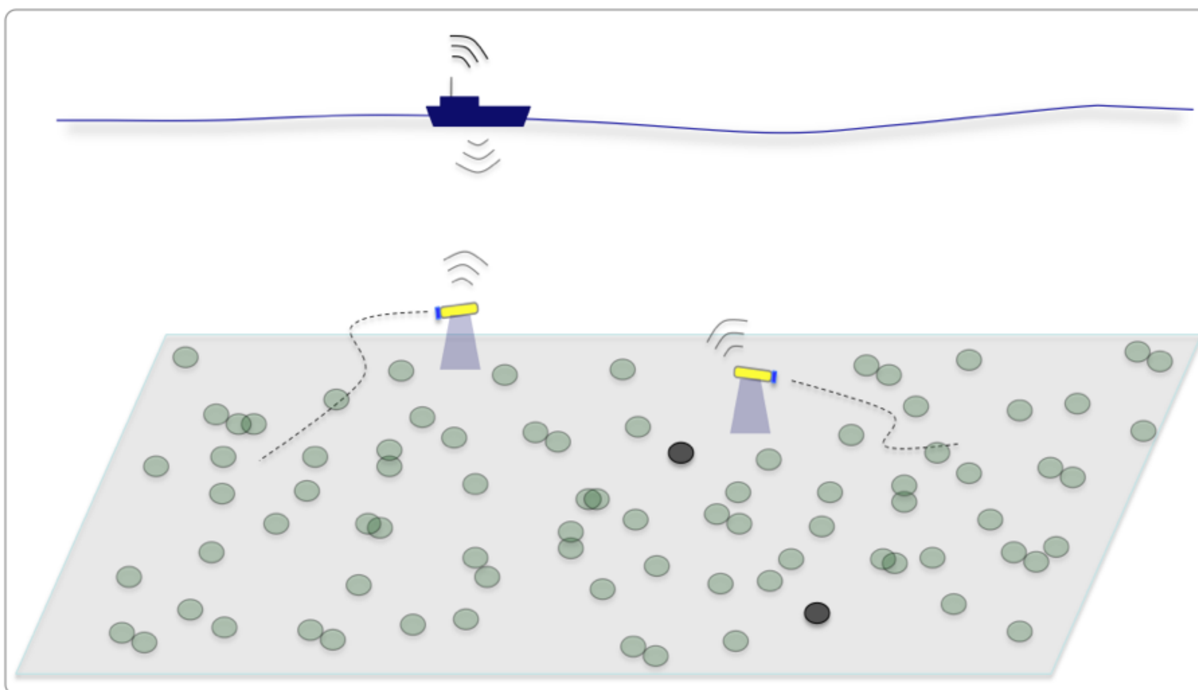
# Autonomy Challenge
# Autonomous Collaborative Hazard Search

MOOS-DAWG'13

**Revision July 26, 2013**

Michael Benjamin, Henrik Schmidt
Department of Mechanical Engineering
MIT, Cambridge MA 02139
mikerb@mit.edu, henrik@mit.edu



# Contents


**1 Overview and Objectives** 4
- 1.1 Background . . . 4
- 1.2 Givens . . . 4
- 1.3 The Challenge and Rules of Competition . . . 4

**2 Problem Description** 6
- 2.1 The Search Region . . . 6
- 2.2 The Object Laydown . . . 7
- 2.3 Metrics of Success . . . 8
- 2.4 Mission and Vehicle Constraints . . . 8
- 2.5 Dynamic Changes to the Rules . . . 9

**3 Getting Started** 11
- 3.1 Get the MOOS-IvP Software . . . 11
- 3.2 Run the Baseline Mission . . . 12
- 3.3 Get the MOOS-IvP Extension Template . . . 12
- 3.4 More Resources . . . 13

**4 The Jake Example Mission** 14
- 4.1 Module Topology in the Jake Mission . . . 15
- 4.2 Key MOOS Variables in the Jake Mission . . . 15
- 4.3 General Description of Events in the Jake Mission . . . 16
- 4.4 Required MOOS Variable Bridges . . . 19
- 4.5 Hazard Files in the Jake Mission . . . 20

**A Hazard File Format** 22

**B Hazard Resemblance Factors** 23

**C Generating Your Own Hazard Files** 25

**D Hazard File Renderings** 26

**E Serialization of Hazard and HazardSet Information** 28
- E.1 Serializing and De-serializing XYHazard Objects . . . 28
- E.2 Serializing and De-serializing XYHazardSet Objects . . . 29

# 1 Overview and Objectives

The hazard search challenge problem is derived from a two-week lab assignment in the MIT class *MIT 2.680, Marine Autonomy, Sensing and Communications*. It is being posted in a slightly modified form as part of MOOS-DAWG'13 to anyone seeking an interesting setting to dive a bit deeper into marine autonomy and the problem of collaborative search with imperfect sensors.

The hazard search challenge problem focusses on the use of collaborative and adaptive autonomy to maneuver a pair of marine vehicles to search a given area for hazardous objects. Each vehicle is equipped with a hazard sensor capable of detecting hazardous objects and further discerning the hazardous objects from benign objects, e.g. rocks. The goal is to search the given region within the given time constraints, and produce report as accurate as possible. The score is based on minimizing penalties for missed hazards and penalties for false alarms. The search strategy, sensor management and inter-vehicle communications strategy is up to the participants.

**How to Participate** A key feature of the competition structure is that a full working *baseline* mission package is distributed with the initial announcement. This baseline mission consists of a few software modules and configuration files and is a syntactically valid submission if simply re-submitted in its original form. The improvements provided by participants may vary in complexity to hopefully allow a wide set of participants to joinl

## 1.1 Background

A new component of MOOS-DAWG'13 is the introduction of an autonomy competition. We plan to make this event a recurring part of these workshops, altering the nature of the competition each year. Entries are purely software in nature, with initial competitions occurring solely in simulation. Top finishers may have their solutions run on MIT-provided marine vehicle hardware on the Charles River at MOOS-DAWG'13. All necessary simulation software and documentation are provided to allow as many participants from different backgrounds to participate. No prior experience with MOOS or MOOS-IvP required The software should work on any Mac or GNU/Linux platform. There is limited support for Windows users.

## 1.2 Givens

- A given search region and set of objects of undisclosed type and location.
- A pair of vehicles with a limited set of sensor settings, set to your choosing.
- A reward structure, in the form of penalties for missed detections, false alarms and mission lateness.

## 1.3 The Challenge and Rules of Competition

- Each entry (team or individual) is allowed two unmanned marine vehicles to search the given region. A maximum time (declared just before mission launch) is allowed for search.
- Each vehicle has a top speed of 2.0 meters per /second.
- Each vehicle is equipped with a hazard sensor which may be configured with one of four allowable swath widths. The larger the swath width, the poorer the sensor performance.

The user may configure the detection threshold for their sensor to increase the probability of detection, but at the expense of increasing the probability of false alarm. The sensor is deactivated during vehicle turns. The swath width may not be changed after mission launch, but the detection threshold may be changed as often as desired.

- Limited communications is allowed between the two vehicles. (Bandwidth, frequency and range limited).
- The two vehicles must submit a single final report comprised of a list of hazardous objects and locations.

### 1.3.1 Getting Started

The starting provisions consist of (a) a baseline mission and (b) documentation including formal competition rules, general software documentation and tutorials for newcomers.

### 1.3.2 The Baseline Mission

A baseline mission will be distributed to participants, consisting of a small MOOS-IvP tree with example MOOS application modules and MOOS mission files for solving the hazard search problem. This tree is in the form of a complete working, syntactically valid competition submission. The search and reporting algorithms however are naive and the vehicles do not coordinate at all. It will not fare well in competitions in its initial form. The purpose of this baseline mission is to reduce the threshold for getting started and to help clarify the format required for competition entries.

### 1.3.3 Documentation / Tutorials

Training labs will be posted, modeled after MIT 2.680 to, to provide a graduated sequence of exercises for new users of MOOS-IvP. These labs are presently available at `http://oceanai.mit.edu/2.680/labs`. General documentation for MOOS and MOOS-IvP is also available at www.moos-ivp.org/docs.

# 2 Problem Description

This section describes the problem parameters, mission constraints and format for a vehicle to generate a report at the end of a mission. The following five points are discussed:

- A search region
- An object lay-down
- Metrics of success
- Mission and vehicle constraints
- Reporting criteria

Values for the first four components are discussed below, but may be altered at competition time. The idea is to have a canonical problem to focus on for development, which will be part of the competition, and have a to-be-determined variation of the problem also part of the competition. The latter will help judge how general the individual submitted solution may be.

## 2.1 The Search Region

The simulated hazard region will be within a polygon with the following four vertices:

- (-150, -75)
- (-150, -400)
- (400, -400)
- (400, -75)

The initial search region is chosen to be rectilinear to simplify a lawnmower pattern generation for a waypoint behavior. See the documentation for this behavior. A pattern may be given to the behavior simply by specifying the characteristics of the lawnmower pattern without having to explicitly determine all the points.

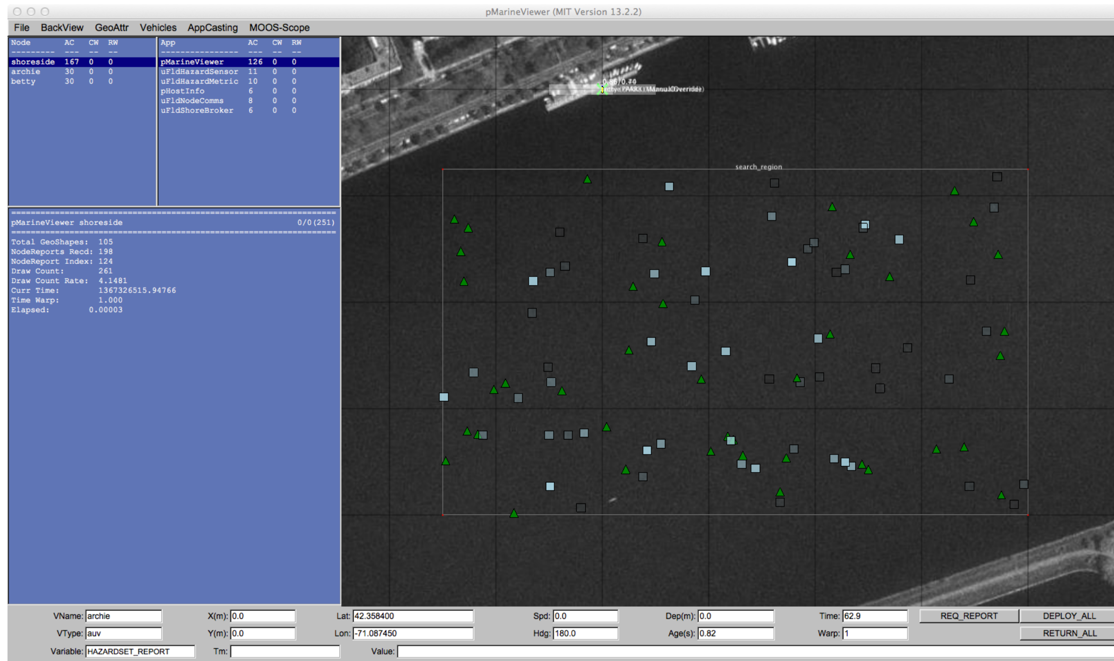



Figure 1: The search region is given in the local coordinates of the MIT Sailing Pavilion on the Charles River. The local (0,0) position is on the dock where vehicles are launched and recovered.

For this competion, the search region will not change during the competition.

## 2.2 The Object Laydown

The object laydown is defined in a *hazard file*, not known to the vehicle at run time. It contains a set of objects, some of which are of type hazard and some of type benign. Identifying and correctly determining the type of these objects is the goal of the mission. For simulation testing and development purposes, a number of hazard files are provided in the baseline mission. A simple example, used as the default in the baseline mission, is shown in Figure 1.

### 2.2.1 Example File Format

A hazard file is simply a text file, with line for each object. An example is below. This is discussed in greater detail in the `uFldHazardSensor` section of [1].

```
hazard = x=165,y=-91,label=32,type=hazard
hazard = x=21,y=-88,label=85,type=hazard
hazard = x=370,y=-219,label=23,type=hazard
hazard = x=-149,y=-318,label=92,type=hazard
o o o
hazard = x=189,y=-326,label=75,type=benign
hazard = x=52,y=-164,label=8,type=benign
hazard = x=174,y=-190,label=46,type=benign
hazard = x=357,y=-177,label=40,type=benign
hazard = x=211,y=-315,label=38,type=benign
```

### 2.2.2 Generating Hazard Files

A hazard file may be generated with a provided command line tool, to allow the user to perform more extensive tests of their choosing. The gen_hazards utility is distributed with the moos-ivp code, and an example command line invocation is below. This tool is discussed in greater detail in the uFldHazardSensor section of [1].

```
$ gen_hazards --polygon=-150,-75:-150,-400:400,-400:400,-75 --objects=5,hazard --objects=8,benign
```

## 2.3 Metrics of Success

The scoring metrics are as follows:

- 150: Penalty for missed hazard
- 25: Penalty for false alarm
- 300: Penalty for being late (later than 7200 seconds)
- 0.25: Additionaly late penalty applied to each second of lateness.

## 2.4 Mission and Vehicle Constraints

There are handful of mission and vehicle constraints enforce to keep everyone on the same playing field.

### 2.4.1 Time

The maximum mission time is 7200 seconds, or 2.0 hours. This value was chosen, in part, since most computers can support a MOOS time warp of 10-20x realtime or much higher on newer systems. A time warp of 20 allows a full mission to be simulated in less than ten minutes. The mission time constraint has a variable penalty associated with it:

- 300: Penalty for being late (1 second later than 7200 seconds), and
- 0.25: Additional late penalty applied to each second of lateness.

### 2.4.2 Top Speed

Vehicles will be limited to a speed of 2.0 meters per second. There is no penalty in terms of sensor performance or comms for moving *at* the top speed. Vehicles moving over the top speed will essentially automatically disable their own sensors until the speed is reduced.

### 2.4.3 Limits on Inter-Vehicle Communication

Communications between vehicles will limited in range, frequency and bandwidth. This will need to be accounted for both during the mission for collaboration, and at the end of the mission for fusing the two vehicles' belief states into a single hazardset report. The restrictions will be

- **Range:** Communications limited to 150 meters

- **Frequency:** Communications will be limited to once per 60 seconds
- **Bandwidth:** Each message will be limited to 100 characters

These configurations are enforced in the uFldNodeComms configuration. This module is on the shoreside and will be configured by the competition organizers, and is configured with these values in the baseline mission. You will be running your own shoreside MOOS community during development, but you should not change this configuration in your uFldNodeComms configuration to ensure you are developing in an environment similar to the competition. Remember that a message that fails to be sent, due to one of the above criteria, will not be queued for later re-sending. You should handle the possibility that message may be dropped.

#### 2.4.4 Sensing During Turns

The simulated sensor will be rendered off during turns of 1.5 degrees per second. This is discussed in greater detail in the uFldHazardSensor section of [1].

#### 2.4.5 Sensor Configuration Options and Configuration Frequency

Vehicles will be allowed to set their sensor configuration once at the outset. Attempts to switch configurations after the outset will simply be ignored. You may, however, switch the the $P_D$ setting (and corresponding $P_{FA}$ setting) as often as you like.

Your sensor configuration options will be:

```
sensor_config = width=5, exp=8, class=0.88, max=1
sensor_config = width=10, exp=6, class=0.70
sensor_config = width=25, exp=4, class=0.63
sensor_config = width=50, exp=2, class=0.55
```

See the documentation for uFldHazardSensor on what this entails in terms of the ROC curve performance of the sensor, and how the vehicle may request a particular configuration. The first sensor setting, with width=5 may be enabled for at most one vehicle.

### 2.5 Dynamic Changes to the Rules

As mentioned at the outset, baseline values are provided for the initial development and first phase of the competition for the following parameters:

- A search region
- An object lay-down
- Metrics of success
- Mission and vehicle constraints

Part of the competition will test exactly on the given baseline parameters. Part of the competition will test on parameters only known at competition time. How will those parameters be made available? They will be set on the shoreside in the uFldHazardMetric application and posted to the shoreside MOOSDB and shared out to the vehicles.

```
UHZ_MISSION_PARAMS = "penalty_missed_hazard=150,penalty_false_alarm=25,
                      penalty_max_time_over=300,penalty_max_time_rate=0.5,
                      search_region=pts={-150,-75:-150,-400:400,-400:400,-75}"
```

If you run the baseline mission and scope the MOOSDB on one of the vehicles, you will notice that a publication similar to the above has been made, originating in the shoreside in the uFldHazardMetric application. For example, after launching the baseline mission, in a separate terminal window you should see something like:

```
$ cd moos-ivp/ivp/missions/m10_jake
$ uXMS targ_betty.moos UHZ_MISSION_PARAMS
Mission File was provided: targ_betty.moos
uXMS_630 thread spawned
starting uXMS_630 thread
----------------------------------------------------
|       This is an Asynchronous MOOS Client        |
|       c. P. Newman U. Oxford 2001-2012           |
----------------------------------------------------

o o o

===================================================================
uXMS_630 betty                                              0/0(1)
===================================================================
VarName             (S)ource          (T)  (C)ommunity  VarValue (SCOPING:EVENTS)
------------------  ----------------  ---  ----------  -------------------------
UHZ_MISSION_PARAMS  uFldHaz..dMetric       shoreside   "penalty_missed_hazard=150,
penalty_false_alarm=25,penalty_max_time_over=300,penalty_max_time_rate=0.5,
search_region=pts={-150,-75:-150,-400:400,-400:400,-75}"
```

Presently, in the baseline mission, this information is not being handled by either vehicle. If the mission were to change, the vehicle would not adjust. But this is why it is a baseline mission; it's not complete and is meant to be a competition starting point.

*In this competition, the only parameters that will change are the penalty values, and the threat lay-down. The search area, top vehicle speed, sensor properties, and communications constraints, will not be changed. You can assume the values used in the baseline mission.*

# 3 Getting Started

The goals of this section are to (a) download and build the moos-ivp software, (b) confirm that the hazard search baseline mission runs on your machine, (c) download and build the moos-ivp-extend software tree, (d) make a trivial modification to the moos-ivp-extend tree to add a hazard search mission.

These steps should take less than a half hour, and by the end you will have a syntactically valid submission for the hazard search competition. From here it is a matter of improving your starting capability.

## 3.1 Get the MOOS-IvP Software

The main moos-ivp software is available from www.moos-ivp.org, following the download link. This tree contains core MOOS, the IvP helm and many other MOOS applications needed for this project. It also includes several example missions including the baseline hazard search mission. There are README files in the top level of the moos-ivp directory to help with identifying platform specific dependencies. Once this is completed, and your shell path has been augmented to include

The steps below are also described on the `www.moos-ivp.org` website, but here is what you need to get started:

### 3.1.1 Step 1: Get the software

To download the software you will need Subversion installed on your system. This free software is available on OS X, GNU/Linux and Windows. Once you have this installed you can download the latest release of MOOS-IvP, which as of this writing is MOOS-IvP 13.5. From the command line this is:

```
$ svn co https://oceanai.mit.edu/svn/moos-ivp-aro/releases/moos-ivp-13.5 moos-ivp
```

Then you should be ready to build.

### 3.1.2 Step 2: Build the software

Before building, read the `README-YOUR_OS` file in the top level of the moos-ivp directory. Ensure that you have any build dependencies installed. After this, the build should be a two step process:

```
$ cd moos-ivp
$ ./build-moos.sh
$ ./build-ivp.sh
```

All executables are built to the `moos-ivp/bin` directory.

### 3.1.3 Step 3: Augment your shell path

Depending on what shell you use, e.g., bash or tcsh, you will need to augment your shell path to include the full path of `moos-ivp/bin`.

### 3.1.4 Step 4: Confirm it works

When all is built properly, you should be able to run the basic mission, the Alpha, and the baseline mission for this project, the Jake mission. For starters, you may want to confirm that the moos-ivp executables are findable in your shell path:

```
$ which pHelmIvP             (or pMarineViewer, MOOSDB, etc.)
  ~/moos-ivp/bin/pHelmIvP
```

Next you may wish to try the alpha mission:

```
$ cd moos-ivp/ivp/missions/s1_alpha
$ ./launch.sh 10
```

This should launch the pMarineViewer GUI and you can launch the mission by hitting the DEPLOY button.

## 3.2 Run the Baseline Mission

A baseline mission is distributed with the moos-ivp tree. It may be run by:

```
$ cd moos-ivp/ivp/missions/m10_jake
$ ./launch.sh 10
```

This baseline mission is described in much greater detail in Section 4. For now just confirm that it launches and vehicles move when the DEPLOY button is hit. The resulting situation should look something like that in Figure 2.

## 3.3 Get the MOOS-IvP Extension Template

The moos-ivp tree is available via anonymous read-only access. Users do not need to register or have a password. However, this tree is read-only to all but the developers. Third party developers (you, if you are participating in this challenge problem), build upon the moos-ivp software and missions by developing a sister stand-alone tree which you may have under your own version control system at some point. A template for this tree is available from www.moos-ivp.org with an example MOOS application and Helm behavior and example build system. There are also a few example missions. This tree may be obtained by SVN as follows:

```
$ svn co https://oceanai.mit.edu/svn/moos-ivp-extend/trunk moos-ivp-extend
```

This tree, if submitted in the form it was downloaded, would constitute a working submission. Submissions consist both of newly created MOOS apps by participants, and a mission file that executes the hazard search mission. Technically a participant could write zero new MOOS apps and perform zero modifications to the mission file and submit this. But the baseline mission is designed to be flawed in some obvious ways. While syntactically valid, it will now perform well. At the very least you will likely want to write your own version of uFldHazardMgr to handle sensor data, and

create an ability for the vehicles to do something else besides perform the hard-coded lawnmower patterns. You can put your modified MOOS app in this tree, and your modified mission in this tree, as part of the competition submission.

### 3.4 More Resources

The below resources are also available for further reference.

- The Collaborative Search lecture notes.
  http://oceanai.mit.edu/2.680/docs/2.680-09-collab_search.pdf
- Further notes on Collaborative Search.
  http://oceanai.mit.edu/2.680/docs/2.680-14-writing-behaviors.pdf
- The MOOS-IvP Autonomy Tools Users Manual
  http://oceanai.mit.edu/moos-ivp-pdf/moosivp-tools.pdf

  This contains the documentation for the uFldHazardSensor, uFldMessageHandler, uFldHazardMetric, uFldHazardMgr, and uFldNodeComms applications.
- The IvP Helm documentation.
  http://oceanai.mit.edu/moos-ivp-pdf/moosivp-helm.pdf
- The moos-ivp.org website documentation.
  http://www.moos-ivp.org

# 4 The Jake Example Mission

The *Jake* mission is distributed with the MOOS-IvP code and was designed to highlight four things:

- The uFldHazardSensor application,
- The uFldHazardMetric application,
- The uFldHazardMgr application, and
- How each of the above work together, in a straw-man solution to the hazard search challenge problem.

The example mission contains a default hazard field and a few other hazard fields for testing. Assuming the reader has downloaded the source code available at www.moos-ivp.org and built the code according to the discussion in Section 3.1, the *Jake* mission may be launched by:

```
$ cd moos-ivp/ivp/missions/m10_jake/
$ ./launch.sh 12
```

The argument, 12, in the line above will launch the simulation in 12x real time. Once this launches, the pMarineViewer GUI application should launch and the mission may be initiated by hitting the DEPLOY_ALL button. Shortly thereafter, two vehicles named archie and betty will begin simple lawnmower patterns over half of the search region each, as shown in Figure 2 below.

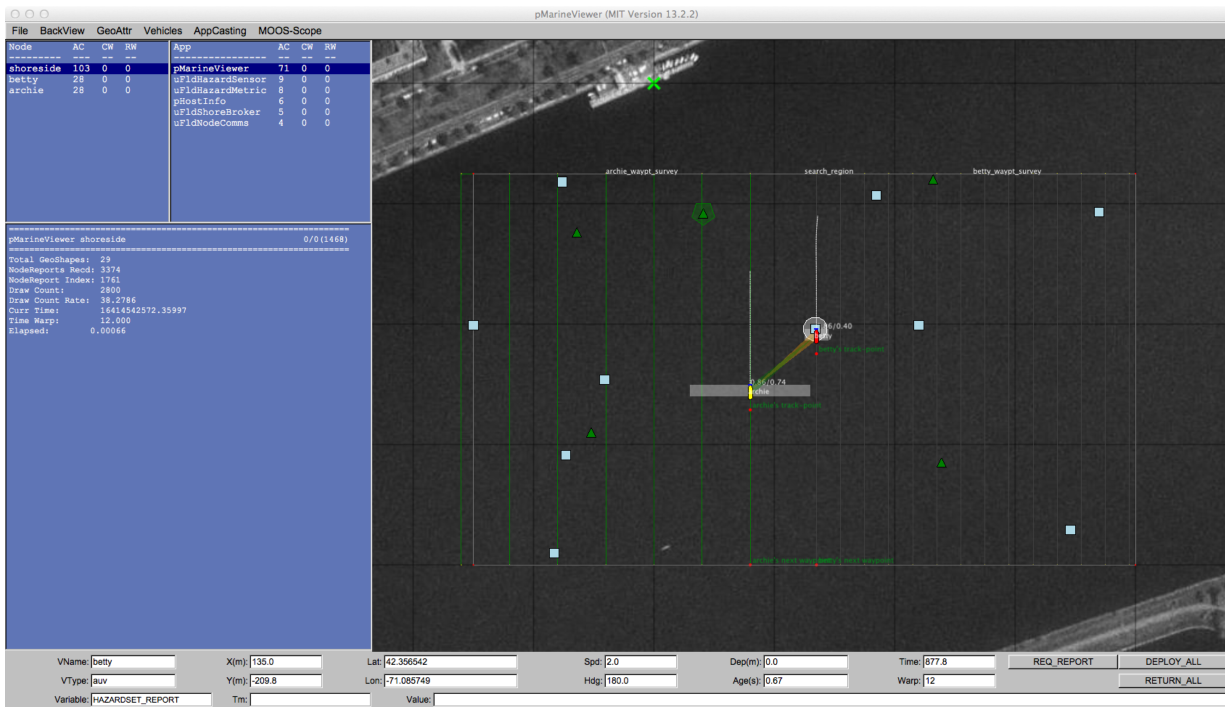



Figure 2: **The Jake example mission**: Two vehicles are on independent lawn-mower shaped search patterns, each with a chosen swath width. Each vehicle will generate a hazardset report upon completion of their initial search. This is a straw-man solution to the hazard-search problem. Several changes or additions need to be made to improve the performance, but this mission constitutes a syntactically valid hazard-search mission.

Any time after the vehicle has been deployed, the user may request the generation of a hazardset report. The REQ_REPORT button sends a HAZARDSET_REQUEST message to the vehicles, each running uFldHazardMgr. Repeated requests result in updated reports. The overall score of the reports tends higher as the mission progresses and more hazards are detected.

## 4.1 Module Topology in the Jake Mission

The key components of the Jake mission are shown in Figure 3 below. The uFldHazardSensor and uFldHazardMetric apps run on the shoreside, the uFldHazardMetric runs on the vehicle. The latter is a straw-man implementation of a application to be implemented by individual developers in the hazard search competition.

All three applications are described in detail in their own separate sections (Sections ??, ??, and ??). A high-level description of operation is that the vehicles (a) configure the hazard sensor, (b) process the sensor information as they move through the field, and finally (c) generate a hazardset report and send it back to the shoreside for evaluation.

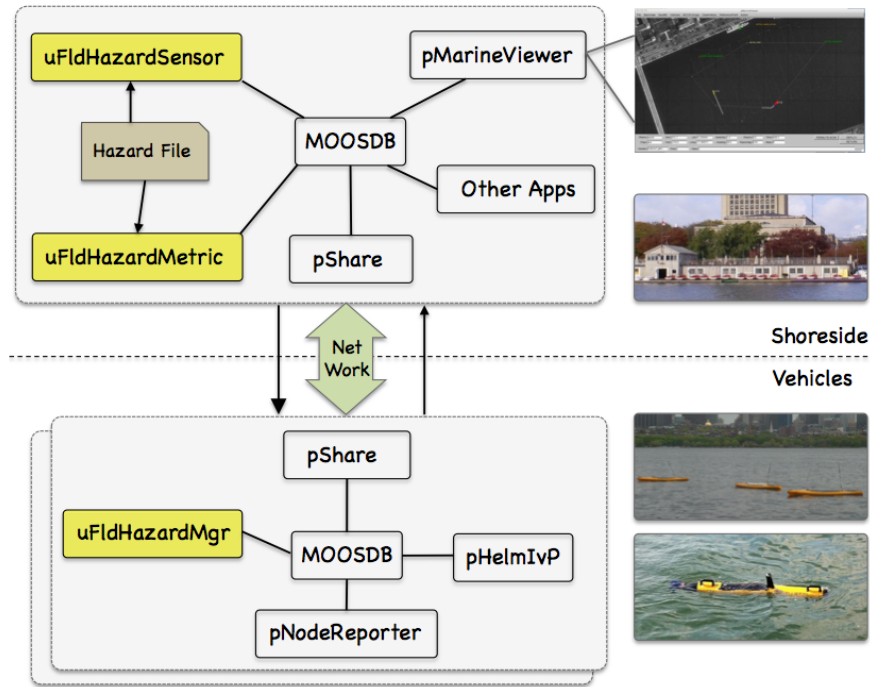



Figure 3: **The Jake Example Mission Module Topology:** The three key modules of the Jake mission are uFldHazarSensor, uFldHazardMetric, and uFldHazardMgr.

## 4.2 Key MOOS Variables in the Jake Mission

To digest what is going on in this mission, familiarity with the key MOOS variables published by each key modules is needed. We list these here for reference in the following discussion operation. It's worth remembering that this information is always available on the command-line, by typing for example:

```
$ uFldHazardSensor --interface or -i
```

The above also provides example values for the key variables, whereas below just the variable names are listed.

#### uFldHazardSensor Interface

- Publications: UHZ_DETECTION_REPORT, UHZ_HAZARD_REPORT, UHZ_CONFIG_ACK, UHZ_OPTIONS_SUMMARY, VIEW_CIRCLE, VIEW_MARKER, VIEW_POLYGON.
- Subscriptions: NODE_REPORT, UHZ_SENSOR_REQUEST, UHZ_CONFIG_REQUEST,UHZ_CLASSIFY_REQUEST.

#### uFldHazardMetric Interface

- Publications: UHZ_MISSION_PARAMS, HAZARDSET_EVAL.
- Subscriptions: HAZARD_SEARCH_START, HAZARDSET_REPORT.

#### uFldHazardMgr Interface

This information, and example values for each variable, may be obtained from the command-line by typing `uFldHazardMgr -i`.

- Publications: UHZ_CONFIG_REQUEST, UHZ_SENSOR_REQUEST, HAZARDSET_REPORT.
- Subscriptions: HAZARDSET_REQUEST, UHZ_CONFIG_ACK, UHZ_DETECTION_REPORT, UHZ_OPTIONS_SUMMARY.

## 4.3 General Description of Events in the Jake Mission

There are roughly three phases to the Jake example mission, (a) initialization and startup, (b) search, and (c) reporting results. These are described below with some reference to what additional measures would be needed beyond the straw-man mission.

### 4.3.1 Phase 1: Sensor Configuration and Handling of Mission Parameters

For each vehicle, there are two steps initialization phase, (a) initializing the hazard sensor, and (b) digesting the mission parameters.

#### Sensor Configuration

The hazard sensor initialization is done in three steps. First uFldHazardSensor publishes the sensor options, with something similar to:

```
UHZ_OPTIONS_SUMMARY = width=10,exp=6,class=0.91:width=25,exp=4,class=0.85:
                      width=50,exp=2,class=0.78
```

Once the vehicle knows the sensor options, it can pick one. In this case the uFldHazardMgr, running on vehicle `archie` posts something similar to:

```
UHZ_CONFIG_REQUEST = vname=archie,width=25,pd=0.9
```

Once the request has been received, uFldHazardSensor, will either accept the requested swath width outright, or map it to the closest legal option. After setting the value for $P_D$, it will determine the $P_{FA}$ and $P_C$, and post a confirmation to the following variable:

```
UHZ_CONFIG_ACK = vname=archie,width=20,pd=0.9,pfa=0.53,pclass=0.91
```

Once this is done, the sensor is configured. In the Jake mission, being a straw-man mission, some of the above steps are over-simplified. In the Jake mission the vehicle does not handle the options summary and just blindly requests a particular sensor setting. It also does not handle the configuration acknowledgement. This may be something improved by a user extending this mission.

**Handling of Mission Parameters**

Upon startup, the mission parameters are published by uFldHazardMetric on the shoreside and sent to each of the vehicles. The format of this message may look something like:

```
UHZ_MISSION_PARAMS = penalty_missed_hazard=100,
             penalty_false_alarm=35,
             penalty_max_time_over=200,
             penalty_max_time_rate=0.45,
             search_region = pts={-150,-75:-150,-50:40,-50:40,-75}
```

The straw-man Jake mission however does not consume and react to these mission parameters. The search mission pattern is hard-coded and the penalties are disregarded, all detections are reported as hazards. This will almost certainly need to be handled by users wishing to improve on the baseline performance provided in the Jake mission.

### 4.3.2 Phase 2: Executing the Search Phase

The activities during the search phase consist of two types of messages originating in the vehicle, and two types of messages originating in the hazard sensor back to the vehicle. Basic sensor operation is turned on when the vehicle publishes the variable:

```
UHZ_SENSOR_REQUEST = vname=archie
```

In the Jake mission, the above message originates in uFldHazardMgr. This is received by the uFldHazardSensor on the shoreside. The hazard sensor regards the sensor to be on if, for the named vehicle, it has received a sensor request recently. If the sensor is on and the vehicle passes over an object and produces a detection, then a detection report is posted by the sensor and sent back to the vehicle:

```
UHZ_DETECTION_REPORT = x=51,y=11.3,label=12
```

The vehicle may decide to request a classification report for a given detection and publish:

```
UHZ_CLASSIFY_REQUEST = vname=archie,label=12,priority=80
```

However, in the Jake mission, no classify requests are generated. The vehicle simply interprets all detections to be hazards and generates a report without using the classify capability of the hazard sensor. This is another area where the user wishing to improve on the baseline mission would focus their effort. Classify requests are eventually answered by the `uFldHazardSensor` publishing:

```
UHZ_HAZARD_REPORT = x=51,y=11.3,type=hazard,label=12
```

#### 4.3.3 Phase 3: Reporting Results

The final phase of the mission involves sending a report from the vehicle to the shoreside, handled by `uFldHazardMetric`:

```
HAZARDSET_REPORT = source=archie#
                   x=-151,y=-217.3,label=01#
                   x=-178.8,y=-234,label=15#
                   x=-59.8,y=-294.1,label=13
```

When is this report sent? In the Jake mission, this report is sent simply after the vehicles have completed their lawnmower pattern. Furthermore, in the Jake mission, each vehicle sends their own report; there is no coordination or communication. The `uFldHazardMetric` evaluates a *single* latest report, so the earlier report in the Jake mission is simply dropped. This is another glaring issue for improvement.

After receiving and evaluating a report, the `uFldHazardMgr` publishes the evaluation to the variable, consisting partly of results, and partly reiterating the criteria for evaluation:

```
HAZARDSET_EVAL  = vname=archie,
           report_name=archie_team,
           total_score=675,
           norm_score=37.5,
           score_missed_hazards=500,
           score_false_alarms=175,
           score_time_overage=0,
           total_objects=10,
           total_time=1284.91,
           received_time=1314.05,
           start_time=29.14,
           missed_hazards=5,
           correct_hazards=5,
           false_alarms=5,
           penalty_false_alarm=35,
           penalty_missed_hazard=100,
           penalty_max_time_over=100,
           penalty_max_time_rate=0.05,
           max_time=1800
```

In the Jake mission, the uFldHazardMgr also will publish a hazardset report upon request when it receives incoming mail HAZARDSET_REQUEST=true. In the Jake mission, the pMarineViewer GUI is configured to generate this request with one of the custom action buttons.

## 4.4 Required MOOS Variable Bridges

The Jake mission requires MOOS variables to be be shared between the vehicle and shoreside community depicted in Figure 3. The variables of key processes were described above in Sections 4.2 and 4.3. The sharing (also referred to as bridging) between MOOS communities is done with the pShare MOOS app distributed with MOOS. Configuration involves specifying which variables are to be sent to which other machine (IP address) along with the port where the other MOOSDB resides. This is configured on the shoreside for variables shared *from* the shoreside *to* the vehicles. A similar configuration exists on the vehicles for variables going in the other direction.

### 4.4.1 Variable Bridges from the Shoreside to the Vehicle

Many of the variables shared from the shoreside to the vehicle were discussed in the previous secions, 4.2 and 4.3. The variable share configuration is handled on the shoreside with the uFldShoreBroker application. Below is the configuration used in the Jake mission:

```
ProcessConfig = uFldShoreBroker
{
  AppTick       = 1
  CommsTick     = 1

  // Note: [qbridge = FOO]  is shorthand for
  //       [bridge = src=FOO_$V,  alias=FOO] and
  //       [bridge = src=FOO_ALL, alias=FOO]

  qbridge  = DEPLOY, RETURN, NODE_REPORT, NODE_MESSAGE
  qbridge  = MOOS_MANUAL_OVERIDE

  bridge   = src=APPCAST_REQ
  bridge   = src=UHZ_MISSION_PARAMS
  bridge   = src=UHZ_OPTIONS_SUMMARY

  bridge   = src=UHZ_CONFIG_ACK_$V,       alias=UHZ_CONFIG_ACK
  bridge   = src=UHZ_HAZARD_REPORT_$V,    alias=UHZ_HAZARD_REPORT
  bridge   = src=UHZ_DETECTION_REPORT_$V, alias=UHZ_DETECTION_REPORT
  bridge   = src=HAZARDSET_REQUEST_ALL,   alias=HAZARDSET_REQUEST
}
```

In addition to the sensor variables discussed previously, certain basic mission variables, e.g., DEPLOY, RETURN are shared to allow for basic command-and-control of the vehicles. More on uFldShoreBroker application can be found in the documentation for that application. In short, the application allows for configurable autonomatic point-to-point sharing of information as vehicles become known.

### 4.4.2 Variable Bridges from the Vehicle to the Shoreside

Most of the variables shared from the vehicle to the shoreside were discussed in the previous secions, 4.2 and 4.3. The variable share configuration, for variables sent fom the vehicle to the shoreside, is handled on vehicle with the uFldNodeBroker application. Below is the configuration used in the Jake mission:

```
ProcessConfig = uFldNodeBroker
{
  AppTick   = 1
  CommsTick = 1

  TRY_SHORE_HOST = pshare_route=multicast_9

  BRIDGE = src=VIEW_POLYGON
  BRIDGE = src=VIEW_POINT
  BRIDGE = src=VIEW_SEGLIST
  BRIDGE = src=APPCAST
  BRIDGE = src=UHZ_CLASSIFY_REQUEST
  BRIDGE = src=UHZ_SENSOR_REQUEST
  BRIDGE = src=UHZ_CONFIG_REQUEST
  BRIDGE = src=HAZARDSET_REPORT
  BRIDGE = src=NODE_REPORT_LOCAL,  alias=NODE_REPORT
  BRIDGE = src=NODE_MESSAGE_LOCAL, alias=NODE_MESSAGE
}
```

In addition to the sensor variables discussed previously, a few geometry variables, e.g., POLYGON, are shared to the shore for rending thing such as the vehicle search path or search region in the shoreside display. More on the uFldNodeBroker application can be found in the documentation for that application. In short, the application allows for configurable autonomatic point-to-point sharing of information as the location of the shoreside is discovered, without requiring changes to the misison files if the shoreside location, i.e., IP address, changes.

## 4.5 Hazard Files in the Jake Mission

There are several example hazard files included in the Jake mission. The default file used is `hazards.txt`, but there are additionally four other files:

- `hazards_01.txt`
- `hazards_02.txt`
- `hazards_03.txt`
- `hazards_04.txt`

They vary in (a) number of benign objects, (b) number of hazards, (c) hazard resemblance of the benign objects, and (d) aspect sensitivity of objects. Each of these files was created with the gen_hazards utility distributed with moos-ivp. The first line of each file contains a comment, a line showing exactly the command-line invocation of gen_hazards used in making the file. You are welcome to create your own to test situations beyond those in the handful of files included in the Jake mission.

To launch the mission with a different hazard file, the `--hazards` command-line switch is supported with the provided launch script. For example:

```
$ ./launch.sh --hazards=hazards_03.txt 10
```

As before, the `10` on the command-line specifies the time warp and individual machines may vary in the magnitude of time warp supported, correlated to the machines raw computing capacity.

# A Hazard File Format

A *hazard file* is a simple text file that:

- describes a hazard field, typically with one object per line,
- is read by uFldHazardSensor to determine the simulated laydown,
- is read by uFldHazardMetric to have a ground truth to grade against,
- may be generated with the gen_hazards command line tool.

A typical hazard file is shown below. The first line is a comment indicating the command line invocation responsible for generating this particular file, using the gen_hazards utility described separately.

```
// gen_hazards --polygon=-100,0:-100,-300:300,-300:300,0 --objects=5,hazard --objects=8,benign
hazard = x=-151,y=-217.3,label=01,type=hazard
hazard = x=263.2,y=-8.5,label=02,type=hazard
hazard = x=48.5,y=-195.7,label=03,type=hazard
hazard = x=165.9,y=-116.4,label=04,type=hazard
hazard = x=101.3,y=-159.3,label=05,type=hazard
hazard = x=257.6,y=-131.3,label=06,type=hazard
hazard = x=217.3,y=-16.7,label=07,type=hazard
hazard = x=-14.2,y=-293.60001,label=08,type=hazard
hazard = x=260.2,y=-66.2,label=09,type=hazard
hazard = x=-65.8,y=-125.2,label=10,type=hazard
hazard = x=171.9,y=-253.7,label=11,type=benign
hazard = x=-150.3,y=-117.5,label=12,type=benign
hazard = x=-59.8,y=-294.1,label=13,type=benign
hazard = x=98.2,y=-127.7,label=14,type=benign
hazard = x=-178.8,y=-234,label=15,type=benign
hazard = x=24,y=-61,label=16,type=benign
hazard = x=250.3,y=-214.6,label=17,type=benign
hazard = x=97.7,y=-245.5,label=18,type=benign
```

The gen_hazards utility will ensure that each hazard has a unique label. The uFldHazardSensor will complain if there are duplicate labels. The hazard file may also be further edited to change rendering hints for the default shape, color or size of the object. These hints are used, for example, by uFldHazardSensor to generate VIEW_MARKER postings consumable by pMarineViewer.

## B Hazard Resemblance Factors

Benign objects may contain an additional *resemblance factor* in the range of $[0,1]$, where zero indicates the object shares very little resemblance to a hazard, and vice versa for values of one. Each benign object $i$ in the hazard file has an associated resemblance factor $R_i$. When the `uFldHazardSensor` application publishes visual markers for benign objects, it reflects the hazard resemblance factor in the marker transparancy. A benign object with hazard resemblance of 0.64 might be posted as:

```
VIEW_MARKER = x=7,y=89,width=8,primary_color=light_blue,type=triangle,label=991,
              fill_transparency=0.64
```

The overall rendering of the hazard field in `pMarineViewer` may look something similar to that in Figure 4.

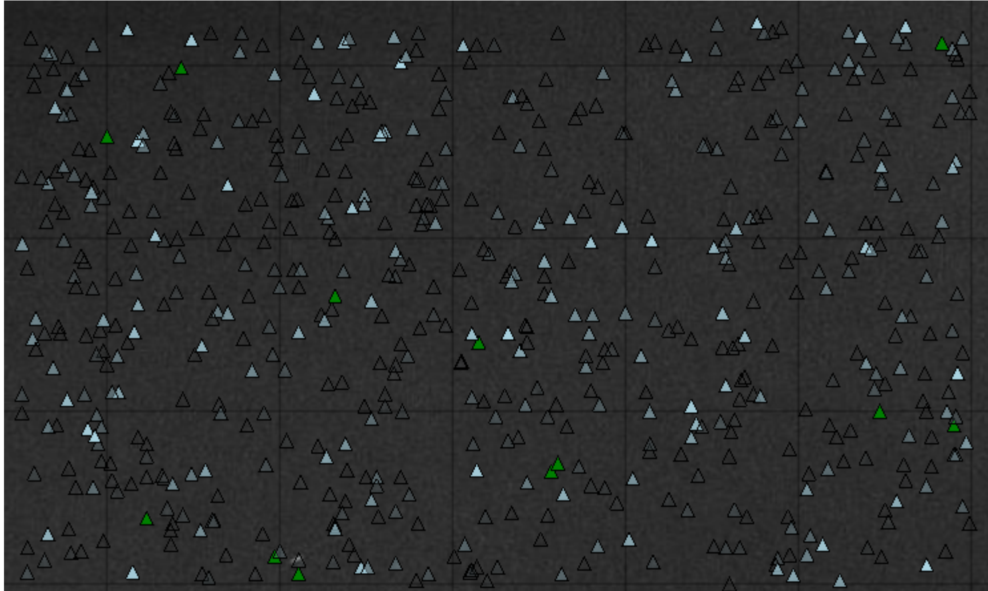


Figure 4: **A simulated hazard field with resemblance factors:** Green markers represent hazards. Light blue markers represent benign objects. The more transparent the markers, the lower the resemblance factor. The resemblance factor ranges from $[0,1]$.

### Influence of Resemblance Factor on $P_{FA}$

The probability of a detection declared on a benign object (a false alarm), is normally given by $P_{FA}$. The hazard resemblance factor is further applied to provide a unique probability of false alarm for a given object $i$.

$$P_{FA}(i) = (P_{FA} + R_i)/2$$

The exception is when $P_{FA} = 1$. In this case $P_{FA}(i)$ is also 1.

**Influence of Resemblance Factor on $P_C$**

The probability of a correct classification on a given object is normally given by $P_C$, in the range of $[0.5, 1]$. The hazard resemblance factor is further applied to provide a unique probability of correct classification for a given object $i$.

$$P_C(i) = P_C + (1 - P_C)(1 - R_i)$$

In short, $P_C(i)$, at worst, is given by $P_C$ for objects closely resembling a hazard ($R_i = 1$), and $P_C$ approaches 1, i.e., perfection, for objects that look nothing like a hazard ($R_i = 0$).

## C Generating Your Own Hazard Files

Random hazard field files may be generated with the gen_hazard utility. For example, the following command-line invocation will generate the file shown in Section A:

```
$ gen_hazards --polygon=-100,0:-100,-300:300,-300:300,0 --objects=5,hazard --objects=8,benign
```

Hazard resemblance factors may be automatically generated with the `--exp=N` command line option. When this option is used, each benign object will have a initial resemblance factor chosen with a uniform random variable in the range of $[0, 1]$. This number is then raise to the power of $N$. For example, the below command line invocation:

```
$ gen_hazards --polygon=-100,0:-100,-300:300,-300:300,0 --objects=5,hazard --objects=8,benign
 --exp 3 > hazards.txt
```

would create a new file, `hazards.txt`, with the below content. The first line is a comment line that reflects the command line invocation responsible for the output.

```
// gen_hazards --polygon=-150,-75:-150,-400:400,-400:400,-75 --objects=5,hazard --objects=8,benign
--exp=3
hazard = x=292,y=-253,label=22,type=hazard
hazard = x=224,y=-322,label=51,type=hazard
hazard = x=-1,y=-136,label=37,type=hazard
hazard = x=131,y=-115,label=55,type=hazard
hazard = x=-14,y=-201,label=76,type=hazard
hazard = x=38,y=-230,label=87,type=benign,hr=0.0199
hazard = x=228,y=-134,label=49,type=benign,hr=0.02172
hazard = x=262,y=-346,label=9,type=benign,hr=0.06842
hazard = x=30,y=-310,label=45,type=benign,hr=0.01544
hazard = x=-29,y=-141,label=59,type=benign,hr=0.00417
hazard = x=57,y=-343,label=23,type=benign,hr=0.17845
hazard = x=148,y=-387,label=86,type=benign,hr=0.31861
hazard = x=30,y=-313,label=84,type=benign,hr=0.37481
```

The gen_hazards tool does not generate output regarding the shape, color or width of an object. Those characteristics need to be added by hand in the hazard file if the user desires.

One caveat is that the polygon provided to gen_hazards, must be convex. For example, note the following erroneous invocation and output (two vertices do not constitute a convex polygon):

```
$ gen_hazards --polygon=400,-400:400,-75 --objects=3,hazard --objects=3,benign
Invalid/Non-convex polygon specified: 400,-400:400,-75
```

# D Hazard File Renderings

A hazard file may be rendered as shown in Figure 5. In this case, the hazards are read by uFldHazardSensor, posted to the MOOSDB as VIEW_MARKER objects, and read and rendered by pMarineViewer.

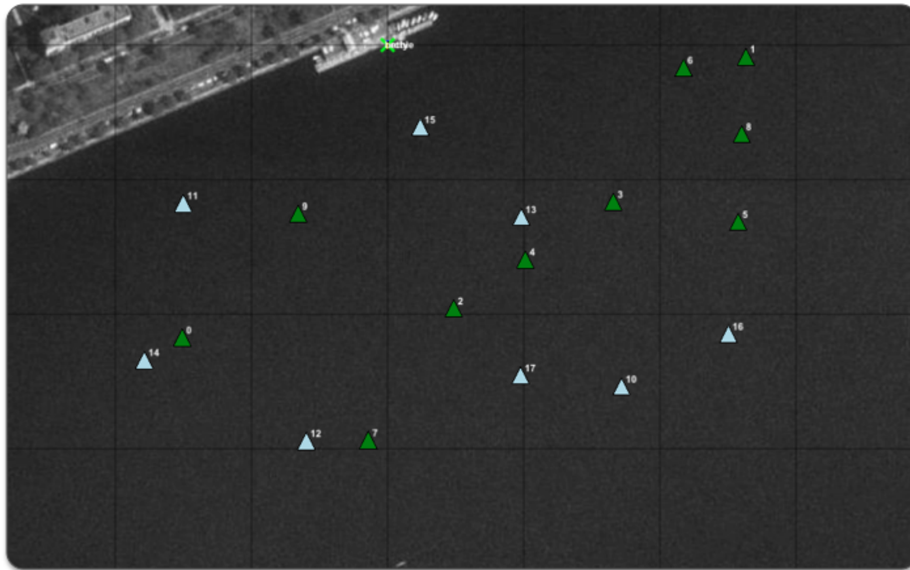


Figure 5: **Simulated Hazard Field:** A hazard field with 18 objects is shown, some are hazardous objects, some are benign objects.

Since the hazard (and benign) objects in pMarineViewer, are simply VIEW_MARKER objects, they may be toggled off and on with the 'm' key and the labels may be toggled off and on with the 'M' key. The can be rendered larger and smaller with alt-'m' and ctrl-'m' respectively, and returned to their original size with ctrl-alt-'m'.

The shapes and colors of both hazard and benign objects may also be altered within the uFldHazardSensor configuration. By default, both hazards and benign objects are rendered as triangles, and hazards by default are rendered green, and benign objects are light blue. Their rendered width by default is 8 meters. These defaults may be altered with the following six uFldHazardSensor configuration parameters:

```
default_hazard_shape = triangle            // default
default_hazard_color = green               // default
default_hazard_width = 8                   // default

default_benign_shape = triangle            // default
default_benign_color = light_blue          // default
default_benign_width = 8                   // default
```

Other shapes allowable are square, circle, and diamond. These are *default* values. Rendering characteristics may be overridden for any particular object. For example, consider the following entry in a hazard file:

```
hazard = x=224,y=-322,label=51,type=hazard,shape=square,color=yellow,width=20
```

The specified `shape=square`, `color=yellow` and `width=20` will override any defaults, making this particular object stand out visually.

# E Serialization of Hazard and HazardSet Information

In the geometry library there are two classes defined relevant to this lab. The `XYHazard` class and the `XYHazardSet` class. The latter is essentially just a collection of the former. It is recommend that you use this data structure internally in your programs, but especially recommended that you use this data structure for serializing.

Serializing and De-serializing are explained further below, but perhaps the best explanation is by example in the straw-man `uFldHazardMgr` application. The `handleMailDetectionReport()` function handles incoming reports from the sensor and builds up a hazardset object for later posting to the `MOOSDB`.

## E.1 Serializing and De-serializing XYHazard Objects

Hazard objects are dealt with in the `XYHazard` class in `lib_ufld_hazards`. It may be best to just explore the class header file to understand its function, but the below example snippets should give the general idea:

```
#include "XYHazard.h"

// Build the hazard object
XYHazard my_hazard;
my_hazard.setX(5);
my_hazard.setY(8);
my_hazard.setType("hazard");
my_hazard.setLabel("128");

// Serialize the object into a string
string msg = my_hazard.getSpec()

// Post to the MOOSDB
Notify("HAZARD_INFO", msg);
```

The above would result in the following posting to the MOOSDB:

```
HAZARD_INFO = "x=5,y=8,type=hazard,label=128"
```

In the reverse direction, creating an `XYHazard` object from a string:

```
#include "XYFormatUtilsHazard.h"

string msg = "x=5,y=8,type=hazard,label=128";

XYHazard my_hazard = string2Hazard(msg);
cout <<  my_hazard.getX() << ", ";
cout <<  my_hazard.getY() << ", ";
cout <<  my_hazard.getType() ", ";
cout <<  my_hazard.getLabel() << endl;
```

The above would result in the following written to the terminal:

```
5, 8, hazard, 128
```

## E.2 Serializing and De-serializing XYHazardSet Objects

HazardSet objects are essentially collections of XYHazard objects. They are dealt with in the `XYHazardSet` class in `lib_ufld_hazards`. It may be best to just explore the class header file to understand its function, but the below example snippets should give the general idea:

```
#include "XYHazardSet.h"

// Build the hazards
XYHazard hazard_01;
hazard_01.setX(15);
hazard_01.setY(18);
hazard_01.setLabel(128);

XYHazard hazard_02;
hazard_02.setX(35);
hazard_02.setY(38);
hazard_02.setLabel(92);

// Build the hazardset
XYHazardSet my_hazard_set;
my_hazard_set.setSource("betty");
my_hazard_set.addHazard(hazard_01);
my_hazard_set.addHazard(hazard_02);

// Serialize the hazardset
string msg = my_hazard_set.getSpec()

// Post to the MOOSDB
Notify("HAZARDSET_REPORT", msg);
```

The above would result in the following posting to the MOOSDB:

```
HAZARDSET_REPORT = "source=betty # x=15,y=18,type=hazard,label=128 # \
                                   x=35,y=38,type=benign,label=92"
```

## References


[1] Michael R. Benjamin. MOOS-IvP Autonomy Tools Users Manual Release 13.5. Technical Report www.moos-ivp.org/docs, MIT Computer Science and Artificial Intelligence Lab, May 2013.

[2] Michael R. Benjamin, Henrik Schmidt, Paul M. Newman, and John J. Leonard. An Overview of MOOS-IvP and a Users Guide to the IvP Helm - Release 13.5. Technical Report www.moos-ivp.org/docs, MIT Computer Science and Artificial Intelligence Lab, May 2013.