Lab #6
Introduction to Helm Autonomy

2.S998 Unmanned Marine Vehicle Autonomy, Sensing and Communications

Contents

1 Overview and Objectives **3**
- 1.1 Preliminaries . . . 3
- 1.2 More MOOS / MOOS-IvP Resources . . . 3
- 1.3 The Basic Helm Structure . . . 4
- 1.4 Putting the Helm into Drive: The High-Level Helm State . . . 5

2 Experimenting with and Modifying the Alpha Example Mission **6**
- 2.1 The basics of launching an autonomy mission . . . 6
- 2.2 Understanding the helm during mission execution . . . 7
- 2.3 Methods in pMarineViewer for understanding and controlling a mission . . . 9
- 2.4 Assignment . . . 10

3 Building Your First Autonomy Mission - The Bravo Mission **11**
- 3.1 Building the First Bravo Mission . . . 11
- 3.2 Add a Second Loiter Behavior to the Bravo Mission . . . 12
- 3.3 Add Depth to the Bravo Mission, Going from Kayak to UUV . . . 13
- 3.4 Assignment . . . 14

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.

1.1 Preliminaries

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/MOOS/MOOSBin/MOOSDB
$ which uTimerScript
/Users/you/moos-ivp/bin/uTimerScript
$ which mykill
/Users/you/moos-ivp/scripts/mykill
```

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 may 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-jsmith
$ ls
CMakeLists.txt  bin/            build.sh*       docs/           missions/       src/
README          build/          data/           lib/            scripts/
```

For each distinct exercise in this lab, there should be a corresponding subdirectory in the missions folder, typically with both a `.moos` and `.bhv` configuration file.

1.2 More MOOS / MOOS-IvP Resources

We will only just touch the MOOS basics today. 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.

  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

1.3 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.

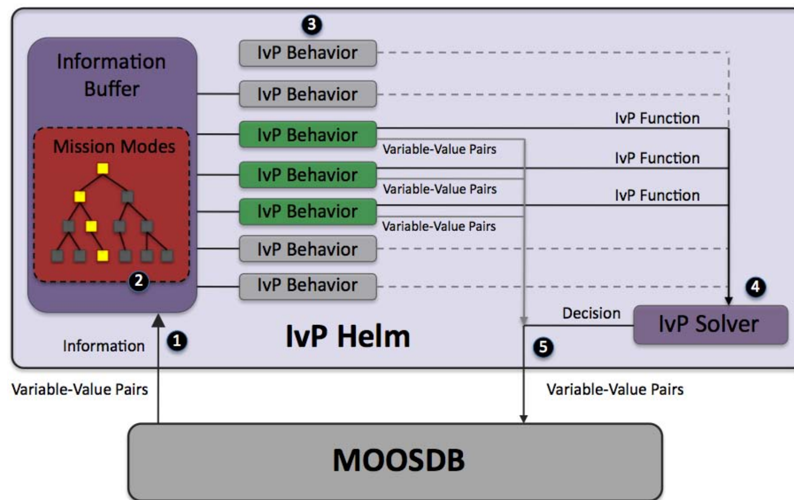

Figure 1: **The Helm Iterate Loop:** (1) Mail is read from the MOOSDB. It is parsed and stored in a local buffer to be available to the behaviors. (2) If there were any mode declarations in the mission behavior file, they are evaluated at this step. (3) Each behavior is queried for its contribution and 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. Of course behavior coordination takes place in the solver, and one 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.

1.4 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 whether it is in the `Park` or `Drive` state:

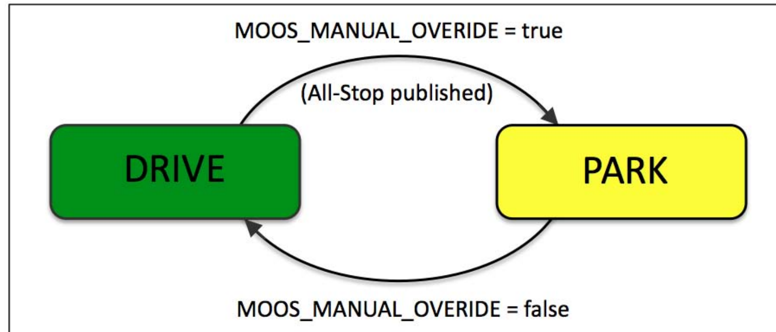

Figure 2: **The Helm State of the IvP Helm**: The helm state has a value of either `PARK` or `DRIVE`, depending on both how the helm is initialized and the mail received by the helm after start-up on the variable `MOOS_MANUAL_OVERRIDE`. The helm may also park itself if an all-stop event has been detected.

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.

2 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 build from the class SVN branch:

```
% cd moos-ivp
% svn update
Updated to revision 4194.
% ./build-ivp.sh
```

Assuming there were no problems with the update and build, we're ready to go.

2.1 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

2.1.1 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.

```
% cp -rp moos-ivp/ivp/missions/s1_alpha  moos-ivp-jsmith/missions/alpha
```

2.1.2 Note the MOOS and helm mission file structures

Note in 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:

```
0  ProcessConfig = pHelmIvP
1  {
2    AppTick    = 4
3    CommsTick  = 4
4
5    Behaviors  = alpha.bhv
6    Verbose    = quiet
7    Domain     = course:0:359:360
```

```
8    Domain      = speed:0:4:21
9  }
```

The two immediately important components of this are (a) the indication of the behavior file on line 5, 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.

2.1.3 Launch and experiment with MOOS time warp

The alpha mission is described in a fair amount of detail in the helm documentation, starting on p. 36:

http://oceanai.mit.edu/moos-ivp-pdf/moosivp-helm.pdf

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

```
% cd moos-ivp-jsmith/missions/alpha/
% pAntler alpha.moos
```

Things should look similar to the figure on page 37 of the helm documentation. 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 most of the time the use of a higher time warp is 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
```

2.2 Understanding the helm during mission execution

In this part we will:

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

2.2.1 Find the vehicle run state during mission execution

The helm is describable at its highest level by two MOOS variables, `IVPHELM_STATUS` and `IVPHELM_ALLSTOP`. These two issues are described in Sections 5.2 and 5.3 in the helm documentation:

http://oceanai.mit.edu/moos-ivp-pdf/moosivp-helm.pdf

Take a few minutes to read this through.

The run 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 run-state, and the second indicates the all-stop status. When the vehicle is deployed by hitting the deploy button, the run-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"`.

2.2.2 Understand the difference between the run-state and all-stop status

The helm run-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 run-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?

2.2.3 Understand the behavior condition and endflag parameters

Two key parameters defined for all behaviors are the `condition` and `endflag` parameters. They are described in Section 6.5 in the helm documentation. Note in the Alpha mission 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 to be `DEPLOY=false` and `RETURN=false` initially.

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.

2.3 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

2.3.1 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 pMarineViewer, but represents how this button was configured for specific use in this mission. The pMarineViewer tool is the primary tool used in this and later labs for rendering the mission as it unfolds and for interacting with the vehicle while deployed. 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  condition  = RETURN = true
```

The syntax and general usage is described in Section 13.5.2 in the helm documentation. In short, there are four configurable buttons, `BUTTON_ONE` through `BUTTON_FOUR`. 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.

2.3.2 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 'a' at any time, and entering the variable name in the dialog box.

2.3.3 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 Section 13.3.6 in the helm documentation. 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 '
' character at the end of the line. Keep it all on one line - a limitation of .moos files, for now at least.

(Try configuring your mission such that both types of events occur with the same mouse click).

2.4 Assignment

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.

- Extending the above example, configure your mission and pMarineViewer to have a second left-mouse-click context where. 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.

3 Building Your First Autonomy Mission - The Bravo Mission

In the second exercise in 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 construction a mission for a surface vehicle, but will migrate shortly to configuring for an underwater vehicle.

Much of information you will need regarding the workings of various behaviors and the helm are described in the helm documentation (https://oceanai.mit.edu/moos-ivp-pdfs/moosivp-helm.pdf). Several of the target missions we will building have been simulated prior to the lab with short videos. They can found on the 2.S998 course website

3.1 Building the First Bravo 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`. 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=0,y=-75`, a radius of 30 meters, and the loiter polygon should have 8 vertices. It should be set with a loiter speed of 1.3 m/sec.
- 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 a the return waypoint behavior.

See clip 03 linked at `https://oceanai.mit.edu/2.S998/clips.html` to get a visual of what the end mission should look like when running. Here are some other links and hints worth noting:

The Loiter behavior is described in Section 8.3 in the helm documentation. The primary parameters of interest here are the `polygon` and `speed` parameters. Use the `format=radial` format for describing the loiter polygon as described on p. 110.

The waypoint return behavior is configured similarly to the alpha mission, but more information on this behavior may be found in Section 8.1 of the helm document.

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 Section 7.2.3 in the helm document.

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 in Sections 6.5.3 and 6.5.4.

3.2 Add a Second Loiter Behavior to the Bravo 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 to use the uTimerScript fast-forward feature.
- Learn about the special but versatile IvPHelm Timer behavior.

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 first bravo mission, creating files `bravo2.moos` and `bravo2.bhv`. Make sure to edit the pHelmIvP config block in the `.moos` file to point to `bravo2.moos`.
- The new bravo mission should have a second loiter behavior, which is not active upon an initial deploy. It should have a location `x=120,y=-75`, a radius of 20 meters, and the loiter polygon should have 8 vertices. It should be set with a loiter speed of 1.3 m/sec.
- Make the two loiter behaviors mutually exclusive using their `condition` parameters, e.g., `condition=LOITER_REGION=west`.
- Configure a uTimerScript script to periodically switch the vehicle between the two loiter regions, once every 150 seconds.

See clip 04 linked at https://oceanai.mit.edu/2.S998/clips.html to get a visual of what the end mission should look like when running.

As a follow-on step, further modify the mission such that:

- Utilize the uTimerScript `UTS_FORWARD` fast-forwarding feature, described in Section 14.3.3 in the helm documentation, to allow the pMarineViewer operator to push a button to send the vehicle to the other loiter region before the script would otherwise do so.
- Configure one of the command buttons in pMarineViewer to accept the above poke to `UTS_FORWARD`, to fast-forward the uTimerScript script.

As a last variation of this mission, further modify the mission such that:

- Utilize the `duration`, `endflag`, and `perpetual` parameters to accomplish the periodic switching between loiter regions otherwise implemented with uTimerScript as above.
- 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.

3.3 Add Depth to the Bravo Mission, Going from Kayak to UUV

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 in the uSimMarine, pMarinePID, 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 pMarinePID when using UUVs
- Understand how to augment uSimMarine 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 second bravo mission, creating files `bravo3.moos` and `bravo3.bhv`. Make sure to edit the pHelmIvP config block in the `.moos` file to point to `bravo3.moos`.
- Make the changes to pMarinePID, uSimMarine, pHelmIvP and pNodeReporter as described on the last page of this lab handout.
- 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.

See clip 05 linked at https://oceanai.mit.edu/2.S998/clips.html to get a visual of what the end mission should look like when running. Here are some other things worth noting:

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.

More information on the ConstantDepth behavior may be found in the helm documentation. You can leave the `peakwidth`, `basewidth`, and `summitdelta` parameters unspecified for now, using their default values. 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.

3.4 Assignment

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 files `bravo4.moos` and `bravo4.bhv`. Make sure to edit the pHelmIvP config block in the `.moos` file to point to `bravo4.moos`.
- Make use of the helm Timer behavior to augment your mission to have the vehicle periodically 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.
- The time surfacing and at the surface should "count" in the time spend in loitering in the west and east regions.

See `clip 06` linked from https://oceanai.mit.edu/2.S998/clips.html.

Simulator Configurations for Operating with depth

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

See https://oceanai.mit.edu/2.S998/updates.html for more info on the below changes and to use for cut-paste.

Modifying the pMarinePID configuration

```
DEPTH_CONTROL = true

//Pitch PID controller
PITCH_PID_KP                  = 1.5
PITCH_PID_KD                  = 1.0
PITCH_PID_KI                  = 0
PITCH_PID_INTEGRAL_LIMIT      = 0

//ZPID controller
Z_TO_PITCH_PID_KP             = 0.12
Z_TO_PITCH_PID_KD             = 0
Z_TO_PITCH_PID_KI             = 0.004
Z_TO_PITCH_PID_INTEGRAL_LIMIT  = 0.05

MAXPITCH     = 15
MAXELEVATOR  = 13
```

Modifying the uSimMarine configuration

```
BUOYANCY_RATE        = 0.025
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.

```
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.

```
VESSEL_TYPE   = UUV
```

MIT OpenCourseWare
http://ocw.mit.edu

2.S998 Marine Autonomy, Sensing and Communications
Spring 2012

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.