1818 Library ◆ Suite 600 ◆ Reston, VA 20190-5602 ◆ (703) 326-2861  
<http://www.metsci.com>



Simulator Documentation  
V1.2.0.4

Mar 3, 2010

Thomas M. Kratzke

Sr. Analyst

Contents

[1. Introduction 1](#_Toc244069969)

[1.1 *Sim* vs Planner 1](#_Toc244069970)

[2. Basic Idea of *Sim*: Particles 2](#_Toc244069971)

[3. XML input file and *Sim*.properties 3](#_Toc244069972)

[3.1 XML Input File 3](#_Toc244069973)

[3.2 Sim.properties 3](#_Toc244069974)

[3.3 DOM Parser 3](#_Toc244069975)

[3.4 Example 4](#_Toc244069976)

[4. The class Model 10](#_Toc244069977)

[4.1 ENVDATA 10](#_Toc244069978)

[4.2 Search Object Type and Originating Craft 10](#_Toc244069979)

[5. Area 12](#_Toc244069980)

[6. Scenarios 13](#_Toc244069981)

[7. Making a random draw for the time of distress using Hazards 14](#_Toc244069982)

[8. Random Draws from Areas 15](#_Toc244069983)

[8.1 Making Draws for an Area from two Uniform-0,1 draws 15](#_Toc244069984)

[8.2 Polygonal 15](#_Toc244069985)

[8.3 Elliptical 17](#_Toc244069986)

[9. Pre-Distress Route of Particle 18](#_Toc244069987)

[10. Linking a draw from one Area to the Next 19](#_Toc244069988)

[11. Wind and Current Values 21](#_Toc244069989)

[11.1 Environmental data 21](#_Toc244069990)

[11.2 Interpolation 21](#_Toc244069991)

[11.3 Random Draws and Exponential Decay of Correlation 21](#_Toc244069992)

[11.4 Gibing 21](#_Toc244069993)

[12. Previous Searches 22](#_Toc244069994)

[13. Checking for Landed Particles 23](#_Toc244069995)

[13.1 Algorithm; Using a Grid 23](#_Toc244069996)

[14. GSHHS Files 24](#_Toc244069997)

[14.1 Resolutions: different GSHHS Files 25](#_Toc244069998)

[15. HydroGp files 26](#_Toc244069999)

[16. Utility Programs for G-files 27](#_Toc244070000)

[16.1 checkAndFixCrossings 27](#_Toc244070001)

[16.2 BuildResolutions 27](#_Toc244070002)

[16.3 CreateDFiles 28](#_Toc244070003)

[16.4 GshhsReader 28](#_Toc244070004)

[16.5 Using the D files to avoid reading in huge swaths of land 28](#_Toc244070005)

[16.5.1 Lake Michigan Problem 28](#_Toc244070006)

[17. USGS files (National Atlas) 29](#_Toc244070007)

[17.1 Shape file 29](#_Toc244070008)

[17.2 GeoTools 29](#_Toc244070009)

[17.3 Combining Touching polygons 29](#_Toc244070010)

[17.4 Filtering Choices 29](#_Toc244070011)

[18. Creating the Land Files 30](#_Toc244070012)

[18.1 Steps for Creating Land Files 30](#_Toc244070013)

[18.1.1 Create the High Resolution G-File HydroGp-f.b, from the HydroGp Shape File 30](#_Toc244070014)

[18.1.2 Create Lower Resolution G-Files from Hydro-f.b 30](#_Toc244070015)

[18.1.2.1 Douglas-Peucker and its Limitations 30](#_Toc244070016)

[18.1.2.2 Replacement Algorithm 30](#_Toc244070017)

[18.1.3 Create D Files from Gshhs files and Hydro files 30](#_Toc244070018)

[19. Using the pair of Land Files, given an AOI 31](#_Toc244070019)

[19.1 Select a Resolution 31](#_Toc244070020)

[19.2 Read each separately 31](#_Toc244070021)

[19.3 Combine, using “Water-Carves-Land” 31](#_Toc244070022)

[19.4 Hierarchize; set the Polygons’ Levels 31](#_Toc244070023)

[20. Data flow 32](#_Toc244070024)

[21. Sim Output Files 33](#_Toc244070025)

[21.1 Particle File 33](#_Toc244070026)

[21.1.1 Dimensions and Variables 33](#_Toc244070027)

[21.2 Log File 35](#_Toc244070028)

[21.2.1 POS Values 35](#_Toc244070029)

[21.2.1.1 TIME tag (attributes of the TIME Tag) 35](#_Toc244070030)

[21.3 Statistics File 36](#_Toc244070031)

[22. Acronym List 37](#_Toc244070032)

[23. References 38](#_Toc244070033)

List of Algorithms

[Algorithm 1. Particle prior to distress 13](#_Toc244070034)

List of Code

[Code 1. Case Mississippi Delta – XML File 1 4](#_Toc244070035)

[Code 2. Case BRAVO\_SAR\_PTRN\_LYRRENDERER – XML File 2 5](#_Toc244070036)

[Code 3. – Sim.properties 7](#_Toc244070037)

[Code 4. Output from Reading Mississippi\_delta 8](#_Toc244070038)

List of Figures

[Figure 1. XML specification for the datum NominalSpeed 4](#_Toc244070039)

[Figure 2. Sim.properties specification for NominalSpeed 4](#_Toc244070040)

[Figure 3. Gorgeous picture 1 16](#_Toc244070041)

[Figure 4. Accumulation of particles between two Areas, if Draws are made Randomly 19](#_Toc244070042)

[Figure 5. Making a Correlated DrawDraw 20](#_Toc244070043)

List of Formulas

[Formula 1. Computing the area between *X\** and *X* 16](#_Toc244070044)

[Formula 2. Using Inverse Cdf functions 19](#_Toc244070045)

[Formula 3. Cdfs to A Random Draw 19](#_Toc244070046)

List of Snippets

[Snippet 1. Abstract methods found in Cdf 15](#_Toc244070047)

List of Tables

[Table 1. Values for intensities 14](#_Toc203499697)

Document Revision History

|  |  |
| --- | --- |
| Version Date | Comments |
| Oct 31, 2008 | For *Sim* V1.2.0.4 |
| Mar 3, 2010 | For Sim V1.3.0.0 |
|  |  |
|  |  |
|  |  |

# Introduction

In this article, we present a detailed view of SAROPS *Simulator* (or *Sim)*. Although we will describe the problem that *Sim* works on and the algorithms that it uses, we also point out where in the code the main outlines of these algorithms are implemented. In fact, this article is as much about describing the critical java classes of the code as it is about the mathematics.

We assume that the reader has a background in the Coast Guard’s search planning problem, and that he has some coding experience. The code is in Java and it would be helpful if he could at least follow Java code.

This does not mean that we will discuss every line of the code; such an article would be longer, far more difficult to read, and more prone to errors than the code itself. Rather, we discuss the big ideas of *Sim*, and the main classes that embody those ideas. As such, this article is intended for people who really want to “dig down” and understand the “inner workings of *Sim*.”

## *Sim* vs Planner

*Sim* is responsible for gathering information about where the missing person is. The program *Planner* takes this information, along with information about the available search assets, and computes a plan for finding the missing person. In short, *Sim* is about processing the information that we have about the case, and *Planner* is about making decisions based on the results of this processing.

Planner is discussed in the two articles *Planner Algorithm* and *Introduction to SAROPS Interactive Planner*.

# Basic Idea of *Sim*: Particles

*Sim* cannot pinpoint exactly where a person is located, but it can compute a rough idea of where a person *probably* is. *Sim* uses a mathematical technique called "particle filters,” to compute something very close to a probability distribution of where the missing person is.

*Sim* cannot get a closed form for the probability distribution of the position of the missing person, but it can run thousands of guesses that are based on what we know about the case and the environment. The guesses differ from each other by small random fluctuations, and these are generated within the program by using a random number generator. The guesses are called "particles," and we base a probability distribution on where the guesses end up being clustered.

An example illustrates the basic principle. Suppose we are in a very strange part of the world where the winds and currents stop cold at 9:00 pm, but commence the following day at 4:00 pm. When they are active, the winds are always 10 kts and blowing north, and the current is always 3 kts and flowing east.

Somebody has lost a ping pong ball and we need to know where to look for it. We know that the person lost it in the vicinity of a point *P* and roughly at 4:45 pm. It is now 4:45 on the following day. We could try to replicate the lost ball by dropping thousands of ping pong balls in the vicinity of *P*, and watching where they end up at 9:00, thereby getting some idea of where the lost ping pong ball was.

There are obvious drawbacks to this scheme, the chief one being that the lost ping pong ball would have drifted farther. Furthermore, we don’t want to wait until 9:00 for an answer, but this illustrates the basic idea of a particle filter. Instead of letting the random fluctuations of the environment scatter ping pong balls, we use computer-generated random fluctuations to distribute our planted ping pong balls, but the idea is essentially the same; replicate the situation as closely as possible many times, and see where the replications end up.

Although this example seems silly, (just buy a new ping pong ball, for crying out loud), it illustrates a few points; the more we know about where the object was and where it was going, the easier it is for us to create meaningful replications. Furthermore, the more we know about how the winds and currents affect the object we are looking for, the more effective our replications will be. For example, if the object we were really looking for was an almost totally submerged sponge, it would end up to the north whereas our ping pong balls would end up to the east.

Since we do not know when something was lost, what it is now (possibly we are looking for a life raft, possibly a boat, possibly a person in a life jacket), we generate thousands of these guesses (henceforth, we will use the term “particle”). We’ll vary the winds and currents a little for each particle, and vary winds and currents that each particle experiences, and how these environmental factors affect it.

At the end of our program, we will look at where all of the particles are, and this will be the closest that we come to a probability distribution for our missing object.

# XML input file and *Sim*.properties

We start by discussing the problem’s inputs. It is a trivial, but often overlooked, truism that you simply cannot understand any code or algorithm unless you understand the data of the problem. To explain the input, we start by discussing the mechanics of getting the data into our program—the XML input and the Sim.properties files.

## XML Input File

Our input XML file has a very simple format; there is no validator or XSD. Every tag has only attributes and sub-tags. For example, we never read CDATA or text within the XML. Furthermore, we never use tags as data; only attributes. Every attribute has a name and a value, where the name precedes an equal sign, and the value succeeds the equal sign and is enclosed by double quotes. We read an attribute’s name only to identify which datum is referenced, and we always get the data from the attributes’ values.

Hence, every datum is associated with a nest of XML tags and a single attribute.

Note that *Sim* does not *write* this XML; *Sim* only *reads* it. The program that writes the XML is the main GUI of SAROPS, and we call it *pre-Sim*. Getting ahead of ourselves a bit, we state here that *Sim* writes files for some other program to read. We call *that* program *post-Sim.*

Despite the general description given here, in practice pre-Sim and post-Sim are the same program, and it is the program that the user sees when he runs a SAROPS case.

## Sim.properties

Sim.properties is a companion file. Different SAR cases have different XML files, but there is a constant Sim.properties file. As of 1.2.0.4, *Sim* is capable of using additional Sim.properties files to override the one global file.

The main purpose of Sim.properties is to provide default values for the values in the XML files. In this way, we can push hard-coded constants into Sim.properties and still retain the option of overriding these values within the XML. If the requirements for *Sim* change, and these constants need to be made specific to a particular SAR case, *pre-Sim* can simply add this attribute to its XML-writing algorithm, and our code will need no updating.

When looking for a particular datum, *Sim* looks at the XML. If it’s not there, we look at Sim.properties. As an example, we give a snippet of an XML in that is optional, since the corresponding data is filled in as a default in Sim.Properties, as shown in .

Note that if we had used “OVERRIDE” in the Sim.properties file, instead of “DEFAULT,” the XML would have been ignored; this is almost never done outside of debugging.

## DOM Parser

We use a DOM parser to read the XML, and it is provided by the standard library Xerces. Our XML is little more than a simple tree of data; sub-tags correspond to sub-branches of the tree, and the attributes correspond to the data at a given node of the tree. Hence traversing the XML is very much like traversing a tree, using a depth-first traversal.

Our XML is not big enough to require a SAX parser; we can easily read the entire XML into memory and process it.

This “traversing code” is found in the method com.metsci.sarops.model.io.ModelReader.traverse. Note that the primary argument to traverse is an org.w3c.dom.Element. This Element is simply the top node of the tree. For more information on DOM parsers and Elements, see the documentation on Xerces (Reference needed)

<Sim>

<!--Generation time: 6/23/08 12:46 PM-->

<ENVDATA>

…

</ENVDATA>

<SEARCH\_OBJECT\_TYPE id="1" name="PIW with PFD (Average)">

<LEEWAY NominalSpeed="20 kts">

Figure . XML specification for the datum NominalSpeed

Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.nominalSpeed.DEFAULT = 20 kts

Figure . Sim.properties specification for NominalSpeed

## Example

We are now ready to give examples of an XML/Sim.properties pair. We give two XML files and one Sim.properties to emphasize that the XML files correspond to individual cases, and that Sim.properties is a single global file. As we discuss the model, which is very nearly a java representation of the XML/Sim.properties data, we will refer to these two examples.

Code . Case Mississippi Delta – XML File 1

<Sim>

<ENVDATA>

<CURRENT type="nc" confidence="low"

file="C:/Cases/Mississippi\_Delta/Env/HYCOM\_063921113910\_39672.NC"

beginTime="181100Z DEC 06" endTime="200200Z DEC 06"/>

<WIND type="nc" confidence="low"

file=" C:/Cases/Mississippi\_Delta/Env/NAM\_063821113834\_57477.NC"

directionFrom="no" beginTime="181100Z DEC 06"

endTime="200200Z DEC 06"/>

</ENVDATA>

<SEARCH\_OBJECT\_TYPE id="1"

name="PIW Vertical PFD type III conscious">

<LEEWAY NominalSpeed="20 kts">

<DWL slope="0.0048" Syx="0.161340487 kts"/>

<CWLPOS slope="0.0015" Syx="0.130238706 kts"/>

<CWLNEG slope="-0.0015" Syx="0.130238706 kts"/>

</LEEWAY>

</SEARCH\_OBJECT\_TYPE>

<SEARCH\_OBJECT\_TYPE id="2"

name="Life Raft - No ballast, Canopy, No drogue">

<LEEWAY NominalSpeed="20 kts">

<DWL slope="0.0339" Syx="0.046652671 kts"/>

<CWLPOS slope="0.0149" Syx="0.046652671 kts"/>

<CWLNEG slope="-0.0149" Syx="0.046652671 kts"/>

</LEEWAY>

</SEARCH\_OBJECT\_TYPE>

<ORIGINATING\_CRAFT id="-1" name=""/>

<SCENARIO name="Scenario 1" id="1" weight="30%">

<PATH>

<DEPARTURE\_LOCATION>

<POINT lat="28.90588" lng="-89.2358" x\_error="2 NM"/>

<TIME dtg="181200Z DEC 06" plus\_minus="1 hrs"/>

</DEPARTURE\_LOCATION>

</PATH>

<SCEN\_OBJECT\_TYPE id="1" weight="50%"/>

<SCEN\_OBJECT\_TYPE id="2" weight="50%"/>

</SCENARIO>

<SCENARIO name="Scenario 2" id="1" weight="30%">

<PATH>

<DEPARTURE\_LOCATION>

<POINT lat="28.90588" lng="-89.2358" x\_error="2 NM"/>

<TIME dtg="181200Z DEC 06" plus\_minus="1 hrs"/>

</DEPARTURE\_LOCATION>

<DEAD\_RECKON motion\_type="RL" course="90 T"

courseError="10 deg" cruisingSpeed="15 kts" minSpeed="12 kts"

maxSpeed="17 kts">

<DISTRESS\_TIME dtg="181400Z DEC 06"/>

</DEAD\_RECKON>

</PATH>

<SCEN\_OBJECT\_TYPE id="1" weight="50%"/>

<SCEN\_OBJECT\_TYPE id="2" weight="50%"/>

</SCENARIO>

<REQUEST type="RunSimulation">

<INPUT displayMap="no" datumTime="200100Z DEC 06" mode="fast"

left="-90.75643 degs" right="-87.10354 degs" top="30.30332 degs"

bottom="27.50844 degs"/>

<OUTPUT firstOutputTime="181100Z DEC 06" freq="60 mins"

file="C:/Cases/Mississippi\_Delta/Out/alpha\_mississippi\_delta"/>

<CASE planner="jrf" case\_name="Mississippi Delta"

run\_date="211126Z DEC 06" search\_plan="ALPHA"

run\_datum\_time="191300Z DEC 06"/>

</REQUEST>

</Sim>

Code . Case BRAVO\_SAR\_PTRN\_LYRRENDERER – XML File 2

<Sim>

<ENVDATA>

<CURRENT type="nc" confidence="low"

file="C:/Cases/BRAVO\_SAR\_PTRN\_LYRRENDERER/Env/FNMOC\_GULFSTREAM\_20070205094059.NC"

beginTime="301200Z JAN 07" endTime="020000Z FEB 07"/>

<WIND type="nc" confidence="low"

file="C:/Cases/BRAVO\_SAR\_PTRN\_LYRRENDERER/Env/NAM\_073705093752\_27560.NC"

directionFrom="no" beginTime="301400Z JAN 07"

endTime="011300Z FEB 07"/>

</ENVDATA>

<SEARCH\_OBJECT\_TYPE id="2"

name="Life Raft - No ballast, No canopy, No drogue">

<LEEWAY NominalSpeed="20 kts">

<DWL slope="0.071" Syx="0.201967188 kts"/>

<CWLPOS slope="0.0245" Syx="0.17222611 kts"/>

<CWLNEG slope="-0.0245" Syx="0.17222611 kts"/>

</LEEWAY>

</SEARCH\_OBJECT\_TYPE>

<SEARCH\_OBJECT\_TYPE id="1"

name="Sailing Vessel Mono-hull, FIN Keel - Shallow Draft">

<LEEWAY NominalSpeed="20 kts">

<DWL slope="0.0267" Syx="0.161340487 kts"/>

<CWLPOS slope="0.0298" Syx="0.130238706 kts"/>

<CWLNEG slope="-0.0298" Syx="0.130238706 kts"/>

</LEEWAY>

</SEARCH\_OBJECT\_TYPE>

<ORIGINATING\_CRAFT id="-1" name=""/>

<SCENARIO name="Scenario 1" id="1" weight="100%">

<PATH>

<DEPARTURE\_LOCATION>

<POINT lat="35.28" lng="-75.49167" x\_error="0 NM"/>

<TIME dtg="301400Z JAN 07" plus\_minus="0 hrs"/>

</DEPARTURE\_LOCATION>

<VOYAGE>

<FINAL\_LEG motion\_type="RL" cruisingSpeed="12 kts"

minSpeed="10 kts" maxSpeed="15 kts" LegID="1"

total\_distance="209.233">

<POINT lat="35.57167" lng="-71.22667"

x\_error="0 NM"/>

<TIME dtg="310726Z JAN 07" plus\_minus="0.000 hrs"/>

</FINAL\_LEG>

</VOYAGE>

</PATH>

<SCEN\_OBJECT\_TYPE id="1" weight="50%"/>

<SCEN\_OBJECT\_TYPE id="2" weight="50%"/>

</SCENARIO>

<FIX\_HAZARD id="1" name="Hazard 1" intensity="L" start="" end="">

<POLYGON>

<POINT lat="35.55" lng="-73.385"/>

<POINT lat="35.27" lng="-73.49834"/>

<POINT lat="35.21833" lng="-72.96667"/>

<POINT lat="35.77833" lng="-72.95333"/>

</POLYGON>

</FIX\_HAZARD>

<COMPLETED\_SEARCH id="1"

name="A-1-Untitled Search Pattern-line color comes from pattern Display tab"

start="311400Z JAN 07" end="311653Z JAN 07" endDTG="1170262424"

percent="100">

<PATTERN>

<WAYPOINT lat="35.2853780578302" lng="-72.7568826530654"

dtg="311400Z JAN 07"/>

<WAYPOINT lat="35.6307203380489" lng="-70.6272780204117"

dtg="311523Z JAN 07"/>

<WAYPOINT lat="35.4911219421698" lng="-70.5932173469346"

dtg="311530Z JAN 07"/>

<WAYPOINT lat="35.1457796619511" lng="-72.7228219795883"

dtg="311653Z JAN 07"/>

</PATTERN>

<COMP\_OBJECT\_TYPE id="1" sw="7.17 NM" ap0="2.909793"

a1="-0.826438"/>

<COMP\_OBJECT\_TYPE id="2" sw="2.79 NM" ap0="1.778314"

a1="-1.385097"/>

</COMPLETED\_SEARCH>

<COMPLETED\_SEARCH id="2" name="A-2-Untitled Search Pattern-whazzup"

start="311400Z JAN 07" end="311651Z JAN 07" endDTG="1170262273"

percent="100">

<PATTERN>

<WAYPOINT lat="35.6699757683565" lng="-73.3470798384489"

dtg="311400Z JAN 07"/>

<WAYPOINT lat="35.8648891374227" lng="-72.7684275233008"

dtg="311435Z JAN 07"/>

<WAYPOINT lat="35.7464786132741" lng="-72.7081139582731"

dtg="311445Z JAN 07"/>

<WAYPOINT lat="35.5515652442079" lng="-73.2867662734212"

dtg="311521Z JAN 07"/>

<WAYPOINT lat="35.4331547200593" lng="-73.2264527083935"

dtg="311530Z JAN 07"/>

<WAYPOINT lat="35.6280680891254" lng="-72.6478003932454"

dtg="311606Z JAN 07"/>

<WAYPOINT lat="35.5096575649768" lng="-72.5874868282178"

dtg="311615Z JAN 07"/>

<WAYPOINT lat="35.3147441959106" lng="-73.1661391433659"

dtg="311651Z JAN 07"/>

</PATTERN>

<COMP\_OBJECT\_TYPE id="1" sw="8.39 NM" ap0="2.869304"

a1="-0.697460"/>

<COMP\_OBJECT\_TYPE id="2" sw="3.48 NM" ap0="1.642597"

a1="-1.045671"/>

</COMPLETED\_SEARCH>

<REQUEST type="RunSimulation">

<INPUT displayMap="no" datumTime="011200Z FEB 07" mode="fast"

left="-77.5727 degs" right="-69.13808 degs" top="37.2705 degs"

bottom="33.58117 degs"/>

<OUTPUT firstOutputTime="301400Z JAN 07" freq="60 mins"

file=" C:/Cases/BRAVO\_SAR\_PTRN\_LYRRENDERER/Out/bravo\_sar\_ptrn\_lyrrenderer"/>

<CASE planner="jim" case\_name="SAR\_PTRN\_LYRRENDERER"

run\_date="050932Z FEB 07" search\_plan="BRAVO"

run\_datum\_time="010000Z FEB 07"/>

</REQUEST>

</Sim>

Case bravo\_sar\_ptrn\_lyrrenderer

Code . – Sim.properties

Sim.ENVDATA.WIND.halfLife.DEFAULT = 60 mins

Sim.ENVDATA.WIND.standardDeviationForLOWConfidence.DEFAULT = 5.0 kts

Sim.ENVDATA.WIND.standardDeviationForHIGHConfidence.DEFAULT = 2.0 kts

Sim.ENVDATA.WIND.confidence.DEFAULT = LOW

Sim.ENVDATA.CURRENT.halfLife.DEFAULT = 264 mins

Sim.ENVDATA.CURRENT.standardDeviationForLOWConfidence.DEFAULT = 0.37 kts

Sim.ENVDATA.CURRENT.standardDeviationForHIGHConfidence.DEFAULT = 0.22 kts

Sim.ENVDATA.CURRENT.confidence.DEFAULT = LOW

# The following refer to the way the resolution in Gshhs is chosen.

Sim.lowest\_res\_radius\_in\_nmi.DEFAULT = 240 NM

Sim.highest\_res\_radius\_in\_nmi.DEFAULT = 240 NM

Sim.COMPLETED\_SEARCH.force\_koopman.DEFAULT = false

# The following is for sim, plan and eval.

Sim.COMPLETED\_SEARCH.motion\_type.DEFAULT = GC

Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.gibingRate.DEFAULT = 4% perHr

Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.nominalSpeed.DEFAULT = 20 kts

Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.DWL.useRayleigh.DEFAULT = false

Sim.REQUEST.INPUT.timeStepForFAST.DEFAULT = 20 mins

Sim.REQUEST.INPUT.timeStepForNORMAL.DEFAULT = 20 mins

Sim.REQUEST.INPUT.timeStepForCOMPREHENSIVE.DEFAULT = 20 mins

Sim.REQUEST.INPUT.particlesForFAST.DEFAULT = 2500

Sim.REQUEST.INPUT.particlesForNORMAL.DEFAULT = 5000

Sim.REQUEST.INPUT.particlesForCOMPREHENSIVE.DEFAULT = 10000

Sim.may\_warm\_start.DEFAULT = false

Sim.REQUEST.INPUT.randomSeed.DEFAULT = 979323

Sim.REQUEST.INPUT.distinctDetectionThreshold.DEFAULT = 5 mins

Sim.FIX\_HAZARD.N.DEFAULT = 1.0

Sim.FIX\_HAZARD.L.DEFAULT = 3.0

Sim.FIX\_HAZARD.M.DEFAULT = 10.0

Sim.FIX\_HAZARD.H.DEFAULT = 100.0

# For testing and debugging, values are in standard deviations.

# POC for radius = 1.0 standard deviations is 0.393469 (~39%)

# POC for radius = 2.0 standard deviations is 0.864665 (~86%)

# POC for radius = 3.0 standard deviations is 0.988891 (~99%),

# but in SAROPS, samples > 3.0 standard deviations are disallowed,

# hence POC for radius = 3.0 standard deviations is 1.0 (100%).

Sim.REQUEST.OUTPUT.ContainmentRadius.1.DEFAULT = 1.0

Sim.REQUEST.OUTPUT.ContainmentRadius.2.DEFAULT = 2.0

Sim.REQUEST.OUTPUT.ContainmentRadius.3.DEFAULT = 3.0

# Planner Properties follow:

# Legitimate values for "mode" are ideal and normal.

# "ideal" uses POD = ERF(C\*SQRT(PI)/2).

# "normal" uses POD = 1 - EXP(-C).

PLAN.mode.DEFAULT = normal

PLAN.searchForLandedParticles.DEFAULT = false

PLAN.searchForAdriftParticles.DEFAULT = true

PLAN.plannerMaximalRuntime.DEFAULT = 90 secs

PLAN.may\_warm\_start.DEFAULT = true

PLAN.must\_warm\_start.DEFAULT = false

PLAN.write\_eval\_file.DEFAULT = false

# Minimum Track Spacing defaults.

PLAN.minimum\_track\_spacing\_for\_unknown.DEFAULT = 0.5 NM

PLAN.minimum\_track\_spacing\_for\_small\_boat.DEFAULT = 0.1 NM

PLAN.minimum\_track\_spacing\_for\_helo.DEFAULT = 0.1 NM

PLAN.minimum\_track\_spacing\_for\_fixed\_wing.DEFAULT = 1.0 NM

PLAN.minimum\_track\_spacing\_for\_vessel.DEFAULT = 0.5 NM

Sim.REQUEST.OUTPUT.write\_grid\_file.DEFAULT = false

Sim.gshhs\_buffer\_size\_in\_lat\_degrees.DEFAULT = 0.5 degrees

Code . Output from Reading Mississippi\_delta

DEFAULT : CoreKey = Sim.gshhs\_buffer\_size\_in\_lat\_degrees: 0.5 degrees

DEFAULT : CoreKey = Sim.lowest\_res\_radius\_in\_nmi: 240 NM

DEFAULT : CoreKey = Sim.highest\_res\_radius\_in\_nmi: 240 NM

DEFAULT : CoreKey = Sim.may\_warm\_start: false

XML : CoreKey = Sim.ENVDATA.CURRENT.file: ../Runs/SimRuns/Mississippi\_Delta/Env/HYCOM\_063921113910\_39672.NC

XML : CoreKey = Sim.ENVDATA.CURRENT.type: nc

XML : CoreKey = Sim.ENVDATA.WIND.file: ../Runs/SimRuns/Mississippi\_Delta/Env/NAM\_063821113834\_57477.NC

XML : CoreKey = Sim.ENVDATA.WIND.type: nc

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.id: 1

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.name: PIW Vertical PFD type III conscious

DEFAULT : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.nominalSpeed: 20 kts

DEFAULT : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.gibingRate: 4% perHr

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.DWL.slope: 0.0048

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.DWL.Syx: 0.161340487 kts

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.CWLPOS.slope: 0.0015

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.CWLPOS.Syx: 0.130238706 kts

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.CWLNEG.slope: -0.0015

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.CWLNEG.Syx: 0.130238706 kts

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.id: 2

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.name: Life Raft - No ballast, Canopy, No drogue

DEFAULT : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.nominalSpeed: 20 kts

DEFAULT : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.gibingRate: 4% perHr

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.DWL.slope: 0.0339

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.DWL.Syx: 0.046652671 kts

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.CWLPOS.slope: 0.0149

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.CWLPOS.Syx: 0.046652671 kts

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.CWLNEG.slope: -0.0149

XML : CoreKey = Sim.SEARCH\_OBJECT\_TYPE.LEEWAY.CWLNEG.Syx: 0.046652671 kts

XML : CoreKey = Sim.ORIGINATING\_CRAFT.id: -1

XML : CoreKey = Sim.SCENARIO.id: 1

XML : CoreKey = Sim.SCENARIO.weight: 100%

XML : CoreKey = Sim.SCENARIO.PATH.DEPARTURE\_LOCATION.POINT.lat: 28.90588

XML : CoreKey = Sim.SCENARIO.PATH.DEPARTURE\_LOCATION.POINT.lng: -89.2358

XML : CoreKey = Sim.SCENARIO.PATH.DEPARTURE\_LOCATION.POINT.x\_error: 2 NM

XML : CoreKey = Sim.SCENARIO.PATH.DEPARTURE\_LOCATION.TIME.plus\_minus: 1 hrs

XML : CoreKey = Sim.SCENARIO.PATH.DEPARTURE\_LOCATION.TIME.dtg: 181200Z DEC 06

XML : CoreKey = Sim.SCENARIO.PATH.DEAD\_RECKON.course: 90 T

XML : CoreKey = Sim.SCENARIO.PATH.DEAD\_RECKON.courseError: 10 deg

XML : CoreKey = Sim.SCENARIO.PATH.DEAD\_RECKON.minSpeed: 12 kts

XML : CoreKey = Sim.SCENARIO.PATH.DEAD\_RECKON.cruisingSpeed: 15 kts

XML : CoreKey = Sim.SCENARIO.PATH.DEAD\_RECKON.maxSpeed: 17 kts

XML : CoreKey = Sim.COMPLETED\_SEARCH.motion\_type: RL

XML : CoreKey = Sim.SCENARIO.PATH.DEAD\_RECKON.DISTRESS\_TIME.dtg: 181400Z DEC 06

XML : CoreKey = Sim.SCENARIO.SCEN\_OBJECT\_TYPE.id: 1

XML : CoreKey = Sim.SCENARIO.SCEN\_OBJECT\_TYPE.weight: 50%

XML : CoreKey = Sim.SCENARIO.SCEN\_OBJECT\_TYPE.id: 2

XML : CoreKey = Sim.SCENARIO.SCEN\_OBJECT\_TYPE.weight: 50%

XML : CoreKey = Sim.REQUEST.INPUT.mode: fast

DEFAULT : CoreKey = Sim.REQUEST.INPUT.particlesForFAST: 2500

DEFAULT : CoreKey = Sim.REQUEST.INPUT.timeStepForFAST: 20 mins

XML : CoreKey = Sim.REQUEST.INPUT.datumTime: 200100Z DEC 06

DEFAULT : CoreKey = Sim.REQUEST.INPUT.randomSeed: 979323

XML : CoreKey = Sim.REQUEST.INPUT.left: -90.75643 degs

XML : CoreKey = Sim.REQUEST.INPUT.bottom: 27.50844 degs

XML : CoreKey = Sim.REQUEST.INPUT.right: -87.10354 degs

XML : CoreKey = Sim.REQUEST.INPUT.top: 30.30332 degs

DEFAULT : CoreKey = Sim.REQUEST.INPUT.distinctDetectionThreshold: 5 mins

DEFAULT : CoreKey = Sim.REQUEST.OUTPUT.write\_grid\_file: false

XML : CoreKey = Sim.REQUEST.OUTPUT.firstOutputTime: 181100Z DEC 06

XML : CoreKey = Sim.REQUEST.OUTPUT.file: ../Runs/SimRuns/Mississippi\_Delta/Out/alpha\_mississippi\_delta

DEFAULT : CoreKey = Sim.REQUEST.OUTPUT.gridDivisions: 80

DEFAULT : CoreKey = Sim.REQUEST.OUTPUT.ContainmentRadius.1: 1.0

DEFAULT : CoreKey = Sim.REQUEST.OUTPUT.ContainmentRadius.2: 2.0

DEFAULT : CoreKey = Sim.REQUEST.OUTPUT.ContainmentRadius.3: 3.0

The output (Code xxx) shows what data was read in, the value, and “were it came from.” If it came from the XML, the line starts with “XML.” If it came from a DEFAULT value provided in Sim.properties, the line starts with “DEFAULT.”

# The class Model

We are now ready to discuss where in the code the inputs are stored. We start with the notion of a *model*. We will use the term *model* to indicate the data that describes the situation. *Sim* uses this data to produce the positions of the particles.

*Sim* uses the classes com.metsci.sarops.model.Model and com.metsci.sarops.model.ScenarioSet to hold the *model*. A *model* is the main object that holds the data gathered by the XML and Sim.properties. The two classes can be thought of as one; it was broken into two classes just to keep the class to a reasonable size. A ScenarioSet has nothing more than a backpointer to a Model, and a list of Scenarios. We will discuss Scenarios shortly. We will refer to the pair Model/ScenarioSet by simply referring to “Model.”

We will describe the Model by walking through the XML that is used to populate it. This reading is done with an object of type com.metsci.sarops.model.io.ModelReader. A ModelReader is initialized with a Model. The Model has many “setter” methods, and the ModelReader simply traverses the XML, and calls these setters to populate the Model.

## ENVDATA

The first block, ENVDATA, tells where to get the nc files that are used to represent the environmental data. nc files are a special format of file that we use to represent the environmental data that was gathered in preparation for running *Sim*. The code that reads the nc files is found in com.metsci.sarops.environment.NetcdfReader, and its extensions DriftsNetcdfReader and DriftsNetcdfReader, found in the same package.

The result of reading in a NetCdf file are DriftNetcdfReader and WindNetcdfReader. Both of these classes are implementations of interfaces, and their main responsibility is to be able to answer the critical question “what is the velocity (speed and direction) of the current (wind) at time *t*, and LatLng latLng?” Reading the data provides enough data to answer this question, since we will be interpolating in both time and space (see Section XXX).

## Search Object Type and Originating Craft

The next section of the XML describes the object types that the distress object might be, as well as what he started as. It is assumed that the latter data is known, and it cannot vary from scenario to scenario.

These are implemented within the code by the class com.metsci.sarops.model.Craft. This class has a SearchObjectType, which has a great deal of information, as well as the weight within the scenario of that SearchObjectType. Although the term “craft” suggests that it is a reference to the originating craft, it is really a reference to any object type.

The class com.metsci.sarops.model.SearchObjectType contains a LeewayData (which is a private inner class) that provides the data for how this object type reacts to the wind, and we will discuss this in more detail when we discuss particles.

The search object type is also used to determine the lateral range curve used for this object type, for each completed search. We will discuss this more when we discuss “completed search.”

# Area

The next section of the XML is “Scenario.” We will discuss the scenarios and pre-distress motion shortly, but we need to develop the concept of *area* first.

Roughly speaking an *area* is a two-dimensional connected region of the earth’s surface. We currently have only two types of *areas*; elliptical (which certainly includes circular and indeed is almost always a circle) and polygonal. For either type of *area*, we use a flat earth approximation. We use a local projection and deal only with the coordinates on the plane of projection. For example, a polygon is a polygon on the projection plane. Even so, since we are using a local projection, the polygon is almost indistinguishable from the polygon on the sphere that has for its vertices the given points, and for its edges, great circle arcs that connect the vertices.

The elliptical area is used as, among other things, a bivariate normal distribution. This requires that we use a planar approximation to the earth.

*Area*s are also used to define “Hazardous” regions, which we will discuss later.

All of these concepts are captured in the abstract class com.metsci.util.cdf.area.Area. We have two concrete extensions, BivariateNormal and Polygon, both in the same package. These correspond to Elliptical and Polygonal areas. Reading in *area*s is routed through the same code whether we are discussing hazards or components of the pre-distress motion. This code is found in the method com.metsci.sarops.model.io.ModelReader.readArea.

# Scenarios

The next section of the XML describes the Scenarios. Scenarios are hypotheses concerning pre-distress motion, or “what we knew about the incident before distress occurred.” We use this information to move the particles until our simulated time advances to the point where the particle changes to a distress state.

Pre-distress movements are not subject to environmental data, only where our ideas of where the missing person intended to go, how long he planned to loiter, etc.

Every case has at least one scenario. There are three basic types of scenarios: LKP (or Last Known Position), Voyage (a path connecting a sequence of areas to each other), and Dead Reckon (an LKP plus a speed and direction). We hope to add a “flare” scenario after version 1.2.0.4.

In *Sim* terminology, LKP is an area, either elliptical or polygonal. The SAROPS term “point” means (circular) elliptical area in *Sim*, and the SAROPS’ term “area” means polygonal area in *Sim*. *In this document, we will use the term “area” as Sim does, as opposed to how SAROPS does.*

A *voyage* is a sequence of *area*s (to emphasize the point made in the previous paragraph, recall that an area can be either elliptical or polygonal). This scenario is most often used for a case involving an overdue traveler. Every case has at least one scenario. Most cases have only one, and the LKP scenario is used when all you have is an estimate of the place and time of distress.

In addition to describing the pre-distress motion, a Scenario also states what happens to a particle when it goes into distress. It does this with the tags “SCEN\_OBJECT\_TYPE.” Note that the attributes of these tags tell both what type of object the distress object is and the probability that a particle from this scenario will turn into that type of distress object. So the algorithm for a particle prior to distress is shown in .

1. For each particle, lay out a distress-free route using random draws from the departure area and an Itinerary Generator.
2. Make a random draw for the time of distress.
3. Make a random draw for the type of distress object that it will become.

Algorithm . Particle prior to distress

Before we discuss the routing from one area to the next, we elaborate on how we make a draw for the time of distress. This would be straightforward except for the presence of hazards.

# Making a random draw for the time of distress using Hazards

A hazard is an *area*, either polygonal or elliptical. It represents a region where distress is more likely to occur. The intensity *i* of a hazard is a positive real number that indicates how much more likely the particle is to go into distress when in the hazard.

These intensities are hard-coded constants. As of 1.2.0.4, they are in Sim.properties (and hence could theoretically be given in the XML), but the XML identifies which of a discrete set of intensities we are using. In 1.2.0.4, the standard Sim.properties gives the values for the intensities as in .

N: 1.0

L: 3.0

M: 5.0

H: 100.0

Table . Values for intensities

To explain how *Sim* uses these intensities, it’s best to take an example. Suppose the intensity *i* = 3, the total time of a particle’s path is 100 hours, and 5 of these hours are spent in the hazardous region. Rather than take a draw for a time between 0 and 100 for the distress time, *Sim* takes a draw for a number between 0 and 97(1) + 5(3) = 112.

The “97(1)” occurs in the sum because for 97 hours, the particle is in a region of intensity 1. Similarly, the “5(3)” occurs because for 5 hours, the particle is in a region of intensity 3. It is straightforward to map the draw between 0 and 112 to a time of distress so that the particle is 3 times as likely (per unit time) to go into distress in the hazard as it is outside of the hazard. The code that *Sim* uses is found in the method com.metsci.sarops.model.VisitsItinerary.findDistressTimeInSeconds.

*Sim* currently uses fixed (in position) hazards that have a time interval of being active. If a particle goes through a hazard when it is inactive, the hazard has no effect.

We could implement moving hazards. For example, a storm that is moving would be well modeled by a moving hazard. The problem would be to describe the motion of the hazard. Our current model is better for modeling shipping lanes and stationary storms.

# Random Draws from Areas

We still have to discuss the three types of pre-distress motion model, as well as how the particles move after distress. Before we discuss the three types of scenarios in some detail, we need to discuss how we make random draws that correspond to *area*.

Central to creating paths for the pre-distress motion of the particles is the idea of making a random draw from an *area*. Using the abstract class com.metsci.util.cdf.Cdf and its extensions BivariateNormalCdf and PolygonCdf from the same package, we can convert two Uniform-0,1 draws to a draw for a position from an *area*. Furthermore, given a draw for a position from its *area*, the Area object can compute the cdf (cumulative distribution function) for its x-coordinate, and the cdf of its y-coordinate, conditioned on the x-coordinate. Hence, the Area object can map either two Uniform-0,1’s to a point, or a point to two Uniform-0,1’s. This code is promised in the abstract methods found in Cdf; see .

public final double[] cdfsToXy(double cdfX, double cdfYGivenX, double[] result);

public final double[] xyToCdfs(double x, double y, double[] result);

Snippet . Abstract methods found in Cdf

Implementations for these two methods are of course found in BivariateNormalCdf and PolygonCdf.

It will come as no surprise that a BivariateNormal Area is used to construct a BivariateNormalCdf, and a Polygon Area is used to construct a PolygonCdf. These Cdf objects are then used to select random points from a given *are*a.

## Making Draws for an Area from two Uniform-0,1 draws

This section describes how the two functions in Code AAA are implemented for each of the two cases. We will use the terminology “from Cdfs” to indicate the code that takes two Uniform-0,1’s and creates a point, and the terminology “to Cdfs” to indicate the inverse. This is equivalent to computing the Cdfs and their inverse functions.

## Polygonal

This is by far the more complicated case when going either to Cdfs or from Cdfs. The code is found in the class PolygonCdf. We start by giving a sketch of how to find the cdf for a given value x.

In the main constructor for PolygonCdf, we sort the endpoints of the polygon by increasing x-coordinate. For each endpoint’s x-coordinate *x*, we store the edges of the polygon that “straddle” it. An edge *straddles* *x* if its left-hand endpoint is no greater than x and its right-hand endpoint is strictly greater than x. In particular, vertical edges straddle no point.

Now consider an endpoint *p*, and its x-coordinate *x\**. Using the slopes and x-coordinates of the straddling edges at *x\**, as well as the area to the left of the previous endpoint, we can compute the area of the polygon that is to the left of *x\**, as well as the bands of *y*-values that are in the polygon and above *x\**. We omit the details here except to say that this is the justification of the code that involves cumY and cumSlope in the constructor.

We now focus on finding the marginal (for *x*) cdf of a point *(x,y)*. Since we are concerned with a marginal, the “*y*” here is irrelevant. Suppose that *p = (x,y)* is a point, and *x* is not the x-coordinate of any point in the polygon. We will eventually have the area of the polygon at our disposal and so finding the marginal cdf is equivalent to finding the area of the polygon that is to the left of *x*.

We find the x-coordinate *x\** of the endpoint that is closest to *x*, but no greater than *x*. By looking at the slopes of the edges that straddle *x\**, (which are exactly the edges that straddle *x*), and other data at *x\**, we can compute the area between *x\** and *x* as in .

Baffling hogwash formula

Formula . Computing the area between *X\** and *X*

Dividing by the area of the polygon gives us the cdf of *x*.

To find the cdf for *y*, given *x*, we note that there is a collection of bands above *x*, and *y* must be a member of one of these bands; see .

Gorgeous picture

Figure . Gorgeous picture 1

These bands at *x* can be computed by again considering *x\**’s data and projecting forward. Given these bands at *x*, and a given *y*, the conditional cdf for *y*, given *x*, is simple to compute.

Note that for long narrow areas, such as a 2-dimensional approximation of a small island’s coastline, simply drawing a pair of points in the bounding box and discarding it if it is not in the area would be very time-consuming. Moreover, it does not fit within the framework that we have set up; namely mapping two Uniform-0,1’s to a point and vice-versa. We will need that framework to make correlated draws in the sequence of Areas of a voyage, and we need this correlation to avoid “hourglassing.”

Thus far, we have discussed only the “toCdfs” computation. As stated before, this is simply finding the cdf values for a given point. The “fromCdfs” computation, which is what is necessary to go from a pair of Uniform-0,1’s to a point in the Area, is essentially the inverse, and we omit its discussion entirely.

The code for a PolygonCdf is useful in other parts; as a side-effect, it finds the center of mass and the area of a polygon. Finding the area of a polygon is useful in Planner when computing overlap.

## Elliptical

This is a much simpler concept, since marginals and conditionals of bivariate normals are normal, and formulas for the approximation of both the normal Cdf and the normal inverse Cdf, are well known. This code is found in BivariateNormalCdf.

# Pre-Distress Route of Particle

Finally, we can discuss the pre-distress motion. Recall that there are three types of pre-distress motion for the particles, and we will discuss them shortly. Regardless of the type, the basic mechanism for laying out the route is the same, and we describe that here.

This paragraph is largely a review of how pre-distress motion works. First enough random draws are taken (and the number of random draws depends on the type of pre-distress motion) to lay out an unrestricted path for the particle. Second, we draw a time for the distress to occur. For right now, we will say that this time is drawn by using a random number between 0 and the total (hazard-adjusted) time taken by the particle. Third, we check to see if the route takes the particle into land.

In earlier versions of Sim, whenever a particle runs into land, whether it is in distress or not, it stops. The term that we use for this approach is “sticky shore.” Version 1.2.0.4 has an option available called “slippery shore.” If *slippery shore is being used,* a post-distress particle “bounces off” the shoreline and continues to drift.

Looking ahead just slightly, we state that the environment affects a particle’s motion only after the distress time is reached within the simulation.

In *this* section, it remains to describe how we use random numbers to lay down a pre-distress route for a particle, for each of the three types of motion models. For all three, the framework is the same. There is a Departure Location and time distribution. For LKP, that’s all there is. For Dead-Reckon, there is a section that describes the direction and navigation type of the hypothesis, and for voyage, there is a sequence of Areas that *Sim* will route the particle through.

Code for reading these data in is found in ModelReader. Not surprisingly, the methods of interest are readDepartureLocation, readDeadReckon, and readVoyage. The last one simply calls readArea over and over, and accumulate a sequence of Areas. But the code must do slightly more than that. It must read in the speed distribution between the areas, as well as the navigation used (rhumbline or great circle) between the areas.

We now discuss the code for moving a particle according to the pre-distress motion. For LKP (indicated by no DEAD\_RECKON tag and no VOYAGE tag in the scenario), there is nothing to do; the particle is immediately in distress.

We built an abstract class com.metsci.sarops.model.Voyage for the other two. DeadReckon, in the same package, extends this class as does the class “Visits.” Both of these have to provide implementations for several methods, but in particular, the method getItineraryGenerator. An ItineraryGenerator is, as the name suggests, something that can take a random number generator and “generate” an “itinerary.” An Itinerary is a static inner class of Voyage. The key characteristic of an Itinerary is that it can identify the time of distress and can create an “UnderwayStateVector” from an existing UnderwayStateVector. Here, a StateVector is simply the information (Lat/Lng, Object Type, etc.) about the particle that is of interest.

# Linking a draw from one Area to the Next

We devote this section to the random draws of a voyage because we used completely new mathematics to eliminate hourglassing. To understand the issue, consider .

Gorgeous Picture

Figure . Accumulation of particles between two Areas, if Draws are made Randomly

If we simply take a random draw from *area* *A* and another independent random draw from *area B*, then the path has a 50% chance of going from a positive x-coordinate to a negative x‑coordinate, or vice versa. When this happens, mass accumulates near the y-axis near y = 0, and this is undesirable. If we make many such draws and graph the resulting paths, we will get a picture has many paths near the origin, and we call this effect “*hourglassing*.”

Carefully reading the last paragraph, the user will conclude that to avoid *hourglassing*, we cannot use an independent draw from the second area, and so we do not. The first thought is to “correlate” the draws and this is a comforting thought until the reader realizes that he does not know what correlated 2-dimensional random draws means. Hence we invent such a notion here.

Suppose we make a draw *(x,y)* from *area A* and transform it into its cdf values as in .

Formula . Using Inverse Cdf functions

We could take the two values *c*1 and *c*2 and use them as Uniform-0,1s to make a draw in Area B as in .

Formula . Cdfs to A Random Draw

This method has several interesting properties. First, it is a perfectly valid way of getting a draw for *area B* in the sense that such draws will produce a collection of points that is distributed as per *B*’s distribution. Second, almost anybody’s definition of correlated 2-dimensional random variables will say that (x,y) and (z,w) would be highly correlated. Too highly correlated, as a matter of fact.

We tone down the correlation by introducing a constant called *stovepiping*. Think of the tall 19th century “stovepipe” hats. If we have “high stovepiping,” our paths look like such hats-very straight and parallel. High *hourglassing* would mean that the paths always cross. Normal *hourglassing* is somewhere in between.

In the code we control how much *hourglassing* we have with a hard-coded constant. Paying homage to the notion of correlation, we named this constant rho in the method generateLatLngFromCdfs of the class Area. If we were to set rho to -1, we would have perfect hourglassing. Set to 1, we would have perfect *stovepiping*, and if rho is set to 0, we have the standard *hourglassing*, which is too high. In the code, we set rho to 0.8 and we briefly describe here how we use it.

The key method is newCdf, still in the class Area. Given a cdf value, we can invert it with respect to a Gaussian (i.e, find InvCdf for a Normal(0,1)), to obtain a Gaussian random variable *Z*. If we multiply *Z* by a correlation coefficient , and add in an independent Gaussian *Z’* that has been scaled by , we will get another Gaussian *Z”* and the correlation coefficient of *Z* and *Z”* will indeed be . This can all be verified by considering the definitions of correlation, and the fact that the sum of Gaussians is Gaussian.

We can then take the Cdf of *Z”* and plug it into the CdfInverse for our Area *B* to get a draw for the x‑coordinate for Area *B*. Similar calculations are undertaken to get the *Y* value, using the conditional for *Y* given *X*.

Note that this all works fine, except that we needed to rotate distributions so that the line connecting the centers of mass of the two *area*s, was the (new) y-axis.

This is all summed up in .

Figure . Making a Correlated DrawDraw

# Wind and Current Values

## Environmental data

## Interpolation

## Random Draws and Exponential Decay of Correlation

## Gibing

# Previous Searches

# Checking for Landed Particles

## Algorithm; Using a Grid

We use the concept of a virtual grid whenever we are searching for crossings between collections of edges. Theoretically, the problem of finding all crossings between one set of edges (e.g., particle path segments) and another set of edges (land contours) requires one to look at every possible intersection. If there are *n* edges in the first set and *m* in the second set, this would be *nm* computations. With several thousand edges in each set, this would be prohibitive.

Instead, we set up a grid. Given an Area of Interest *AOI*, we compute a local projection and do all of our computations on that. Hence, we will assume that we are working on a finite section of a plane; just the part of the plane that involves the AOI. Moreover, each edge will no longer be an arc on a sphere, but a simple pair of (x,y) points on the plane.

Scaling and shifting as necessary, we assume that these points range from -0 to 80 in both directions. There are 6400 cells in this grid, where the positive integers from 0 to 80 form the bounding coordinates for the cells.

Each edge (actually, a pointer to the edge) will be placed in each cell that it intersects. With land data, there are very few cells that have very many edges. To see if an edge *e* intersects any other edge, we look at all of the cells that *e* is in, and take the union of all of the edges across these cells. These are the only edges that we need to check.

Skipping ahead just slightly, we will state that we have an efficient way of collecting the land edges that are of interest to us. For example, if the SAR case is off the coast of Florida, our method for collecting edges gets only polygons that are close to Florida and truncates those to a very small number of edges. With this small, fairly compact collection of edges, we can use the method outlined in this section to determine when each particle’s path crosses a land barrier. This code is found in com.metsci.sarops.environment.GridBasedLandMassReader.firstLanding.

# GSHHS Files

Our main source of land data is the GSHHS files [Wessel and Smith, 1996]). GSHHS stands for “Global Self-consistent Hierarchical High-resolution Shorelines.” It is a very concise collection of the world's shorelines stored as *complete polygons*, which is very handy.

GSHHS also forms the basis of the coastline support in the Generic Mapping Tools (GMT), which is used by over 10,000 users worldwide (gmt.soest.hawaii.edu; Wessel and Smith [1998]). It is supported by the National Science Foundation, and is released under the GNU General Public License. (See <http://www.gnu.org/copyleft/gpl.html>).

The GSHHS files’ format is so simple, flexible and easy to work with, that we converted data from our other source of land data, to this format. Doing this allowed us to re-use the code com.metsci.util.gshhs.GshhsReader. We describe this format here.

We define a “G-file” as a file of polygons that is in the format of the GSHHS files. This format is useful because it allows us to skip vast stretches of data so reading the entire world’s worth of data while concentrating on a small portion of the world, is quite easy. The key here is that we can select a few points from a polygon, skip some (saving time), read a few more relevant points, skip some more, etc. The G-files make it relatively easy to do that.

We simply use the binary reader within Java, the java.io.DataInputStream. We can skip entire polygons and, when a polygon cannot be skipped, but we are reading a part of the polygon that is far from *AOI*, we can skip several Lng/Lat pairs by using the DataInputStream’s method skipBytes. This code is all found in the class com.metsci.util.gshhs.GshhsReader.

The method that is used is simple; we read in the header information (See Table xxx), which includes the number of Lng/Lat pairs, and

1. int id; Unique polygon id number, starting at 0

2. int n; Number of points in this polygon (these polygons repeat the first and last point, so the minimum number that can appear here is 4)

3. int flag; subflag + (version << 8) + (greenwich << 16) + (source << 24)

a. subflag = (fCodeIndicator << 4) + level, where:

level: 1 land, 2 lake, 3 island\_in\_lake, 4 pond\_in\_island\_in\_lake

level: 6: water body from Hydrogp020,

level: 7: island inside of Hydrogp020 body

version: Set to 6 for GSHHS version 1.11

greenwich: 1 if Greenwich is crossed <br>

source: 0 = CIA WDBII, 1 = WVS, 3 = Hydrogp020

4. int west; (in microdegrees) <br>

5. int east; (in microdegrees) <br>

6. int south; (in microdegrees) <br>

7. int north; (in microdegrees) <br>

8. int area; (in 1/10 km^2 <br>

Note: Iff \_level is odd, the polygon is traced counterclockwise.

Table xxx

## Resolutions: different GSHHS Files

GSHHS comes in different resolution files. Each files represents the entire world, but at different resolutions. The “full” resolution is the data as it was gathered by Wessel and Smith. The remaining resolutions are what are created by either gshhs\_dp or BuildResolutions.

A resolution measure in km represents the maximum amount from the full resolution, that a polygon’s border is allowed to stray when points are removed, and replaced by chords. The following are the resolution measures that we have been using:

Crude (c), Resolution 0: 25 km

Low (l) Resolution 1: (5 km)

Intermediate (i) Resolution 2: (1 km)

High (h) Resolution 3: (5 km)

Table XXX

GSHHS Files as Java Resources: Bookmark class

# HydroGp files

In addition to the GSHHS data, we use another source of data. This data gives us much better inland water data than GSHHS. GSHHS, although it has some lakes and reservoirs, is primarily intended for the world’s shorelines, so we supplement the GSHHS data with another source of data. We use this data to process lakes, streams, canals, and reservoirs for the United States and its territories.

The data that we are using is from the United States Geological Survey (USGS). Data and information authored or produced by the USGS are in the public domain. **National Atlas of the United States®** is a registered trademark of the US Department of the Interior…

# Utility Programs for G-files

The land files do not come completely ready for our use. We do some pre-processing of these files before we include them in the SAROPS distribution. This section describes the suite of programs that we use for this.

These “programs” are really just classes that can serve as the entry point for a java application (i.e, they have a public static void main(String[] args) method). We refer to a “program” by referring to its main class.

## checkAndFixCrossings

The GSHHS files do not necessarily come with no crossings, so the first step is to “clean them up.” GSHHS polygons are supposed to have no crossings but version 1.11, downloaded on July 3, 2008, did have some. There are two types of crossings that we call “intra-loop” and “inter-loop.” Intra-loop crossings are crossings within a polygon, and inter-loop crossings are crossings between two polygons. Fortunately, we have not seen inter-loop crossings, and the intra-loop crossings have been very minor from a geographical point of view. An intra-loop crossing turns a polygon into a “Figure-8” and the smaller of the two loops in the “8” have always been extremely small.

Therefore, simply eliminating the small loop of the “Figure 8” is a reasonable way of fixing the polygon (See Figure XXX)

Gorgeous Figure

Figure XXX

(Sidebar)The author has communicated the crossings to the provider Paul Wessel, of GSHHS, and he is working to clean up the crossings that have been found. Therefore, future versions of GSHHS data might have no crossings, but there’s no guarantee for this, and there’s no guarantee that the author has found all of the crossings.(/Sidebar) Still, the program CheckAndFixCrossings will be necessary to verify that there are no crossings.

The second step is to create the d files for any new GSHHS file that you wish to use. For this, you use the program CreateDFiles.

The program CheckAndFixCrossings does this. It takes a GSHHS file and creates a “re-written” G-file with two changes: 1. The intra-loop crossings are eliminated, 2. If one polygon intersects another, the smaller one is eliminated entirely. As mentioned, the latter situation never occurs in GSHHS version 1.11. One can either fix the crossings of the full resolution file and create the lower resolution files, or fix the crossings of all 5 resolutions. As of GSHHS 1.11, there were crossings in both the full and high resolutions, and no crossings in the intermediate, low, or crude resolutions.

## BuildResolutions

The second step is to create (or at least check) the different resolutions for the GSHHS files. There are at least two ways of doing this. The first is to use the program Gshhs\_dp from GSHHS. This uses the Douglas-Peucker algorithm (reference needed) to eliminate vertices that do not stray far from the chord that connects their two immediate neighbors. The problem with the Douglas-Peucker as it is provided, is that it does not guarantee that no crossings are created. It is theoretically possible that both intra-loop and inter-loop crossings could be introduced. Moreover, it is not only theoretically possible for the HydroGp files, but with the Mississippi river, an almost certainty.

Hence, we wrote another way of creating resolutions, and it is called BuildResolutions. This is much slower, but before it replaces a pair of edges with a chord, it checks if such a replacement would introduce any crossings.

## CreateDFiles

GSHHS files come with a “.b” suffix. We use the program

## GshhsReader

Using the AOI; Truncated Polygons

## Using the D files to avoid reading in huge swaths of land

### Lake Michigan Problem

# USGS files (National Atlas)

## Shape file

## GeoTools

## Combining Touching polygons

## Filtering Choices

# Creating the Land Files

There is a suite of programs that are useful for preparing the land files. These “programs” are really just classes that can serve as the entry point for a java application (i.e, they have a public static  void main(String[] args) method). We refer to a “program” by referring to its main class.

## Steps for Creating Land Files

There are basically three steps to creating the Land files. The first is to clean up the polygons from the input shape file, and create the highest resolution HydroGp file. The second is to create the other resolutions of the HydroGp files, using a Line simplification algorithm. The third is to create the D files for both the GSHHS files and the HydroGp files.

### Create the High Resolution G-File HydroGp-f.b, from the HydroGp Shape File

### Create Lower Resolution G-Files from Hydro-f.b

#### Douglas-Peucker and its Limitations

#### Replacement Algorithm

### Create D Files from Gshhs files and Hydro files

# Using the pair of Land Files, given an AOI

## Select a Resolution

## Read each separately

## Combine, using “Water-Carves-Land”

## Hierarchize; set the Polygons’ Levels

# Data flow

# Sim Output Files

## Particle File

The main output file that Sim provides is a NetCdf file called the Particle File. A NetCdf file can be thought of as a collection of multi-dimensional arrays, and here we describe these arrays.

Before diving into the description of this file, the reader should be aware that we cannot give a byte-by-byte description of it. We can only introduce and discuss the concepts that the programmer uses when writing and reading such a file. Hence, we start by introducing these concepts.

### Dimensions and Variables

Another word for the term “Dimension” is “Index Set.” The arrays that we will store will be indexed by one or more sets, and NetCdf calls these sets *Dimensions*. The dimensions that are of interest to us are “Scenario, Time, and Particle-index-within-scenario.” We will abbreviate “Particle-index-within-scenario” as simply “Particle” in this section.

We carefully explain the dimension “time” here. This dimension indicates the times that we will be storing information for. We create this dimension (or any dimension) by specifying the dimension’s name and the number of elements that are in this dimension, in other words, the size of the index set. The first variable that we index with this dimension is the set of times that are of interest to SAROPS.

A “Variable” is simply an array that is indexed by one or more Dimensions. We have variables as in Table XXX

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Variable | Dimensions | Type | Description |  |
| Time | Time | int | Set of times for this Particle File |  |
| Particle | Particle | int | Index within scenario |  |
| Scenario | Scenario | int | Index within this case |  |
| ScenarioName | Scenario | short | input id of scenario |  |
| ScenarioWeight | Scenario | float | input weight of scenario |  |
| DistressTime | Scenario,Particle | int | time at which this particle went into distress |  |
| DistressType | Scenario,Particle | int | id of what the particle became at distress |  |
| UnderwayType |  | Scenario,Particle | int | id of what the particle started as |
| LandingTime | Scenario,Particle | int | when it landed |  |
| Lat | Scenario,Particle,Time | float | where the particle was at the given time |  |
| Lng | Scenario,Particle,Time | float | where the particle was at the given time |  |
| Probability | Scenario,Particle,Time | float | The proportion of weight that this particle was at this given time |  |
| PFail | Scenario,Particle,Time | float | The cumulative pFail for this particle was at this given time |  |

In addition, there are attributes (strings) that are not related to any dimensions. These are essentially comments, but they can be read and we use them to store the model. In this way, we record the model that was used to create the Sim run.

The attributes that we store are, because we are working within NetCdf, associated with a variable name (which we invent), an attribute name (optional, but which we use as additional information), and a value (which we use the same way that we use the attribute).

For xmlmodel and xmlmodel2, we store the same information; we have two strings for backward compatibility reasons.

|  |  |  |
| --- | --- | --- |
| Variable | Attribute | Value |
| Time | units | seconds since 01/01/00 |
| Xmlmodel | -- | The string containing the xml representation of the model |
| xmlmodel2 | -- | The string containing the xml representation of the model |

## Log File

The log file is produced by the Simulator. This file contains the values of different forms of POS, as well as the “out-of-area” incidents.

### POS Values

A recurring theme is that we often are reporting probabilities as if there had been no search, and also reporting probabilities that take into account the search. If we update the weights of the particles by taking into account the search, we call this “*probability with search*,” and if do not update the weights, we call this “*probability without search*.”

#### TIME tag (attributes of the TIME Tag)

For each time step, there is a “TIME” tag. Within the tag itself (and not the sub-tags), there are the following entries:

|  |  |
| --- | --- |
| distressParticlesWithSearch | *Probability with search* (at this time) of being in distress |
| landedParticlesNoSearch | *Probability without search* of being landed (and in distress) |
| landedParticlesWithSearch | *Probability with search* of being landed (and in distress) |

##### SCENARIO Tags

Within the “TIME” tag, there is a “SCENARIO” subtag for each scenario. A subtag of the “SCENARIO” subtag gives geographic information about the scenario. In the “SCENARIO” tag itself, there are the following entries:

|  |  |
| --- | --- |
| distressCount | Number of Particles in the scenario that are in distress |
| distressParticlesNoSearch | *Probability without search* of being in this scenario, given that it is in distress |
| distressParticlesWithSearch | *Probability with search* of being in this scenario, given that it is in distress |
| jointDistressParticlesNoSearch | *Probability without search* of being in this scenario and in distress |
| jointDistressParticlesWithSearch | *Probability with search* of being in this scenario and in distress |
| jointLandedParticlesNoSearch | *Probability without search* of being landed (and in distress) and in this scenario |
| jointLandedParticlesWithSearch | *Probability with search* of being landed (and in distress) and in this scenario |
| landedCount | Number of particles in this scenario that are landed (and in distress) |
| Conditionals (conditioned on scenario) | |
| landedParticlesNoSearch | *Probability without search* of being in this scenario, given that it is landed |
| landedParticlesWithSearch | *Probability with search* of being in this scenario, given that it is landed |

##### POS Tags

Also within the “TIME” tag, there is a “POS” subtag for each search object type. A subtag of the “POS” subtag gives geographic information about the particles that are (at this time) of that search object type. In the “POS” tag itself, there are the following entries:

|  |  |
| --- | --- |
| conditionalPOS | Probability of finding the object, given that it is this search object type |
| initialProbability | *Probability without search* of being this search object type |
| jointPOS | Probability of finding the object *and* it is this search object type. |
| numberInDistressAndLanded | Number of particles of this search object type that are in distress and landed |
| numberInDistressAndNotLanded | Number of particles of this search object type that are in distress and not landed |
| remainingProbability | Probability without search of being this search object type, minus the probability of success, given that the target is this search object type. |
| remainingTargetProbability | Same as remainingProbability. The two attributes are here for backward compatibility reasons. |

## Statistics File

# Acronym List

[to be added]

# References

Wessel, P., and W. H. F. Smith (1996), A global, self-consistent, hierarchical, high-resolution shoreline database, J. Geophys. Res., 101(B4), 8741–8743.

Wessel, P., and W. H. F. Smith (1998), New, improved version of Generic Mapping Tools released, Eos Trans., AGU, 79(47), 579.