSHHS]). To our knowledge GSHHS data satisfies the requirements for our data. Most land data seems to be used primarily for display, and the nesting of polygons is not required for that. In contrast, we are using this to determine whether a given LatLng is on land or in water and when a closed path crosses from one to the other. To answer these questions, one needs the conditions laid out in this section.

## Land Storage and Truncation of Polygons

There are roughly twelve million edges in the data. The class we use to represent an edge is the GreatCircleArc, but this is fairly heavy, and there is nothing gained by converting the data’s edges to twelve million GreatCircleArcs. Rather, we cache the file’s data in memory in the very succinct way that the file stores its data. Each polygon has a header that includes the number of vertices, followed by the vertices lat and lng values. These are stored as integers which represent the number of degrees. Then for each case, we get only the data we need by reading through this cache rather than the file on disc and get only the part of the polygon that we need for the case. Of course, we must be careful to retrieve nested non-crossing polygons.

Furthermore, we must do this quickly. Suppose we have a case off of Florida and we start to read in the North America Polygon (). If we were to examine each of the over one million vertices in for whether or not it was of interest to our Florida case, we would waste a lot of time. For example, we would be considering a point on the Yukon River, then the next point on the Yukon River, etc.

We avoid this by using the result of BuildSimLand; it generates files that tell us about the maximum lengths of the edges. For example, if the maximum edge length in the data is , and the point on the Yukon River is from the closest point of interest for our case, we can simply skip the next points.

A Sim case comes with an Area of Interest, or and this is given as a Lat/Lng box. Because of the above, it is convenient to quickly find the distance to the region of interest and it is easier to do this for a circle than for a Lat/Lng box. Hence, Sim computes the Circle of Interest or , which is the smallest circle containing the . The distance to the is simply the distance to the ’s center minus the ’s radius. Of course, distances in this case refer to great circle distances.

It’s interesting to note that truncating a polygon can result in multiple polygons and this is most often the case. In Figure 10, the blue circle represents the and the red polygon will be truncated into three polygons. Also in this case, there are four sections of the polygon’s that intersect the . Assume that the red polygon is clockwise, so the resulting polygons are formed by:

1. A and part of the
2. B, part of the , D, and part of the
3. C and part of the

Table 9: Polygons from Sections of a Polygon

A

B

C

D

Figure 10: Truncation Produces Multiple Polygons

This illustrates that here can be polygons formed form non-consecutive stretches of the original polygon. This is quite common and is not an unusual pathological case.

“Bridges” that fill in for the sections that are “part of the ” must still be constructed. This usually is not a problem except in the Great Lakes. Consider a case where the intersects Isle Royale, a large island in Northern Lake Superior. Then there will be three polygons that require bridges; , the Great Lakes polygon (), and the Isle Royale polygon (. We form the ’s bridges by extending them the farthest outside of the , ’s bridges by extending them less, and ’s bridges by extending them the least. We see this in the left-hand picture of Figure 11. The sliver on the edge is the part of the truncated that is not overwritten by the truncated version of . Although it is difficult to tell the difference, is a different color from ; this program colors islands (e.g. ) differently than continents or ocean-islands (). Also notice the significantly different blues between ocean (outside of ) and lake.

The second picture in Figure 11 shows the non-truncated version of and . We’re not zoomed out far enough to see the entire version of either of these, but they are there, as is , which we don’t see at all.

In Table 10, we list the different levels and show how that applies in this case. “Island-within-Lake” is certainly adequate for Sim, but the data does have some lakes-within-islands-within-lakes-within-continents.

1. Dark Blue: Ocean (not contained in any polygon)
2. Dark Tan (barely visible strip between ocean and lake): Continent or Ocean Island (
3. Lighter Blue: Lake within Continent ()
4. Light Tan: Island within Lake ()

Table 10: 4 Levels and their Colors

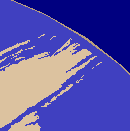
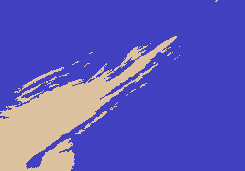
 

Figure 11: Truncated Island inside of Truncated Lake inside of Truncated Continent

Note that, even though the extends a long way north and east, as far as this case is concerned, there is ocean where the dark blue is in the first picture. Furthermore, note that the polygons do not cross. They are, to be sure, very close, but the bridge building process guarantees that the truncated polygons will not cross and will maintain the inclusion relationship of the original polygons. This screenshot was taken with some parameters adjusted to make the sliver of large enough to see. In practice, the bridges are much closer together.

## Storing the Edges; GeoMtx

When storing *points* so they can be found quickly, a binary tree of sets of points has been adequate for our purposes. Edges are not so easy to store, and that’s because edges are 2-dimensional whereas points are 0-dimensional.

For riverine interpolation, we need to find the closest edge to a given LatLng. For finding land crossings, we need to find the first edge that another edge (the particle path during a time step) crosses. After a few attempts failed because of undue memory usage or slow performance, we are now using a class called a GeoMtx. This class was primarily developed for BuildSimLand, but it’s useful for the two of this paragraph. Here we give a rough description, mentioning other Java classes that are involved.

An Extent is essentially a LatLngBox, and a GeoBox is something that can provide an Extent. The most common GeoBox is a GreatCircleArc (or ). In fact, other GeoBoxes are collections of GeoBoxes, so really every GeoBox has at its core, a collection of s. Polygons, (the Java class is Loop), and strings of s (GcaSkein) are also examples of GeoBoxes.

A small note here. For most of these GeoBoxes, the Extent is easy to compute. For a however, only the west and east longitudes of a can be directly obtained by considering their endpoints. The the north and south extremes of a do *not* necessarily occur at an endpoint, so they are computed and cached as part of a .

For right now, we say that “we sort GeoBoxes.” They are sorted by increasing left longitudes, and then sections of this sequence are sorted by increasing low latitude. For the rest of this section, simply assume that there is nothing of interest anywhere near the dateline.

The critical point is that with the sort and a little extra storage, one can identify only those GeoBoxes whose Extents overlap a given Extent. There are too many details to cover in this document, but let’s consider the left-to-right sorting by left longitude and suppose we are given an Extent with . Any GeoBox in our list with a left endpoint of 10 or higher cannot overlap so we can eliminate it and everything to its right, and this took only a binary lookup. However, of the ones we kept, it is much more difficult to eliminate the GeoBoxes that end to the left of 5. This is where the extra storage comes in.

We wrap the GeoBoxes in a class called BoxWrapper, and it is actually BoxWrappers that we sort. Each BoxWrapper has a list the BoxWrappers to its left that have a right-hand endpoint bigger than its own. This allows us to iterate through a small set of BoxWrappers until we will know that we have not missed any BoxWrapper with an overlapping Extent.

Finding the closest , as is required in riverine cases, capitalizes on this sorting scheme as well. As we find something that is close, we tighten the bounds and therefore eliminate some of the BoxWrappers from consideration. The details are too extensive, but summarizing, we have:

A GeoMtx is a collection of GeoBoxes, and we sort the GeoBoxes so that looking up which GeoBoxes intersect or are close to a given Extent, is fast.

Equation 6: Main Idea behind a GeoMtx

### The Dateline is a Nuisance

“We sort by increasing left longitudes.” Such a statement begs the question “what about the dateline?” There are very few edges that *cross* the dateline. Eurasia crosses the dateline by over of longitude, but there are not many edges that individually cross the dateline.

Our approach is to simply take those edges out of the picture and set them aside. Because our edges are relatively small, the edges that cross the dateline are few and are confined to a very narrow strip of longitude. When we ask any questions such as “what is the closest edge,”, we process this strip, called the separately.

It is interesting to note that the dateline still causes problems and we once again consider Eurasia. After we remove the edges that cross the dateline, we sort the rest by increasing left longitude. For North America, which does not cross the dateline, the edges will form a sequence of edges that is unbroken with respect to longitude. By this we mean that for any longitude between the left-most longitude and the right-most longitude, some edge covers that longitude. This is not so for Eurasia. After the Bering Strait, there is a huge gap in longitudes to the east, until Western Spain, and this gap causes problems.

We include this section simply to point out that there are details about the dateline that we are aware of and have dealt with. We have worked around these problems, and it is better to picture a problem that does not cross the dateline. The workarounds involving the dateline are mostly confined to the and including this in the discussions clutters the exposition without really providing any interesting information.

## Slippery Shore, Stop-and-Block

In §VI.C, we gave a very brief discussion of a data structure that allows us to look up nearby edges. The data structure is far less important than noting that we have addressed the task of quickly looking up edges that are close to or cross a given edge. We do this by loosening the problem slightly and solving the analogous problem concerning LatLngBoxes, which we call Extents.

In this section, we use this ability to find where particles hit land. Slippery Shorelines also require some additional considerations.

Given a large set of particles near a complex shoreline that has a lot of edges even after truncation, and many time-steps, the discovery of where the particles cross the shoreline can be the most time-consuming step in the simulation. We address this with a “Stop-and-Block” approach to the problem.

All of this is relevant only for the more difficult problem of Slippery Shore. For Sticky Shore, once a particle hits any shoreline, it sticks there and is essentially removed from the problem. Slippery shore is the far more difficult problem and, since riverine cases use Slippery Shore and 2-minute time-steps, handling them efficiently is critical for some cases.

For a Slippery Shore case, when a particle hits shoreline, Sim backs it off, and tries to move it parallel to the shoreline edge that it hit. That would be fine if it did not hit another edge. That this can happen repeatedly is apparent when one considers small inlets that the particles get “stuck in.” Repeatedly trying to get a particle out of such an inlet wastes a lot of time and we address that problem here.

In a given time-step, Sim computes the distance that the particle was supposed to move, as discussed in §V. If it is blocked by land, Sim moves it parallel to the blocking edge to use up the rest of . Of course, the particle can run into more land and, after 3 tries, Sim gives up and the particle’s end-point for that time-step is where the particle ended after the 3rd try.

But Sim does more than that. When such a “dead-end” time-step occurs for a particle, that particle is put in the “penalty box,” and there it will stay for 3 more time-steps. The idea is that the particle seems to be immobilized by the winds and currents at that time, and we wish to let the environment change before trying to extricate it from the trap that it has worked its way into.

We call this Stop-and-Block, and we log the occurrences of Stop-and-Block, and the penalty boxes. The problem is particularly acute in regions that have intricate shoreline and rivers that the particles can get up such as the Potomac, the Rappahannock, the York, and the James Rivers of Virginia.

## SimLand and BuildSimLand

We close §VI by mentioning that there is another document describing the data that Sim uses. We call these data files SimLand, and it is put together from several sources, plus corrections as the need becomes apparent. The bulk of the data comes from GSHHS, which in turn is largely a cleaned-up version of WDBII. The combining of all this data, and the modifying of it, makes extensive use of a GeoMtx.

# Bathymetry and Anchoring

Sim uses bathymetric data to determine where a particle, once it goes into distress, might have been able to anchor. There is a single anchoring depth of 50 meters, set in Sim.properties, and overridable in a case’s xml file. If a particle goes into distress at a LatLng where the depth is less than 50 meters, there is a probability that he will anchor, Sim makes a random draw, and the particle either sticks to that LatLng or drifts with no further potential for anchoring. The probability is given in the xml.

To determine the depth of a LatLng, we use ETOPO-1 ([ETOPO]), which is provided by NOAA. For the Great Lakes and, indeed, any lake for which the depth is not simply given by the height of the earth’s floor compared to sea-level, we must do more than that.

## ETOPO

The ETOPO data gives us the height above sea-level of the ocean floor at each point in a grid of points. The data is stored as a short integer, so it has a range of . The grid has rows and columns. The extra row corresponds to having an entire row for the North Pole and an entire row for the South Pole. For longitudes, we have a column for -180 but we don’t need one for 180.

When we are given a LatLng , we do not interpolate, but rather find the grid point that is closest to and use that depth.

The format of the data file is extremely simple. It is simply a binary file with 2 bytes for each depth, and the depths are stored in rows that go north-to-south and, within a row, from to .

The code uses a combination of skips and byte reading to pick up the necessary depths. It is all very straightforward except for cases that cross the dateline. No matter what, the set of rows and columns are identified, and the rows that are north of the relevant rows are skipped over in the file. An array of rows is allocated within memory, with row having room to store the number of columns in the data.

Following that, for a normal non-dateline case, the initial part of the data for the first row is skipped over, the relevant part is read into the first row of memory, and the rest of that row is skipped over in the data file. The process repeats until we have read in the entire row.

For a dateline case, the rows are allocated as before, and data in the file is skipped over to get to the correct set of rows. But the process changes here. In this case, the eastern part of each row must be read in to the right side of the row in memory, a single skip is made in the data to get to the western part of the data in the file, and the western part of the data is read into the first part of the memory row. A short example will help.

Suppose that the LatLng box has longitudinal range of from to . We allocate shorts for each row. But we start by reading in the shorts from to into positions through of our array in memory, we skip the shorts in the file to get to , and then read the final shorts into positions through of our array in memory. Then we repeat this process for each row.

## Great Lakes

The above process is straightforward when depth and meters-above-sea-level are each other. However, in the Great Lakes, ETOPO gives not the depth of the lake, but the elevation of the lake floor above sea level. Therefore, we must adjust these depths by the altitude of the lake’s surface. To do this, we created “Lake Basins.” These have nothing to do with the drainage basins of the lakes, but are simply a convenient construction for adjusting the depth information. In Figure 12, we depict the lake basins of the Great Lakes. There are 4 because Lakes Michigan and Huron have the same surface level. We certainly could have made them much smaller, but there is no need to and, since we will be checking if points are within these polygons, it is helpful to keep the number of sides in these polygons down.[[2]](#endnote-2)

We examine these basins in the order Superior, Michigan-Huron, Erie, and Ontario. If a point is in a basin, its depth from the ETOPO data is adjusted accordingly. The basin and their adjustments are stored with ETOPO data itself in Java resources, and are included in the Land Jar file.

In theory, we could do this for other lakes, such as Champlain and Tahoe. But the data within ETOPO does not support this. In ETOPO, we have only found lake bottom depths for the Great Lakes.

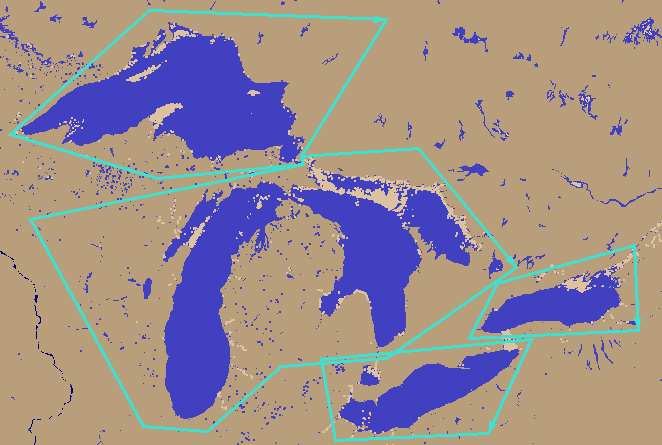


Figure 12: Lake Basins for Superior, Michigan-Huron, Erie, Ontario, and Champlain

# Completed Searches

Sim computes probabilities of particles. It can use completed searches to adjust the probabilities of the particles. Strictly speaking, particles do not have probabilities; our underlying probability model is not simply many point-masses. But for this discussion, we will ignore this inaccuracy and refer to the weights on the particles as probabilities. We would use the word “weight” interchangeably with “probability,” except that if we are using the word “probability,” we will assume that these numbers (weights or probabilities) have been normalized.

With those technical details out of the way, we will simply talk about completed searches updating the weights. When they are updated, we can normalize them, i.e. divide each by the sum so that they add to 1. Initially the weights are determined by the scenario weight, and the number of particles for a given object type is determined by that object type’s weight within the scenario. A short example will illustrate this. Suppose there are 2 scenarios with weights 0.6 and 0.4 respectively, and two object types, with object type weights within the first scenario of and respectively. Suppose also that we are using 5,000 particles per scenario. Then the initial weight assigned to any particle in the first scenario is *and* there are of these particles. We will use 1666 particles of one object type and of the other; they must both be integer and whether we give the remainder to one object type or the other is immaterial.

## Lateral Range Curves and

The completed searches component of Sim adjusts these weights by considering the previous searches that failed. Intuitively, a particle that had a high probability of being seen by a previous search should have low weight; “we looked there, we did not see him, so that particle is probably not the one.” To capture this notion rigorously, we need the notion of a particle’s . A particle ’s is the Probability of not seeing , given that is the missing object.[[3]](#endnote-3). The rest of this section deals with how we use Lateral Range Curves to compute the .

Note that this computation does not affect the paths that Sim produces; a particle’s path is completely determined by the pre-distress motion model and the movement of the post-distress particle due to the environment. Only after the path is laid out, does Sim use the completed searches and these paths to update the weights of the particles.

We give a very brief discussion here about updating these weights. An excellent discussion is found in Chapter 11 of [ELEM]. In fact, we will refer to that paper and say only enough that we can describe what Sim does that is not in [ELEM].

In [ELEM], the main discussion focuses on the use of a Lateral Range Curve (*LRC*) to update a particle’s weight by using the . The for a given leg is computed by finding the point on the leg where the distance between the searching vehicle () and particle, when plugged into the *LRC*, is maximized. For the *LRC*s in [ELEM], and for most of ours, this point is the point of the minimum distance between the *SRU* and the particle. We call this distance the closest point of approach (or *CPA*)[[4]](#endnote-4). Hence, in [ELEM] and for most of our *LRC*’s, we simply find the *CPA*, and plug it into the *LRC*, and call that our probability of detection, or , for that leg. is simply .

## ESS

Because the *LRC*’s discussed in [ELEM] are omnidirectional, symmetric about 0, and monotonically decreasing, this method does indeed produce the maximum *POD* for that leg. Sim deals with two sensor-types now that are not. One is *ESS* and the other is *ESS+NVG*. We start our discussion with *ESS*.

*ESS* is not omnidirectional and in fact, can only be used for ladder patterns. An *ESS* sensor looks only “down-creep.” Furthermore, there is a minimum distance and a maximum distance that it can see. Finally, it is not only blind to up-creep, but its angle of view is very limited. In Figure 13, we assume that the *SRU* is flying from the origin along the positive axis, and we illustrate the region that the *SRU* can see.in Figure 13.

Detection Region  
(or Region)

Down-Creep

Figure 13: Non-zero Detection Region for ESS

By subtracting the *SRU*’s straight-line velocity vector from the particle’s straight-line particle vector, we can assume that the *SRU* is stationary, and that the particle is moving at almost the *SRU*’s, velocity, but right-to-left instead of left-to-right.

Let’s suppose that the particle is moving directly right-to-left and near the minimum range from the x-axis. There would be up to 5 different segments for the particle; outside and to the right of the region, inside the region and near the lower right corner, outside the region because it is too close but within the angle of detection, inside the region and near the lower left corner, and outside and to the left of the region. Only two of those segments, the 2nd and 4th were within the region.

To compute the maximum *POD*, we take the *CPA* within each segment for which the particle is in the region. In this case, there would be two segments. Almost always, there is one.

Because the particle moves so slowly compared to the *SRU*, in reality, one could approximate the detection region as the region between the two red bands in Figure 14. In that case, there would be no need to specify the angle of the detection region.

Down-Creep

Detection Region

Figure 14: Practical Detection Region

## ESS+NVG

Most of our ESS sensors are combined with NVG, which is a more standard lateral range curve. We combine them into a single *LRC*. Recall that we found the time for which the distance between the *SRU* and particle at , maximized the *LRC*. We do the same thing here, but the LRC that we use is evaluated by assuming independence between the two sensors and taking the product:

Equation 7: LRC For ESS+NVG

Because the *ESS* component is not omnidirectional, the *LRC* does not have that form; that form would be true if the were the same for any direction, but it is not since *ESS* only looks down-creep. Therefore, our *LRC* is not a simple function of the distance.

However, we use it almost as if it were. We do not simply plug in the distance , and we cannot do that for *ESS* either. There *is* a function of distance that we use to get , but we must make sure that the particle is within the sensor’s detection region. Otherwise, the term is replaced by 0.

For *ESS+NVG*, we find the *CPA* of each section, but apply it to Equation 7, modified as per the above paragraph. Then among the segments, we take the one with the highest value.

## 5-Minute Rule

We compute the *POD* value for each leg. Note that on cross legs, the *POD* value is zero for *ESS* and the ESS component of *ESS+NVG*. But suppose there is a detection near the end of a leg, and this detection was a good one primarily because of the *NVG*. The NVG component operates on the cross legs and there will be another one almost immediately. We do not wish to count both of these, so we use what we call the “5 minute rule.”

Roughly speaking, if two detections occur close to each other, we only wish to count one of them, and the better one at that. A leg’s *detection time* is the time that gave it its highest *POD*. We start by calculating the *POD* value for each leg and putting the legs onto a cafeteria stack.

We set a pointer to the last leg in the stack. If the leg pointed at by and the leg before that have their s within 5 minutes, we take the leg with the larger *POD*, throw the other one away, put the winning leg back onto the stack, and keep pointing at it. If these two legs have s that are more than 5 minutes apart, we keep them both and move down one, so that it’s now pointing at the second leg in the comparison.

## Common Roots with Planner

Almost all this work pertains to Planner as well. Indeed, Planner simply generates a pattern from the parameters of the pattern, and then applies this method to compute the POD for a given particle/SRU combination.

# Particle File

Finally, we discuss the main output file of the Simulator; the . The is a NetCDF file (See [NetCDF]). We give a functional description of the relevant characteristics of NetCDF files here. By this we mean that we are *not* trying to specify a byte-by-byte description of NetCDF files, but rather we describe the concepts behind the format. Indeed, we do not even *know* the byte-by-byte format, but rely on 3rd party libraries to read and write NetCDF files, using the data within our own code in the calls to these library routines. Finally, we mention that we are simply scratching the surface of the NetCDF format. We use a very small subset of the capabilities of NetCDF files.

Our main purpose for using the NetCDF file format to store our is that it is a very convenient way of storing multi-dimensional arrays (or tables) of data. The process is straightforward:

1. Declare the different types of s, and how long they are
2. Declare the arrays (called ) and what they are dimensioned by
3. Populate the arrays
4. Add the global strings

Table 11: Steps for Creating a NetCDF File

## Dimensions and Variables of Particle File

Another word for could be , which we prefer, but will not use. We would *like* to say that the “tables are indexed by one or more s,” but the NetCDF terminology is that “ are indexed by one or more s,” and we will adopt that language here.

The main s that we use are time, scenario, and particle-within-scenario. We give an example, using the lat. The variable lat is indexed by all 3 main s. Hence,

Equation 8: Indexing of the Variable lat in NetCDF File

There is a similar variable lng for the longitudes of the particles. Recall the notion of from §II.F.5, means . This is the of the particle, and it changes with time. For a sailboat voyages, it can be either distress or the values listed in Table 4. For a regular voyage, the will always be “underway” or “distress.”

In Table 12, we list the s. In Table 13, we list the basic variables in a .

|  |  |
| --- | --- |
| Name | Description |
| time | For tables that have a time component, e.g. lat. |
| scenario | For tables that have a scenario component. |
| particle | particle-within-scenario. Anything that references individual particles will need to include both scenario and particle in its . |
| svts | State Vector Types. Variables that are indexed by svts are describing the s, and typically are not indexed by scenario or particle |
| sots | Search Object Types. Similar comment as for svts. |

Table 12: Dimensions in a Particle File

One “trick” we use is to store variables whose values are ordinals of sets, rather than values of the sets themselves. For example, consider . We have a corresponding to , and a String variable that is dimensioned by the . This String variable simply gives the names of the different values of . When we want a variable to take on values of type , we use an int, and those int values are the ordinals of the values. We do this for , , and .

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | s | Type | Description |
| time | time | int | List of #Secs Since 01/01/2000, 0000:00Z |
| svts | svt | String | Names of State Vector Types |
| lat | time,scenario,particle | float | Latitudes of particles at time-step |
| lng | time,scenario,particle | float | Longitudes of particles at time-step |
| probability | time,scenario,particle | float | Probability of particle at time-step |
| pfail | time,scenario,particle | float | Cumulative pFail of particle as of time-step |
| scenarioOrdinal | scenario | int | Ordinal of Scenario |
| scenarioWeight | scenario | float | Weight of Scenario |
| anchoringTime | scenario,particle | int | Time (see 1st variable) of particle’s anchoring |
| birthTime | scenario,particle | int | Time of particle’s birth |
| landingTime | scenario,particle | int | Time of particle’s landing |
| expirationTime | scenario,particle | int | Time of particle’s expiration (not used) |
| underwayType | scenario,particle | int | Ordinal of particle’s underway type |
| distressType | scenario,particle | int | Ordinal of particle’s distress type |
| initPrior | scenario,particle | float | Initial probability of particle |
| originalPrior | scenario,particle | float | Initial probability of particle[[5]](#endnote-5) |
| distressLat | scenario,particle | float | Latitude of particle at distress |
| distressLng | scenario,particle | float | Longitude of particle at distress |
| svtOrdinal | time,scenario,particle | int | Ordinal of of particle |

Table 13: Basic Variables or Tables of a Particle File

In addition, there are several variables that store averages for each scenario across the particles of each Search Object Type. Such variables will not have the “particle” , but rather the “sots” . Since they are the averages rather than all of the values, these variables’ sizes are far smaller than, e.g., lat. We list these averages in Table 14.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | s | Type | Description |
| meanBirthTime | scenario,sots | int | For given scenario, Search-Object-Type combination, the average time of the particles’ “birth” |
| meanLandingTime | scenario,sots | int | Similar to above for landing |
| meanLat | time,scenario,sots | float | At each time-step, the average lat |
| meanLng | time,scenario,sots | float | At each time-step, the average lng |

Table 14: Variables Averaging out Particle

In 2.2, there is an attribute writeOcTables that can appear on the top tag either explicitly in the xml or indirectly through Sim.properties. If that is present and set to true, the variables of Table 15 will also be in the . The “Oc” in writeOcTables refers to “Originating Craft” and these list what *would* happen to each particle if it does not go into distress. These all refer to the pre-distress motion of a voyage. Typically, they are used for sailboat voyage; we really have little interest in the wind or currents or boat-speeds for the voyages of §II.D.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | s | Type | Desccription |
| ocLat | time,scenario,particle | float | Latitude of particle’s path, assuming no distress |
| ocLng | time,scenario,particle | float | Longitude of particle’s path, assuming no distress |
| ocEastDnWind | time,scenario,particle | float | Down-wind East component at time/point specified as per above |
| ocNorthDnWind | time,scenario,particle | float | Down-wind North component at time/point specified as per above |
| ocEastDnCurrent | time,scenario,particle | float | Down-current East component at time/point specified as per above |
| ocNorthDnCurrent | time,scenario,particle | float | Down-current North component at time/point specified as per above |
| ocEastBoat | time,scenario,particle | float | East component of boat movement, assuming no distress |
| ocNorthBoat | time,scenario,particle | float | North component of boat movement, assuming no distress |
| ocSvtOrdinal | time,scenario,particle | int | of boat, assuming no distress |

Table 15: OcTables of Particle File

## Strings of Particle Files

In addition, there are three strings and they are stored as global attributes.

The 1st string has the most rigid requirements, and is called model. This is a misnomer; it is simply the content of the xml file. Because of Sim.properties, this does not necessarily represent the model itself, but the specification for the is that the attribute called model will contain the content of the xml file. Note that, in particular, additional standard xml comments within this string are forbidden because there is no guarantee that the other parts of SAROPS that require this string can parse such comments. The requirements on this string are very rigid; just the file content.

The 2nd string is called simEchoModel, and it really *does* represent the model. The only modules of SAROPS that read this string are Planner and SimWebServer. Currently, Sim, Planner, and SimWebServer are all written by the same person and they will almost certainly always be written by one person or a small team. Hence, neither the name of the string nor its value is as rigidly specified.

The 3rd String is called stateVectorTypesString and stores the names of all the s. A 4th string stateVectorTypeColorsString merely gives some suggestions for colors using standard HTML color names. Port sailings are “reddish,” starboard sailings are “greenish,” etc., but these are merely suggested, and are subject to change.

# REFERENCES

1. [CASP 1] Computer Aided Search and Rescue Planning (CASP), Volume 1
2. [CASP 2] Computer Aided Search and Rescue Planning (CASP), Volume 2
3. [NAT SAR 1] National Search and Rescue Manual, Volume I: National Search and Rescue System, Joint Chiefs of Staff, U.S. Coast Guard, Joint Pub 3 50, COMDTINST M16120.5a, 1 February 1991
4. [WANGSNESS], A New Method of Position Estimation using Bearing Measurements, IEEE Transactions on Aerospace and Electronic Systems.
5. [ADD] Addendum to The United States National Search and Rescue Supplement (NSS)  
   The International Aeronautical And Maritime Search And Rescue Manual (IAMAR),  
   U.S. Department Of Homeland Security,  
   U.S. Coast Guard, COMDTINST M16130.2d, 29 April 2004
6. [LRC] Memorandum “SAROPS Lateral Range Curves,” USCG (G-Opr-1), 16131, 11 March 2004
7. [LRC 1] Enclosure (1) To Memorandum “SAROPS Lateral Range Curves,” USCG (G-Opr-1), 16131, 11 March 2004, “Discussion of SAROPS LRC And MSPP Issues”
8. [LRC 2] Enclosure (2) To Memorandum “SAROPS Lateral Range Curves,” USCG (G-Opr-1), 16131, 11 March 2004, Excerpt From “Elements, Modules, And Algorithms For Planning Optimal Searches
9. [SOFT NAVSAR] Software Requirements Specification For The Naval Search And Rescue (NAVSAR), Naval Oceanographic Office, NAVSAR 01-12, December 2001
10. [LEEWAY] Allen, Art, Leeway Divergence, 2005
11. [GSHHS] Wessel, P., and W. H. F. Smith, A Global Self-consistent, Hierarchical, High-resolution Shoreline Database, J. Geophys. Res., 101, #B4, pp. 8741-8743, 1996.
12. [ETOPO} Amante, C. and B.W. Eakins, 2009. ETOPO1 1 Arc-Minute Global Relief Model: Procedures, Data Sources and Analysis. NOAA Technical Memorandum NESDIS NGDC-24. National Geophysical Data Center, NOAA. doi:10.7289/V5C8276M [access date].
13. [ELEM] Frost, J.R., Elements, Modules, and Algorithms for Planning Optimal Searches, prepared for US Coast Guard Office of Search and Rescue, 30 September, 2002
14. [NetCDF] NetCDF Documentation, https://www.unidata.ucar.edu/software/netcdf/docs/index.html

1. On a sphere, this is technically incorrect. A polygon simply divides the sphere into two regions, and there really is nothing different about one from the other. We will adopt the convention that the smaller region is the interior and the larger the exterior. Note that we *need* this convention to talk about whether or not a polygon is clockwise or counterclockwise. A polygon is clockwise or counterclockwise from the perspective of someone standing in its smaller region. [↑](#endnote-ref-1)
2. It is of interest to note that to check for inclusion of a LatLng in a Polygon, we use the same GeoMtx structure mentioned in VI.C. Here we are looking for the first edge we cross when we go straight North. If that edge crosses left-to-right, is in water. Hence, we store the LakeBasins as clockwise (water) polygons. [↑](#endnote-ref-2)
3. Bayesian updating multiplies each particle ’s weight based on some data . It does this by multiplying ’s previous weight by . This quantity is called the *likelihood* for and . In our case, is simply “we did not find the object.” Hence, multiplying weights by *Pfail*s is exactly a Bayesian update. This suggests that other Bayesian updates could be applied as well. [↑](#endnote-ref-3)
4. The Computation of CPA is also in [ELEM] [↑](#endnote-ref-4)
5. Duplicate of initPrior, for backward compatibility purposes. Only one of these will be in each Particle File [↑](#endnote-ref-5)