Lab 5 - Introduction to Helm Autonomy

2.680 Unmanned Marine Vehicle Autonomy, Sensing and Communications

Feb 27th, 2024

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

1 Overview and Objectives

In today's lab we will create our own autonomy missions by constructing helm configuration files. Up to now, configuring a MOOS community has consisted of configuring a single `.moos` file. In using the helm, a second file, referred to as the behavior file, will be constructed, with suffix `.bhv`. We begin with the alpha mission downloaded along with the moos-ivp tree and proceed by making our first simple missions emphasizing the usage of the basic helm features and MOOS autonomy utilities. This is followed by a couple more complex applications assignments.

- The Basic Helm Structure
- The High-Level Helm State - Putting the Helm into Drive
- Launching the Alpha Autonomy Mission
- Understanding the Helm During the Alpha Mission Execution
- Methods in pMarineViewer for Controlling a Mission
- Assignment: Modify the Alpha Mission to Accept a User Return Point
- Assignment: Build the Single Double Loiter Bravo Missions
- Assignment: A Double Loiter Bravo Mission with Periodic Surfacing

2 Preliminaries

Make Sure You Have the Latest Updates

It is possible we will be making changes to the MOOS-IvP tree during the semester. Always make sure you have the latest code:

```
$ cd moos-ivp
$ svn update
```

If you see see a response similar to the below, indicating that no updates were pulled in from the server, then you shouldn't need to re-build the software.

```
Updating '.':
At revision 9615.
```

Otherwise you will need to rebuild:

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

Make Sure Key Executables are Built and In Your Path

This lab does assume that you have a working MOOS-IvP tree checked out and installed on your computer. To verify this make sure that the following executables are built and findable in your shell path:

```
$ which MOOSDB
/Users/you/moos-ivp/bin/MOOSDB
$ which pHelmIvP
/Users/you/moos-ivp/bin/pHelmIvP
```

If unsuccessful with the above, return to the steps in Lab 1:
http://oceanai.mit.edu/ivpman/labs/machine_setup

Where to Build and Store Lab Missions

As with previous labs, we will use your version of the `moos-ivp-extend` tree, which by now you should have re-named something like `moos-ivp-jsmith`, where your email may be `jsmith@mit.edu`. In this tree, there is a missions folder:

```
$ cd moos-ivp-extend
$ ls
CMakeLists.txt  bin/        build.sh*     docs/     missions/       src/
README          build/      data/         lib/      scripts/
```

For each distinct assignment in this lab, there should be a corresponding subdirectory in a `lab_05` sub-directory of the `missions` folder, typically with both a `.moos` and `.bhv` configuration file. See Section 8 for the full requested file structure.

Documentation Conventions

To help distinguish between MOOS variables, MOOS configuration parameters, and behavior configuration parameters, we will use the following conventions:

- Applications or any executables are rendered in `magenta`, such as `MOOSDB` or `pHelmIvP`.
- MOOS variables are rendered in `green`, such as `IVPHELM_STATE`, as well as postings to the `MOOSDB`, such as `DEPLOY=true`.
- MOOS configuration parameters are rendered in `blue`, such as `AppTick=10` and `verbose=true`.
- Behavior parameters are rendered in `brown`, such as `priority=100` and `endflag=RETURN=true`.

More MOOS / MOOS-IvP Resources

A few further resources are worth mentioning for following up this lab with your own exploration.

- See the slides from the today's class which give a bit more background into marine autonomy and the IvP Helm: http://oceanai.mit.edu/2.680/docs/2.680-06-intro_to_autonomy_2024.pdf
- The IvP Helm and Utilities documentation: http://oceanai.mit.edu/ivpman
- The moos-ivp.org website documentation: http://www.moos-ivp.org

3 The Basic Helm Structure

In the course of today's lab it may be helpful to keep in mind the high-level components of the helm and flow of events covered in today's class. The key components to keep in mind are shown

in Figure 1 below. All behaviors produce either an objective function, or a set of variable-values pairs posted to the MOOSDB. The helm minimally posts, on each iteration, a set of variable-value pairs representing the current decision. This is typically DESIRED_HEADING and DESIRED_SPEED, but also DESIRED_DEPTH when the helm is implemented on an underwater vehicle.

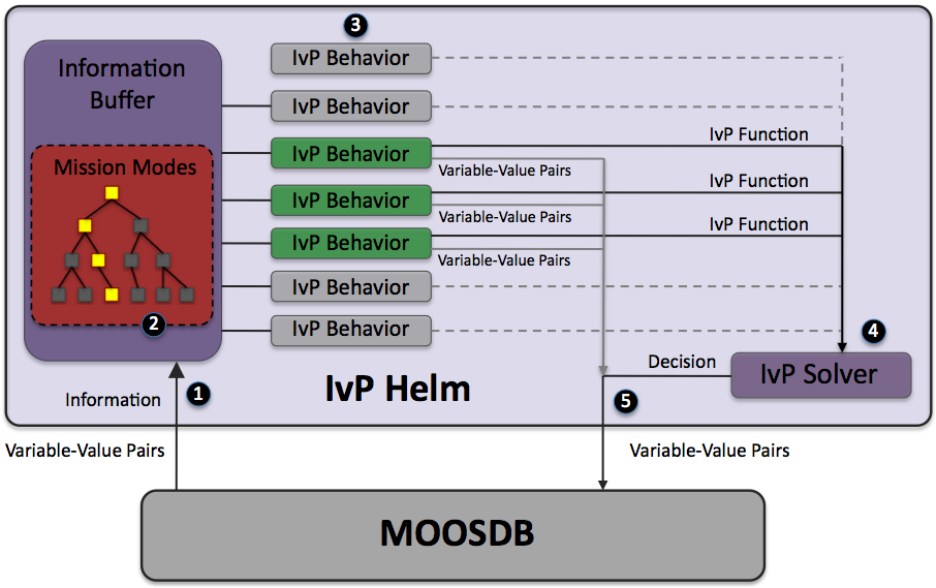

Figure 1: **The Helm Iterate Loop:** (1) Mail is read from the MOOSDB. It is parsed and stored in a local buffer to be available to the behaviors. (2) If there were any mode declarations in the mission behavior file, they are evaluated at this step. (3) Each behavior is queried for its contribution and may produce an IvP function and a list of variable-value pairs to be posted to the MOOSDB at the end of the iteration. (4) The objective functions are resolved to produce an action, expressible as a set of variable-value pairs. (5) All variable-value pairs are published to the MOOSDB for other MOOS processes to consume.

The behaviors get their information from a common data structure on each iteration. This information buffer gets its information from the MOOSDB by processing mail just like any other MOOS app. The behaviors perform their function effectively in parallel. Internally the helm calls for their contribution in sequence, but since they all operate on an identical information buffer, and no behavior produces output used by any other behavior as input, they are essentially operating independently and in parallel. Behavior coordination takes place in the solver, ultimately choosing a single decision of `heading`, `speed` and `depth`. Additionally, any behavior may post a variable-value pair to the MOOSDB read by another behavior on the *next* iteration. However, within a given iteration, behaviors are essentially independent.

4 Putting the Helm into Drive: The High-Level Helm State

One of the first topics encountered in using the helm is how to turn it loose! The helm's highest level state description is simple - it is in the PARK or DRIVE state:

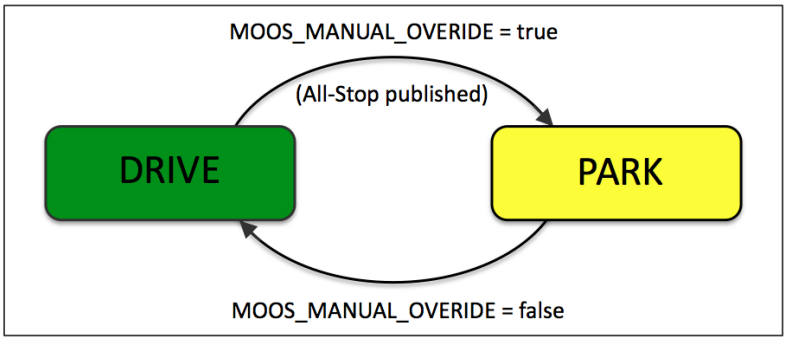

Figure 2: **The Helm State of the IvP Helm**: The helm state has a value of either `PARK` or `DRIVE`, depending on both how the helm is initialized and the mail received by the helm after start-up on the variable `MOOS_MANUAL_OVERRIDE`. The helm may also park itself if an all-stop event has been detected.

Often when a mission is launched, the helm begins in the park state, and the user *deploys* the vehicle by putting it into drive. This is done by changing (poking) the MOOS variable `MOOS_MANUAL_OVERRIDE` from true to false. The helm state may simply be apparent by watching what is going on in the GUI, and often the helm state is also rendered next to the vehicle in a GUI like `pMarineViewer`. But one may also confirm the helm state by scoping on the variable `IVPHELM_STATE`.

Just because a vehicle is not moving doesn't mean the high-level helm state is in park. As with driving a car, there may be several reasons why a car is not moving while the car is still in drive (traffic light, pedestrian crossing, stopping to look at a road sign, etc.). Further information regarding *why* a vehicle is stopped may be gleaned from scoping on the MOOS variable `IVPHELM_ALLSTOP`. We will explore some cases in the first exercise.

5 Experimenting with and Modifying the Alpha Example Mission

In the first exercise in today's lab, the goal is to use the alpha example mission to explore a few ideas with the helm and tools available for configuring and interacting with the helm. As the very first step, we begin by updating the moos-ivp tree and making sure we have the very latest version of the moos-ivp trunk:

```
$ cd moos-ivp
$ svn update
Updated to revision 10507.
$ ./build-moos.sh
$ ./build-ivp.sh
```

The revision number presented to you with `svn update` command may be higher than the number above. Assuming there were no problems with the update and build, everything should be ready to go.

5.1 The basics of launching an autonomy mission

In this part we will:

- Prepare a copy of the alpha mission for experimenting
- Examine the alpha mission file structure
- Launch the alpha mission using MOOS time warp.
- Recognize when something has gone wrong

5.1.1 Make a copy of the alpha mission

In this lab we will mostly be building and modifying mission files rather than writing source code. The mission files consist of `.moos` (MOOS) files, and `.bhv` (Helm) files. The first step is to copy the alpha example mission from the moos-ivp tree into your own moos-ivp-extend tree. This tree by now should be re-named something like `moos-ivp-jsmith` where your MIT email is `jsmith@mit.edu`. As with lab 4, this directory is what you will be submitting in later problem sets, so it's good to begin the habit of working in this directory.

```
$ mkdir moos-ivp-jsmith/missions/lab_05
$ cp -rp moos-ivp/ivp/missions/s1_alpha  moos-ivp-jsmith/missions/lab_05/alpha
```

(Note in the `cp` command we use the `-r` switch to recursively copy the entire directory and the `-p` switch to preserve file meta data such as mode, ownship and timestamps.)

5.1.2 Note the MOOS and helm mission file structures

Note that the alpha mission has two files; the MOOS mission file, and the helm behavior file. The `alpha.moos` contains a configuration block for the `pHelmIvP` MOOS app (the helm). This block looks like:

```
1  ProcessConfig = pHelmIvP
2  {
3    AppTick    = 4
4    CommsTick  = 4
5
6    behaviors  = alpha.bhv
7    domain     = course:0:359:360
8    domain     = speed:0:4:41
9  }
```

The two immediately important components of this are (a) the indication of the behavior file on line 6, and (b) the decision domain specified in lines 7 and 8. The latter indicates that the helm is expected to make a decision for each iteration on the desired heading and speed of the vehicle. The possible heading values are between 0 and 359 on one-degree increments, and the possible speed values are between 0 and 4 meters per second in 0.1 m/s increments.

5.1.3 Launch and experiment with MOOS time warp

The alpha mission is described in a fair amount of detail in the helm documentation:
http://oceanai.mit.edu/ivpman/examples/alpha

Take a few minutes to read this through and then go ahead and launch the mission:

```
$ cd moos-ivp-jsmith/missions/lab_05/alpha/
$ pAntler alpha.moos
```

Or you can launch the mission with the launch script as below. This will launch pAntler and then launch the uMAC interactive utility.

```
$ cd moos-ivp-jsmith/missions/lab_05/alpha/
$ ./launch.sh
```

Things should look similar to the figure in the alpha mission description. One thing you may notice right away is that the simulation progresses slowly. This may be sped up by altering the MOOSTimeWarp. This parameter is set to 1 at the top of the .moos file. Try changing this to something, like 20, and re-launch. You will find that the using a higher time warp is often essential for quick experimentation. A short-cut for using the time warp is to pass the time warp to pAntler on the command-line rather than editing the line in the MOOS file. This is done by:

```
$ pAntler --MOOSTimeWarp=20
```

Many mission folders also have a launch script (`launch.sh`) to make launching even easier, especially with multi-vehicle missions we encounter later on. So you could also launch the alpha mission with:

```
$ ./launch.sh 20
```

where 20 is the time warp.

5.2 Understanding the helm during mission execution

In this part we will:

- Find the vehicle helm-state during mission execution
- Understand the difference between helm-state and all-stop status
- Understand the behavior condition and endflag parameters

5.2.1 Find the vehicle helm state during mission execution

The helm may be characterized at its highest level by two MOOS variables, IVPHELM_STATE and IVPHELM_ALLSTOP. These two issues are described in Sections 6.2 and 6.3 in the helm documentation:

http://oceanai.mit.edu/ivpman/helm/helm_state

Take a few minutes to read this through.

The helm-state and all-stop status may be determined at run-time by simply scoping on the two MOOS variables. In typical pMarineViewer configurations, they are also typically viewable right next to the vehicle icon. When you launch the alpha mission, you should see the following text right next to the vehicle icon: `"(PARK)(Manual Override)"`. The first indicates the helm-state, and the second indicates the all-stop status. When the vehicle is deployed by hitting the DEPLOY button, the helm-state changes to DRIVE, and the all-stop status disappears. Examination of the IVPHELM_ALLSTOP variable would show that it equals `"clear"`, but pMarineViewer is implemented to just not render the all-stop status when it is equal to `"clear"`.

5.2.2 Understand the difference between the helm-state and all-stop status

The helm helm-state is always either in PARK or DRIVE. The all-stop status is a way of further indicating why the helm is not moving the vehicle (equivalent to producing a DESIRED_SPEED decision).

Experiment with this a bit. First note that, in the alpha mission, once the vehicle completes its waypoints and returns to its mission, the vehicle is still in the DRIVE helm-state, but the all-stop status indicates `"NothingToDo"`. This simply means that all behaviors have *completed*, and the mission is effectively over.

Try poking the MOOSDB during the alpha mission with the following variable-data pair: DEPLOY=false. What happens?

5.2.3 Understand the behavior condition and endflag parameters

Two key parameters defined for all behaviors are the condition and endflag parameters. They are described here:

http://oceanai.mit.edu/ivpman/helm/conditions

and

http://oceanai.mit.edu/ivpman/helm/flags

Note in the alpha mission (in alpha.bhv) the conditions and endflags for the waypoint-survey behavior:

```
condition = RETURN = false
condition = DEPLOY = true
endflag   = RETURN = true
```

and the conditions and endflags for the waypoint-return behavior

```
condition  = RETURN = true
condition  = DEPLOY = true
endflag    = RETURN = false
endflag    = DEPLOY = false
```

Both behaviors are conditioned on the variable DEPLOY being true. But the waypoint behavior will only be active if RETURN is false. Also note that these two conditions are initialized in the top of the behavior file with the two lines `initialize DEPLOY=false` and `initialize RETURN=false`.

When the waypoint-survey behavior completes its set of waypoints, it posts its endflag, RETURN=true, which is just what the waypoint-return behavior needs to satisfy its condition and begin executing. This mechanism is not only a way to string together a sequence of behaviors, essentially a *plan*, but run conditions may switch in and out of satisfaction to implement a helm mode-space where behaviors are periodically active and not active.

5.3 Methods in pMarineViewer for Understanding and Controlling a Mission

In this part we will:

- Use pMarineViewer buttons for poking the MOOSDB and altering the helm
- Scoping with pMarineViewer
- Poking with pMarineViewer geo-referenced mouse clicks

5.3.1 Use pMarineViewer Buttons for Poking the MOOSDB and Altering the Helm

In the alpha mission, the DEPLOY button is configured to start the mission upon a user click. This is not hard-coded in the pMarineViewer implementation, but represents how this button was configured for specific use in this mission. The pMarineViewer tool is a primary tool used in this and later labs. It is used for rendering a mission as it unfolds, and for interacting with the vehicle while it is deployed. It is used not only in simulation, but also when deployed on the water. It has a few configuration hooks that are worth knowing about.

In the case of the alpha mission, the DEPLOY and RETURN buttons are configured with the following three lines found in the `alpha.moos` file:

```
button_one = DEPLOY # DEPLOY=true
button_one = MOOS_MANUAL_OVERRIDE=false # RETURN=false
button_two = RETURN # RETURN=true
```

The syntax and general usage is described in pMarineViewer documentation (in the section "Command and Control"). In short, there are up to twenty configurable buttons, button_one through

button_twenty, or button_1 through button_20. If they are not configured, they are not shown. They may be configured to make one or *more* distinct pokes upon click.

Try configuring the third and fourth buttons in the alpha mission to poke something to the MOOSDB upon click, and verify this works by scoping on the variable. This step of creating a specific button configuration for simple command and control will be a component of lab assignments frequently in this class. Try it here.

5.3.2 Scoping with pMarineViewer

The pMarineViewer application may also be used as limited scope. The bottom row of fields in the window show the variable name, time of last write, and variable value, forming a "single variable scope". The variable to be scoped is set in the pMarineViewer configuration block with a line like:

```
scope = NAV_X
```

Multiple lines may be provided. The scoped variable may be changed via the MOOS-Scope pull-down menu in pMarineViewer, or changed by repeatedly hitting `ctrl-'/'`. A variable may be added to the scope list by typing `<shift> 'A'` at any time, and entering the variable name in the dialog box.

2022 Note: Recently, "Realmcasting" has been added to MOOS-IvP with support n pMarineViewer. To enable realmcasting, just run pRealm in your mission (no configuration block required). During your mission you can see the realmcasting output by toggling between realmcasting and appcasting with the `'a'` key.

5.3.3 Poking with pMarineViewer geo-referenced mouse clicks

The pMarineViewer app supports a further useful method for poking the MOOSDB with mouse clicks containing the location of the click in the operation area in the value poked to the MOOSDB. This is described in pMarineViewer documentation, http://oceanai.mit.edu/ivpman/apps/pMarineViewer, in the section "Custom Poking of the MOOSDB with the Operation Area Position". Try adding the below line, for example, in the pMarineViewer configuration block of your `.moos` file.

```
left_context[view_point]  = VIEW_POINT = x=$(XPOS),y=$(YPOS),label=hello
```

This line adds the ability to left-click on the pMarineViewer window with the result that a VIEW_POINT message will be posted to the MOOSDB with the location of the mouse-click embedded in the message. It is possible to configure pMarineViewer with a selectable list of left-mouse-click contexts. For example, add the below line to the first one above. You should be able to then choose what action a left click performs by using the Mouse-Context pull-down menu.

```
left_context[view_poly]  = VIEW_POLYGON = format=radial,x=$(XPOS),y=$(YPOS),
                           radius=10, pts=8, edge_size=1,label=mypoly
```

In the above line, don't actually use the backslash character at the end of the line. It all needs to be on one line - a limitation of .moos files, for now at least.

Try configuring your mission with a third context, e.g., `"view_both"`, such that both types of events occur with the same mouse click. You will need this trick in Assignment 2 below.

5.4 Assignment 1 (self check off) - The Alpha Return Mission

This assignment involves the modification of the alpha example mission to accept a return waypoint for the vehicle based on a user click in the pMarineViewer window. Your goals are:

- Modify the `alpha.moos` and `alpha.bhv` (if necessary) files to accept a user left-mouse click in pMarineViewer determining the point to where the vehicle should return. Your modification should result in the posting of a point to the pMarineViewer window with the label "return_point" immediately upon a user click. After the vehicle has completed its waypoint survey, it will proceed to the return point.

Hint: You will need to utilize the updates parameter defined for all behaviors. The return home waypoint behavior already has an updates parameter configured to name the MOOS variable RETURN_UPDATE. Your mouse click needs to result in a posting to this variable. The contents of this posting should be a string, where the string is a drop-in replacement for any configuration line in the behavior configuration block. More on the updates parameter can be found here:

http://oceanai.mit.edu/ivpman/helm/updates

It should look something like the video posted at:

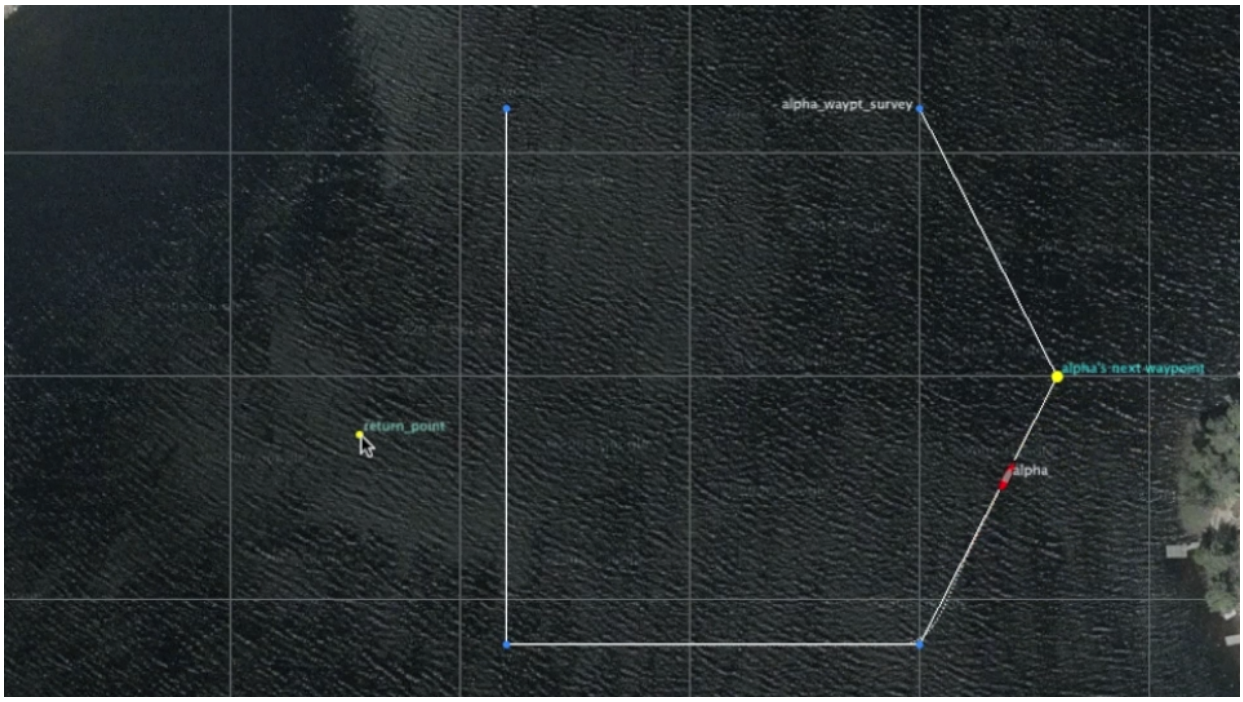

Figure 3: The modified alpha mission with user commanding the return point.

video:(0:23): https://vimeo.com/87301755

5.5 Assignment 2 (check off) - The Alpha Return Now Mission

Extending the first mission, configure your mission and `pMarineViewer` to have a second left-mouse-click context directing the vehicle to return as soon as the mouse is clicked. Instead of the vehicle waiting until the waypoint-survey behavior to complete, a left-mouse click results in the immediate return to the specified point.

Hint: the only thing that needs to be done additionally in this mission is to tie two MOOS postings to one mouse click.

It should look something like the video posted at:

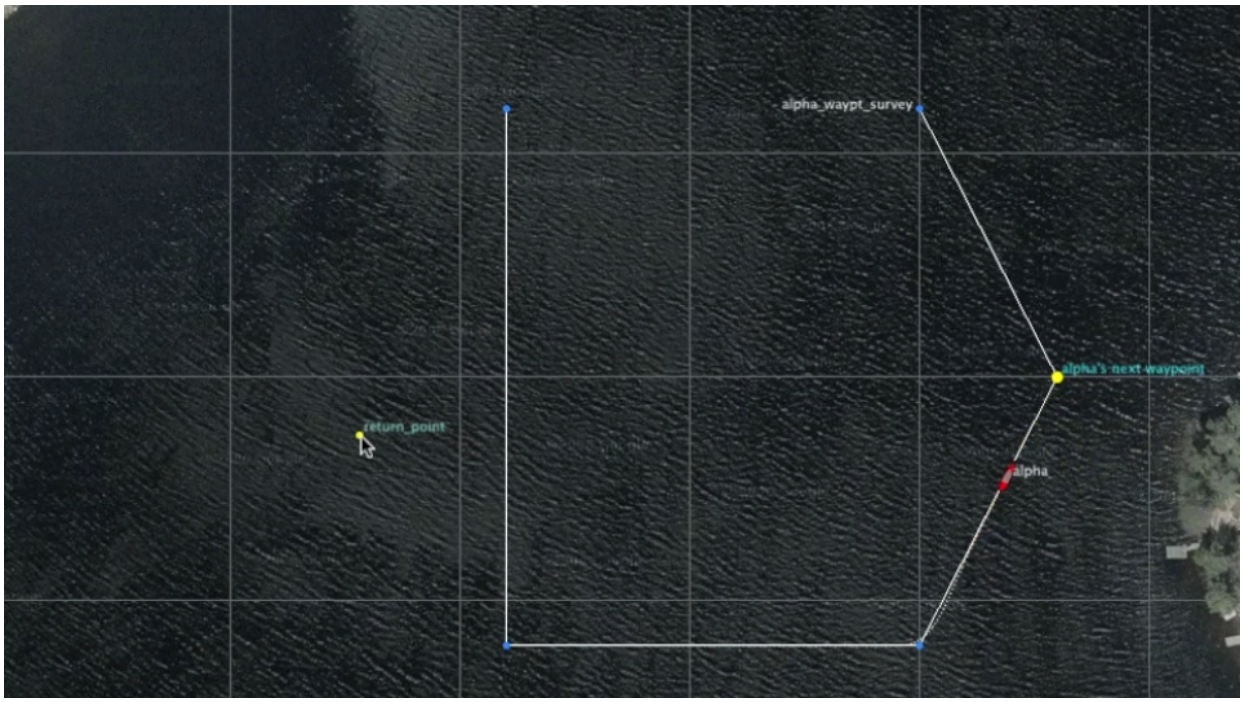

Figure 4: The modified alpha mission with user commanding the return point. The selection of the return point also triggers the switch to return mode.

`video:(0:15): https://vimeo.com/87301981`

6 Building Your First Autonomy Mission - The Bravo Mission

In the second part of today's lab, the goal is to create an autonomy mission from scratch. Primarily we will be using the Loiter behavior, but we will also have occasion to use the Waypoint, ConstantDepth, ConstantSpeed, and Timer behaviors. We will start by constructing a mission for a surface vehicle, but will migrate shortly to configuring for an underwater vehicle.

Much of the information you will need regarding the workings of various behaviors and the helm are described in the helm documentation (http://oceanai.mit.edu/ivpman). Several of the target missions we will be building have been simulated prior to the lab with short videos available.

6.1 Assignment 3 (self check off) - The Bravo Loiter Mission

In this part we will:

- Prepare a new mission from scratch
- Get familiar with the Loiter behavior
- Introduce the notion of behavior run-states.
- Understand the `"perpetual"` flag defined on all behaviors.

Our first step is to create a new mission and behavior file, and call them `bravo.moos` and `bravo.bhv`. This should be put into a folder in `missions/lab_05/` called `bravo_loiter`. It is certainly fine to copy the alpha mission as a starting point. The bravo mission should be configured with the following features:

- It should have a Loiter behavior, which is primarily active upon an initial deploy. It should have a location `x=100,y=-75`, a radius of 30 meters, and the loiter polygon should have 8 vertices. It should be set with a loiter speed of 2.5 m/sec. It should loiter counter-clockwise.
- It should have a waypoint behavior that simply returns the vehicle to the vehicle starting position `x=0,y=0`, when the variable `RETURN=true` as in the alpha mission.
- The Loiter behavior should utilize the `duration` parameter to automatically "complete" after 150 seconds, triggering the return waypoint behavior.

It should look something like the video posted at:

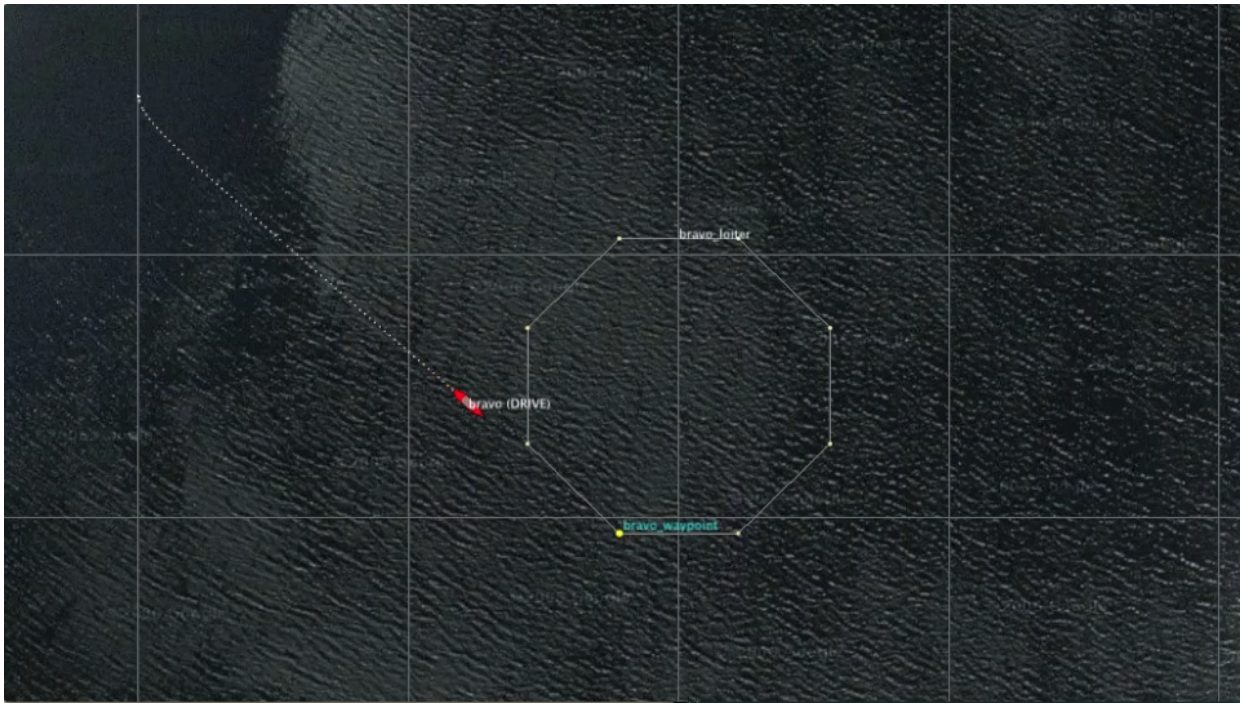

Figure 5: A simple loiter mission (the Bravo Loiter Mission) - the vehicle loiters at the prescribed location for the prescribed amount of time, and returns home automatically.
video:(0:14): https://vimeo.com/87311982

The Loiter behavior is in the helm documentation (http://oceanai.mit.edu/ivpman/bhvs/Loiter). The primary parameters of interest here are the `polygon`, `clockwise`, and `speed` parameters. Use the `format=radial` format for describing the loiter polygon as described in the above referenced Loiter documentation in the section `"Setting and Altering the Loiter Region"`.

The waypoint return behavior is configured similarly to the alpha mission, but more information on this behavior may be found at http://oceanai.mit.edu/ivpman/bhvs/Waypoint.

The duration behavior parameter is defined for all behaviors. Most behaviors regard the duration to be limitless if left unspecified. A more detailed discussion of this parameter may be found in the helm documentation: http://oceanai.mit.edu/ivpman/helm/param_duration.

We will be exploring the concepts of behavior run-states and run-flags in this and the following missions. A good discussion of this may be found here: http://oceanai.mit.edu/ivpman/helm/run_states.

6.2 Assignment 4 (self check off) - The Bravo Double Loiter Mission

In this part we will:

- Explore a non-sequential mission - a mission that alternates between modes.
- Become familiar with the `duration`, `endflag`, and `perpetual` behavior parameters.
- Learn about the special but versatile Timer behavior in `pHelmIvP`.

Our next step is to add a second loiter behavior to the bravo mission. The idea is to construct a mission where the vehicle is able to periodically switch between loitering at the two locations. We will experiment with ways to enable this switching, both automatically and via user interaction. The second bravo mission should be configured with the following features:

- Make a copy of the previous bravo mission, making a new directory in the `missions/lab_05/` folder called `bravo_loiter_dbl`.
- The new bravo mission should have a second loiter behavior, which is not active upon an initial deploy. It should have a location `x=160,y=-50`, a radius of 20 meters, and the loiter polygon should have 8 vertices. It should be set with a loiter speed of 2.5 m/sec.
- Make the two loiter behaviors mutually exclusive using their `condition` parameters, e.g., `condition=LOITER_REGION=west`.
- Utilize the `duration`, `endflag`, and `perpetual` parameters to accomplish the periodic switching between loiter regions. Set the `duration` to be 150 seconds for each, use a couple of `endflag` parameters in each behavior to both (a) trigger the conditions of the other loiter behavior, and (b) negate the condition of the behavior ending. Set the `perpetual` parameter to be true so that a completed behavior does not complete permanently, but simply awaits its conditions to once again be satisfied.
- **Note:** You may need to understand how the `duration` timer works. Recall the `duration` feature is defined for all behaviors. By default the timer re-starts immediately once it has count down completely and `perpetual` is set to true. This may not be what you want. You may want your behavior to wait until it is again running (logic conditions satisfied) before resuming the duration timer count-down. In this case you would need to set `duration_idle_decay` to be false. The default is true.

It should look something like the video posted at:

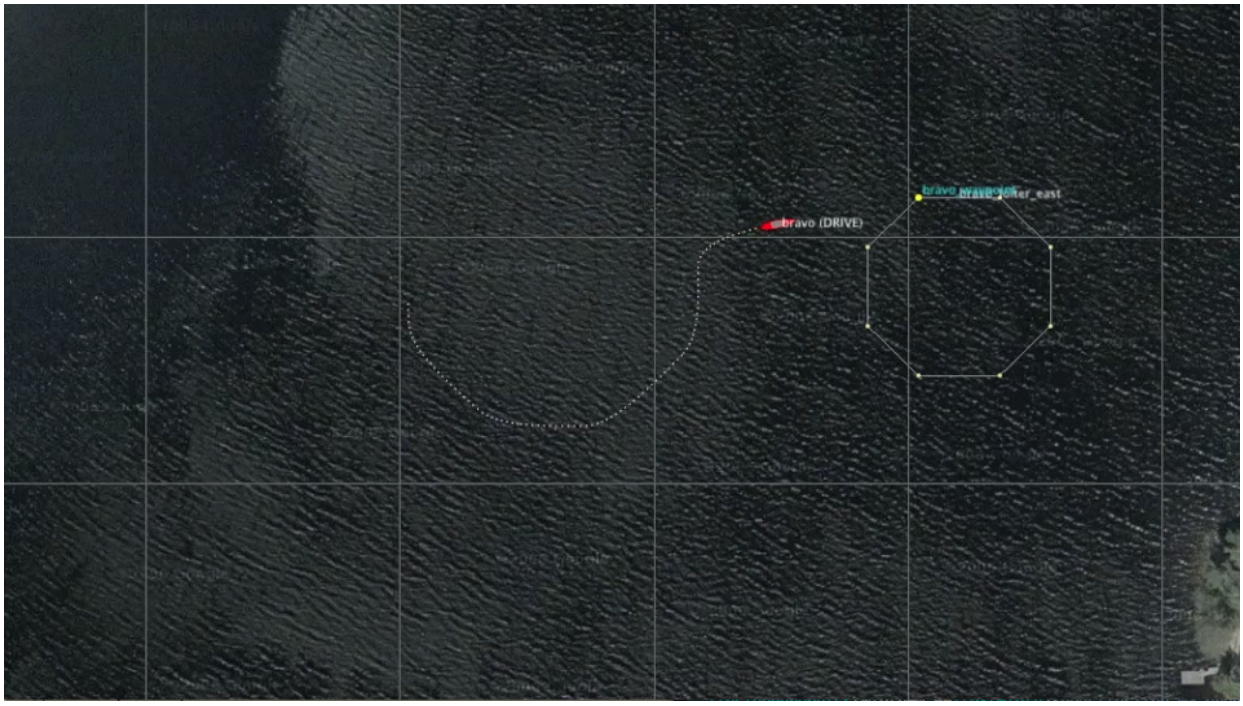

Figure 6: A simple loiter mission (the Bravo Double Loiter mission) - the vehicle loiters at one of two prescribed locations for a prescribed amount of time before automatically switching to the other location. The vehicle doesn't return until the user explicitly calls for its return by hitting the RETURN button.

video:(0:42): https://vimeo.com/87349082

As bonus, can you make a variation of this mission where the second (east) loiter location is chosen by the user at run time with a mouse click?

6.3 Assignment 5 (check off) - The Bravo UUV Mission

Note: The pMarinePID and pMarinePIDV22 apps may be used interchangebly in this lab. Same with uSimMarine and uSimMarineV22. The V22 version is newer and the other version will soon be deprecated. They should have the same config parameters however.

Our next step is to change the Bravo mission to simulate a UUV instead of a surface vehicle. We'll need to modify a few configurations for uSimMarineV22, pMarinePIDV22, and in pHelmIvP. And of course, we will also add some behavior components to our mission so the UUV may actually dive.

In this part we will:

- Understand how to augment pMarinePIDV22 when using UUVs
- Understand how to augment uSimMarineV22 when using UUVs
- Understand how to augment pHelmIvP when using UUVs
- Learn to use the ConstantDepth behavior
- Learn to tie behaviors together to operate in coordination.
- Learn how to use pMarineViewer when dealing with underwater vehicles.

The first step is to make modifications to a handful of MOOS apps use thus far in our examples, to support depth. These modifications are listed on the last page of this lab handout, and also on the course website under lab updates. The latter may be better for the purposes of cutting and pasting into your files. The third bravo mission should be configured with the following features:

- Make a copy of the previous bravo mission, making a new directory in the `missions/lab_05/` folder called `bravo_loiter_uuv`.
- Make the changes to pMarinePIDV22, uSimMarineV22, pHelmIvP and pNodeReporter as described in Section 7.
- Add a pair of ConstantDepth behaviors to the behavior file, and configure them such that the vehicle operates at 30 meters depth when loitering in the west, and 10 meters depth when loitering in the east.

It should look something like the video posted at:

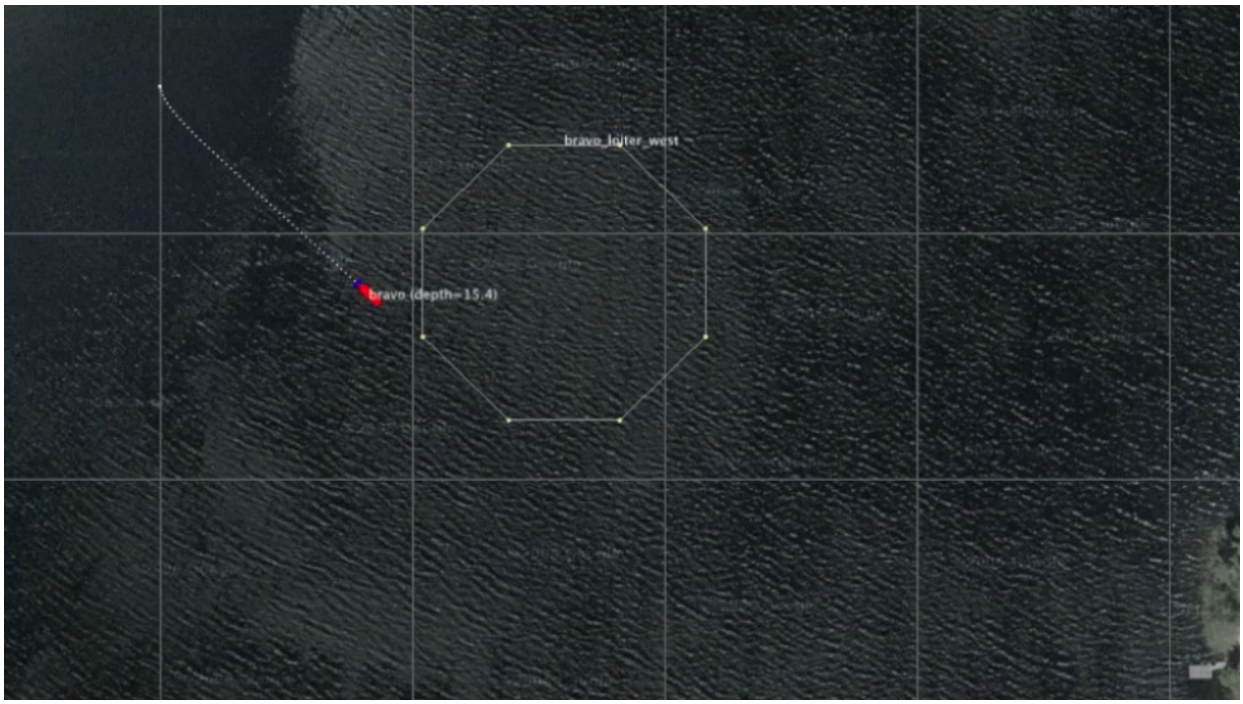

Figure 7: The modified bravo double-loiter mission now with a UUV operating at a depth of 30 meters.
video:(0:25): https://vimeo.com/87399032

Tips:

You may change the label rendered next to the vehicle in pMarineViewer by repeatedly hitting the 'n' key. It's helpful to have the depth of the vehicle rendered while operating. You can configure pMarineViewer to come up in this mode by setting:

```
vehicles_name_mode = names+depth
```

More information on the ConstantDepth behavior may be found in the helm documentation:

http://oceanai.mit.edu/ivpman/bhvs/ConstantDepth.

You can leave the `peakwidth`, `basewidth`, and `summitdelta` parameters unspecified for now, using their default values. **Note:** The ConstantDepth behavior *must* have its `duration` parameter set to a non-zero value, or set to `duration=no-time-limit`. You may want to read more on these parameters after the lab.

To tie the depth behaviors to the loiter behaviors, simply make their run conditions identical.

6.4 Assignment 6 (check off) - The Bravo UUV Surface Mission

This assignment involves a final modification to the Bravo example mission. In this mission, the bravo vehicle will periodically come to the surface, wait some number of seconds at the surface at zero speed, and then dive and resume its mission. Presumably this to simulate roughly what happens in a UUV that needs to occasionally come to the surface for a GPS fix to re-set its navigation solution. Your goals are:

- Make a copy of the previous bravo mission, creating a new directory `bravo_uuv_surface`.
- Make use of the helm Timer behavior to augment your mission to have the vehicle periodically stop and come to the surface (every 200 seconds). Make use of another Timer behavior that begins when the vehicle is at the surface, to wait 60 seconds before allowing the vehicle to dive again.

It should look something like the video posted at:

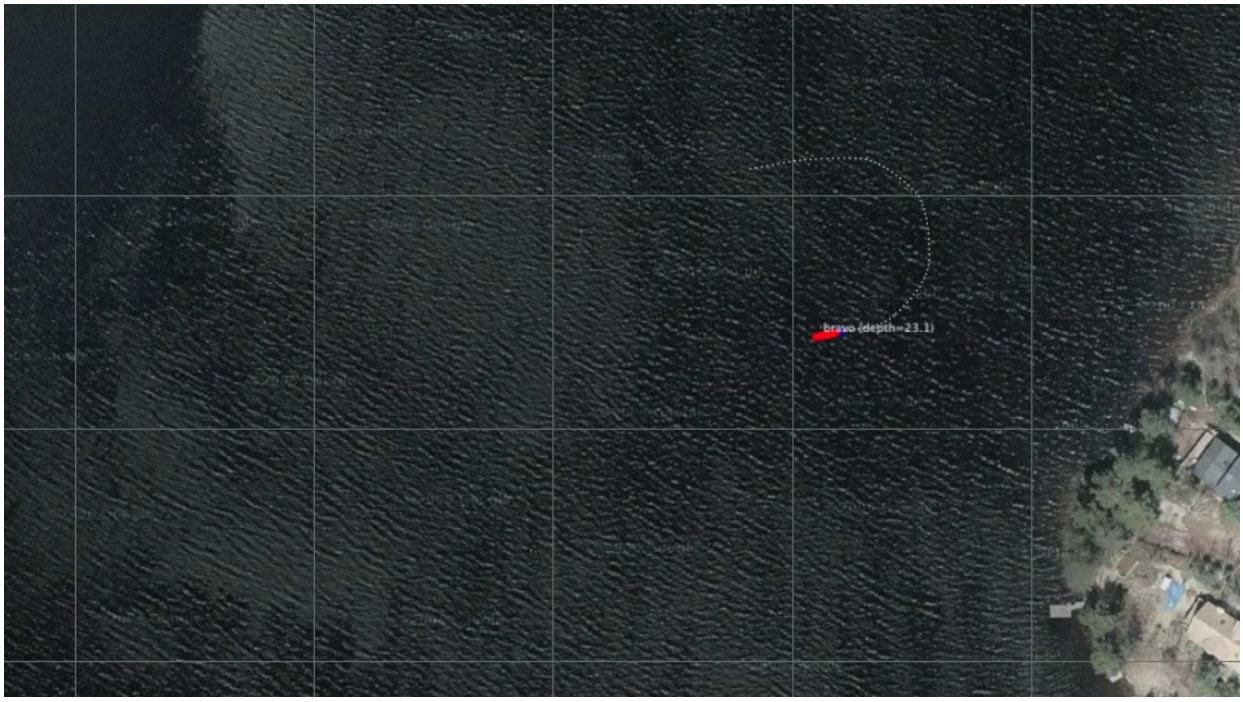

Figure 8: A UUV loiters between two regions and periodically comes to the surface and waits for a GPS fix before proceeding.
video:(0:45): https://vimeo.com/87422920

NOTE: There is an existing helm behavior called `BHV_PeriodicSurface` that can accomplish some of the functionality of this lab. Your goal is to complete the lab without using this behavior, but instead use a compbination of other simpler behaviors.

6.5 Assignment 7 (checkoff) Bravo UUV Fair Time Mission

You may noticed with your Assignment 6 mission that when the vehicle surfaces and waits for the prescribed time at the surface, when it finally does resume it usually heads off directly to the opposite loiter region. Take a look again at the video from Figure 8. Why is this? The loiter behavior that *was* active when the surfacing sequence began immediately became idle, and the duration clock for that loiter behavior never stopped during the surfacing sequence.

We would like the vehicle instead to resume its interrupted loiter for the time duration that remained when the loiter was interrupted. To do this, see the parameter defined on all behaviors called `duration_idle_decay`. See http://oceanai.mit.edu/ivpman/helm/param_duration

The new mission should look something like the video posted at:

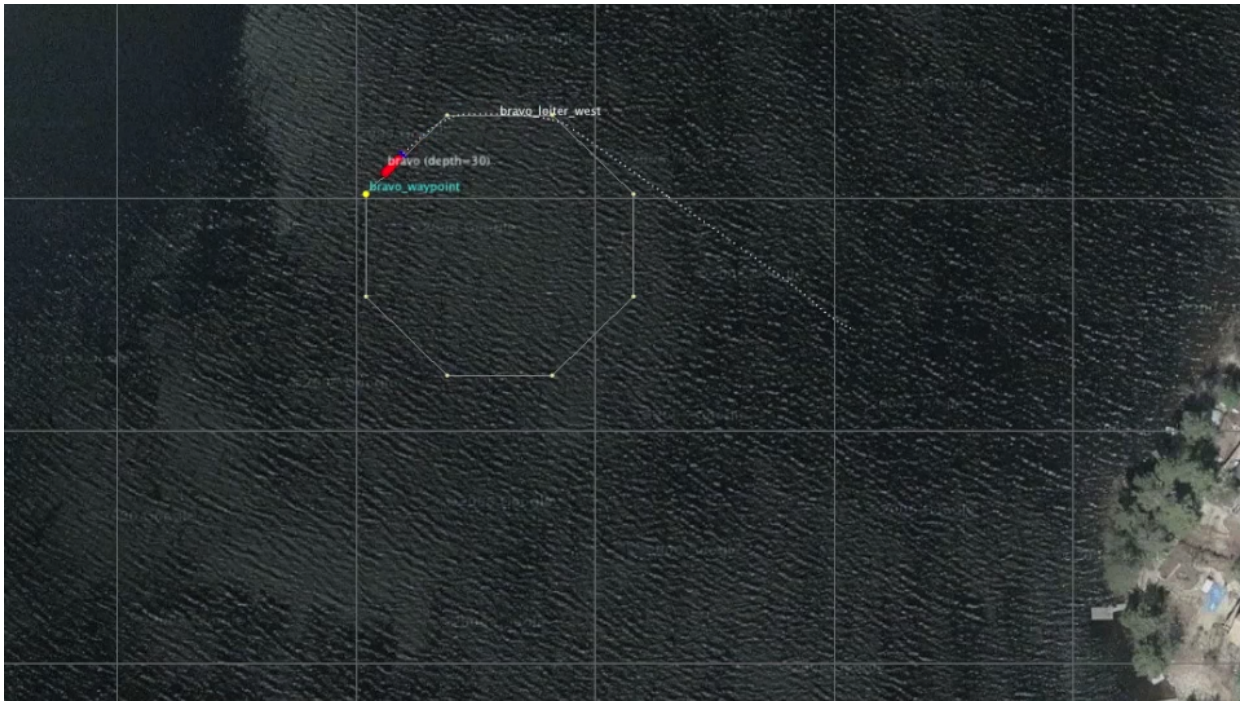

Figure 9: A UUV loiters between two regions and periodically comes to the surface and waits for a GPS fix before proceeding. The time spent during the surfacing sequences does not count in time spent spent at each loiter region. video:(1:10): https://vimeo.com/87425349

6.6 Assignment 8 (checkoff) Bravo UUV Odometry Misssion

In the final step, dust off your pOdometry app and modify it to accept a configuration parameter depth_thresh. This parameter will accept a depth, in meters. When enabled (the default value is zero), the odometry distance will only accumulated when the vehicle depth NAV_DEPTH is greater then depth_thresh.

You should modify the appcasting output of your Odometry behavior to display both the total odometry distance, and the distance meeting the depth threshold. The latter should be published to the MOOSDB as ODOMETRY_DIST_AT_DEPTH. In your mission, you can have a return behavior (a Waypoint behavior) configured to return home, with a condition based on the above MOOS variable.

Modify the double-loiter mission to have the vehicle return home after a set amount of distance, 200 meters, at a depth greater than 25 meters. As a BONUS, make the distance and depth configuration parameters of your `launch.sh` script. See how far you can get on this step on your own, but we will post some info/tips on this topic during the week.

7 Simulator Configurations for Operating at Depth

The below four modifications are needed for configuring your simulation to simulate a UUV, i.e., simulating depth in a vehicle.

Modifying the pMarinePIDV22 configuration

To have a simulated *underwater* vehicle, add the below lines to your pMarinePIDV22 configuration block in your mission file(s). Note that in the case of depth_control=true, you are probably *replacing* an existing line.

```
depth_control = true

//Pitch PID controller
pitch_pid_kp                   = 1.5
pitch_pid_kd                   = 0.3
pitch_pid_ki                   = 0.004
pitch_pid_integral_limit       = 0

//ZPID controller
z_to_pitch_pid_kp              = 0.12
z_to_pitch_pid_kd              = 0.1
z_to_pitch_pid_ki              = 0.004
z_to_pitch_pid_integral_limit  = 0.05

maxpitch     = 15
maxelevator  = 13
```

Modifying the uSimMarineV22 configuration

To have a simulated *underwater* vehicle, add the below lines to your uSimMarineV22 configuration block in your mission file(s).

```
buoyancy_rate        = 0.15
max_depth_rate       = 5
max_depth_rate_speed = 2.0
default_water_depth  = 400
```

Modifying the pHelmIvP configuration

The below augments the helm decision space to include 101 possible depth decisions. In deeper water, a different configuration may be used. This line needs to be added to the pHelmIvP configuration block in your mission file(s).

```
domain = depth:0:100:101
```

Modifying the pNodeReporter configuration

The modification below to pNodeReporter is mostly cosmetic. It changes the vehicle type to "UUV" so you see a UUV icon in your simulator diving, rather than a kayak. This line can be found in the pNodeReporter configuration block in your mission file(s).

```
platform_type = UUV
```

8 Instructions for Handing In Assignments

Missions should be added to your Git trees. In this lab, only the mission files are added - no source code.

8.1 Requested File Structure

Here is the requested file structure:

```
moos-ivp-extend/
  missions/
    lab_05/
      alpha_return/              // Assignment 1 - self check off
      alpha_return_now/          // Assignment 2 - check off
      bravo_loiter/              // Assignment 3 - self check off
      bravo_loiter_dbl/          // Assignment 4 - self check off
      bravo_loiter_uuv/          // Assignment 5 - check off
      bravo_uuv_surface/         // Assignment 6 - check off
      bravo_uuv_fair/            // Assignment 7 - check off
      bravo_uuv_odometry/        // Assignment 8 - check off
```

8.2 Due Date

This lab should have its check-off components checked off, and labs under version control by the end of lab on Thursday Feb 29th, 2024. Lab hours on Thu Feb 29th, will be dedicated to working on Lab 05. The beginning of lab Tue Mar 5th may be used to demonstrate and check off outstanding components, but the focus of the Mar 5th lab will be on Lab 06.