Lab 4 - Introduction to MOOS Programming

2.680 Unmanned Marine Vehicle Autonomy, Sensing and Communications

February 22nd, 2024

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

1 Overview and Objectives **3**
- 1.1 Preliminaries . . . 3
- 1.2 MOOS, MOOS-IvP and Your Applications . . . 3
- 1.3 More MOOS / MOOS-IvP Resources . . . 4
- 1.4 The MOOS Application Structure . . . 4
- 1.5 Handy Functions Defined on MOOSMsg (MOOS mail) . . . 5
- 1.6 Assignment 1 (self check off) . . . 7

2 Getting Started with Version Control **7**

3 Build Your First MOOS App - An Odometry MOOS App **8**
- 3.1 Generating a Template App and Augmenting the Build System . . . 8
- 3.2 Assignment 2: (self check off) Verify Skeleton pOdometry has Built . . . 9
- 3.3 Writing the Odometry MOOS App . . . 9
- 3.4 Use Your pOdometry app in the Alder mission . . . 10
- 3.5 Assignment 3 (check off) Verify Your pOdometry Works . . . 12
- 3.6 Assignment 4 (check off) Convert pOdometry to an AppCasting MOOSApp . . . 13

4 Some Hints for Debugging **14**
- 4.1 Run the Logger . . . 14
- 4.2 Debugging with Additional MOOS Publications . . . 14
- 4.3 Debugging with Terminal Output, Method 1 . . . 15
- 4.4 Debugging with Terminal Output, Method 2 (Recommended) . . . 15

5 Additional Exercises **16**
- 5.1 Adding Terminal Output to your App for Debugging . . . 16
- 5.2 Adding Run Time Checks for Steady Nav Information . . . 16
- 5.3 Bonus: Configure the Staleness Threshold . . . 16
- 5.4 Bonus: Configurable Units . . . 17

6 Due Date and Grading Criteria **18**
- 6.1 Due Date . . . 18
- 6.2 Grading Criteria . . . 18
- 6.3 Submitting Instructions for Lab 04 . . . 18

1 Overview and Objectives

In this lab we will produce our own MOOS applications. We begin by downloading an example application complete with it's own build structure. We proceed by making our first simple MOOS app emphasizing the usage of the basic MOOS components. This is followed by a couple more complex application exercises.

This lab contains our first hand-in assignment. Although later in the class we will be working with partners, in this lab, all work should be done by individuals. When students post questions of general relevance, we will make that information available to everyone.

- The MOOS Application Structure (Iterate, OnNewMail, OnStartUp Methods)
- The MOOS Message Structure
- Getting Started with Software Version Control
- Downloading and Building the moos-ivp-extend tree
- Build Your First MOOS App - An Odometry App

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

If unsuccessful with the above, return to the steps in Lab 1:

https://oceanai.mit.edu/ivpman/labs/machine_setup

1.2 MOOS, MOOS-IvP and Your Applications

In the previous lab we discussed the relationship between MOOS and MOOS-IvP. The MOOS tree is a body of software distributed as part of the MOOS-IvP tree as depicted in Figure 1. MOOS-IvP provides additional MOOS applications, including the IvP Helm behavior-based architecture, and has C++ build dependencies on the MOOS libraries. Today the focus is on building additional MOOS applications. Your MOOS apps will have a build dependency on the MOOS libraries, and you may choose to utilize libraries in the MOOS-IvP tree. We start by downloading the `moos-ivp-extend` tree which may be regarded as a template for extending the MOOS-IvP tree with apps and behaviors. This tree also includes a functioning build structure to ease the learning curve on C++ build issues for now. You likely have already downloaded and built this tree if you finished the last part of the previous lab.

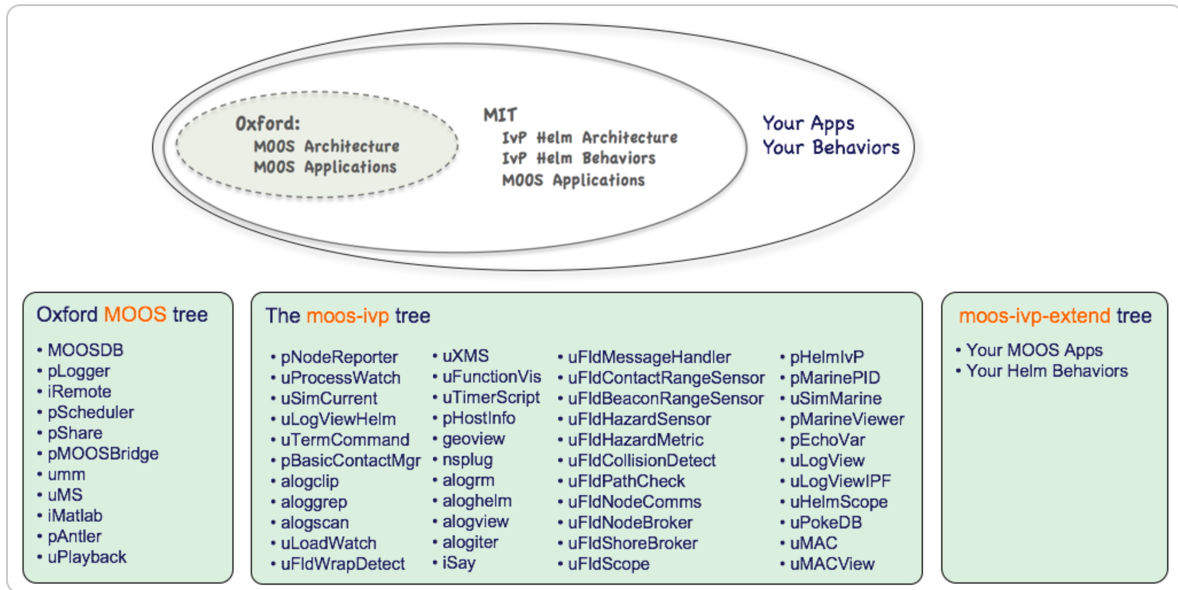

Figure 1: **Nested Repositories:** The MOOS-IvP tree contains the Oxford MOOS tree and additional modules from MIT including the Helm architecture, Helm behaviors and further MOOS applications. The set of modules may be expanded with user third-party applications or behaviors.

1.3 More MOOS / MOOS-IvP Resources

We will only just touch the MOOS programming 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 MOOS and MOOS-IvP related to marine robotics.
  `https://oceanai.mit.edu/2.680/docs/2.680-04-moos_programming_2024.pdf`
- The Programming with MOOS documentation.
  `https://oceanai.mit.edu/2.680/docs/ProgrammingWithMOOS.pdf`
- The moos-ivp.org website documentation.
  `http://oceanai.mit.edu/ivpman`

1.4 The MOOS Application Structure

The main idea explored today is the notion and structure of a MOOS application. We know from the last lab that MOOS apps publish, subscribe for, and handle mail passed from one application to another through the `MOOSDB`. In the last lab we worked with existing MOOS apps, only modifying their functionality through configuration options provided by the application author. Even without modifying the code of existing MOOS apps there are many ways to configure a system for unique domains. However, the real power of MOOS comes from the fact that no application is sacred. If you don't like what it does (or doesn't) do, you are free copy *and rename it*, and modify the code to your satisfaction. Or you can just build your own application from scratch. This is the focus of today's lab.

The key components to keep in mind in today's lab are shown in Figure 2 below. All MOOS apps begin by being a subclass of the MOOSApp superclass defined in the Oxford MOOS library. The primary work of the app developer is in writing the three functions shown in the figure.

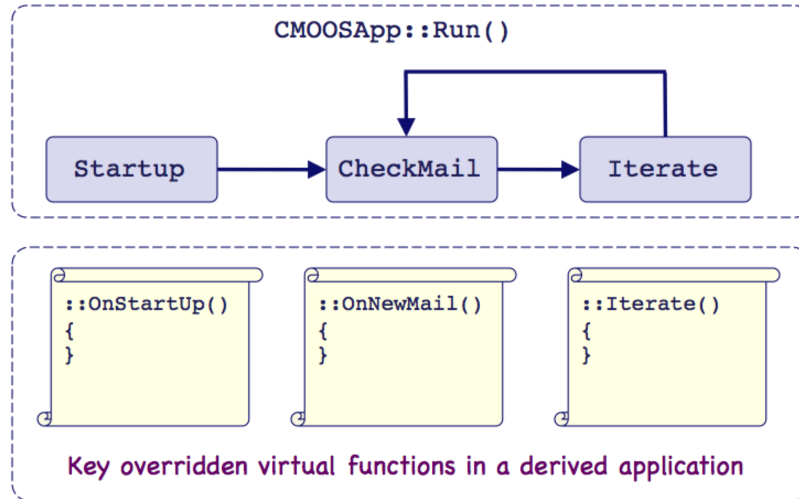

Figure 2: **The MOOSApp Key Functions:** All MOOS apps are a subclass of the MOOSApp superclass. Development mostly boils down to overriding the three functions below with the particulars of one's own liking.

Documentation Conventions

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

- 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`. Except for `AppTick` and `CommsTick`, configuration parameters are generally lower case.
- MOOS apps and executables generally, are rendered in magenta as with `MOOSDB` and `GenMOOSApp`.
- IvP Helm Behavior configuration parameters are rendered in `brown`, such as `priority=100` and `endflag=RETURN=true`.

This convention closely follows the convention used in the helm documentation.

1.5 Handy Functions Defined on MOOSMsg (MOOS mail)

The below methods are defined on an instance of `MOOSMsg`. Typically we are dealing with MOOS messages in the `OnNewMail()` loop of an application. Besides knowing the name of the variable, you may also want to know (a) what is its value, (b) by whom or which app was it posted, (c) when was it posted, (d) is its value of type string or double, and so on.

In our `pXRelayTest` example app, we only call `msg.GetKey()` in `Relayer.cpp`, but all the below methods are available for getting the message fields. (See today's lecture notes for a description of a

MOOS Message.) This information is probably in the MOOS documentation, but is repeated here for convenience.

```
// return the name of the message
std::string GetKey();

// return the name of the message (just another way)
std::string GetName();

// check data type is double
bool IsDouble();

// check data type is string
bool IsString();

// return time stamp of message
double GetTime();

// return double val of message
double GetDouble();

// return string value of message
std::string GetString();

// return the name of the process (as registered with the DB) which
posted this notification
std::string GetSource();

// return the name of the MOOS community in which the orginator lives
std::string GetCommunity();
```

Later on in this lab you will see that we provide a template for generating the skeleton code of a new MOOS app. This is done using the `GenMOOSApp` script. This skeleton code will generate the following `OnNewMail()` method:

```
bool MyApp::OnNewMail(MOOSMSG_LIST &NewMail)
{
  MOOSMSG_LIST::iterator p;
  for(p=NewMail.begin(); p!=NewMail.end(); p++) {
    CMOOSMsg &msg = *p;

#if 0 // Keep these around just for template
    string key   = msg.GetKey();
    string comm  = msg.GetCommunity();
    double dval  = msg.GetDouble();
    string sval  = msg.GetString();
    string msrc  = msg.GetSource();
    double mtime = msg.GetTime();
    bool   mdbl  = msg.IsDouble();
    bool   mstr  = msg.IsString();
#endif
   }
   return(true);
}
```

Note all the typical function calls you may want to make on an incoming MOOS message are right there, commented out to start with, but ready to use if you need them in your application. (Everything between the `if 0` and `endif` is ignored by the compiler, so these lines have the same effect as comment lines.)

1.6 Assignment 1 (self check off)

Verify that you have successfully downloaded and built the `moos-ivp-extend` tree, and that the binaries of this tree are findable in your shell path. If you haven't done this yet, see Section 2 for info on how to obtain the `moos-ivp-extend` tree from `git`.

Verify with the following:

```
$ which pXRelayTest
$ /home/you/moos-ivp-you/bin/pXRelayTest
```

If the `which` function returns no value, then either the binary `pXRelayTest` was not built or you haven't successfully augmented your shell path. First check to ensure it was built by examining the contents of `moos-ivp-you/bin/`. If `pXRelayTest` is there, then double check that you have properly augmented your shell path. If need be, return to the help page:

http://oceanai.mit.edu/ivpman/help/cmdline_augment_shell_path/

You can also display your shell's current path with:

```
$ echo $PATH
```

If you don't see `/home/you/moos-ivp-ivp/bin` in the colon-separated list of locations constituting your shell path, then you are not done.

2 Getting Started with Version Control

In Lab 3, a supplement document distributed for getting started with `git`, and obtaining a copy of your `moos-ivp-extend` tree. This document is online here:

https://oceanai.mit.edu/ivpman/labs/git

At this stage you should have few steps under your belt:

- Created a gitlab account
- Obtained a copy of the `moos-ivp-extend` tree and properly renamed it as your own
- Provided access to the TAs
- Confirm you know how to modify a file in your tree and commit the changes
- Confirm you can add a file or directory to your tree and commit changes.

If the above steps are not complete, you should make sure they are complete before moving on to

the next section. If you are proficient in some other version control environment, or if you are more comfortable using GitHub, this should be fine, but please let us know.

3 Build Your First MOOS App - An Odometry MOOS App

The Focus of this section is on building your first MOOS App. Your goals are:

1. Learn how to generate a MOOS application from scratch using a template-generating script.
2. Learn how to add the new MOOS application to the build system.
3. Write your new MOOS app, pOdometry, to calculate the total distance traveled by the vehicle.
4. Test your MOOS app by using it in the Alder example mission to have the vehicle return after traveling 50 meters.
5. Convert your MOOS app into an AppCasting MOOS App.

3.1 Generating a Template App and Augmenting the Build System

Your goal in this part is to generate a new application by using a shell script to generate a full application template. The resulting template should be buildable immediately without further modification, but will essentially do nothing meaningful until you add your own functionality.

3.1.1 Generate the MOOS App

To generate the new MOOS app, in a terminal window change directories to `moos-ivp-extend/src/`. From here you should be able to invoke the below command:

```
$ GenMOOSApp_AppCasting Odometry p "Jane Doe"
```

The above script, GenMOOSApp_AppCasting, should be in your shell path under `moos-ivp/scripts/`. You may need add this directory to your shell path if you haven't done so already. The above script invocation builds a new app called pOdometry. The third argument is your name. You can type GenMOOSApp_AppCasting -h in the future to remind yourself.

3.1.2 Add Your Name

Always, always, always put your name at the top of your source code files! This is a good practice in general, but even more so in a class setting, or an open source environment where people are sharing code. It wouldn't hurt to add the date and organization information too.

3.1.3 Add Your New Application to the Build System

Add your new app to the build system. Edit the file `moos-ivp-extend/src/CMakeLists.txt` and look for the line referring to pXRelayTest. Copy that as an example for your new app.

```
#==========================================================================
# List the subdirectories to build...
#==========================================================================
ADD_SUBDIRECTORY(lib_behaviors-test)
ADD_SUBDIRECTORY(pXRelayTest)
ADD_SUBDIRECTORY(pExampleApp)
ADD_SUBDIRECTORY(pOdometry)                    <-- Add your line here
```

3.2 Assignment 2: (self check off) Verify Skeleton pOdometry has Built

Go back and re-run your build script. Check to see that pOdometry has been built, and is in your bin directory (e.g., moos-ivp-you/bin).

Verify that pOdometry is in your shell path with:

```
$ which pOdometry
$ /home/you/moos-ivp-you/bin/pOdometry
```

If the which function returns no value, then either the binary pXRelayTest was not built or you haven't successfully augmented your shell path. See above in Section 1.6 for tips on how to remedy the problem.

3.3 Writing the Odometry MOOS App

The goal of this step is to build a new simple MOOS application that calculates vehicle odometry. It will:

1. Register for the NAV_X and NAV_Y position of the vehicle.
2. Repeatedly read in the NAV_X and NAV_Y position of the vehicle.
3. Update the total distance traveled by calculating the distance between the present position and the previous position.
4. Post the odometry information in the MOOS variable ODOMETRY_DIST.
5. The posted odometry distance should be consistent regardless of the AppTick of the pOdometry app. You can check this by changing the AppTick of your app while leaving the AppTick of uSimMarineV22 unchanged.

We will do this in your newly checked out moos-ivp-extend tree. Begin as follows:

- Edit Odometry.h to have at least the six member variables:

```
bool   m_first_reading;
double m_current_x;
double m_current_y;
double m_previous_x;
double m_previous_y;
double m_total_distance;
```

You may of course use further member variables if you see the need.

- Edit Odometry.cpp to initialize the above member variables, in the class constructor.
- Edit Odometry.cpp to register for NAV_X and NAV_Y.
- Edit the Odometry::OnNewMail() method to handle the updated navigation position.
- Edit the Odometry::Iterate() method to calculate the new distance. Handle the special case of the first navigation measurement. Post the total distance to the MOOS variable ODOMETRY_DIST.
- Verify that everything builds...

There is one common "gotcha" in this lab. Keep in mind that the NAV_X and the NAV_Y variables are being published by the uSimMarineV22 application. This application is running at 10Hz and producing on average 10 incoming mail messages for your pOdometry app. If your pOdometry app is running at the default rate of 4Hz, then on average your app will have 2-3 incoming mail messages on each iteration. If you find that your odometry calculations only reflect about half of the actual distance traveled, reconsider how you are calculating the length of new legs.

3.4 Use Your pOdometry app in the Alder mission

The first goal is to just make sure it runs and posts the correct information. The second goal is to use the information to alter the mission behavior.

NOTE: Depending on when you cloned the moos-ivp-extend tree, you may need to ensure that all references to uSimMarine are replaced with uSimMarineV22 and all references to pMarinePID are replaced with pMarinePIDV22, in the alder.moos file.

3.4.1 Running the Un-modified Alder Mission

First make sure that you can run the Alder mission. Try it:

```
$ cd moos-ivp-extend/missions/alder
$ pAntler --MOOSTimeWarp=10 alder.moos
```

It should look something like the video posted at:

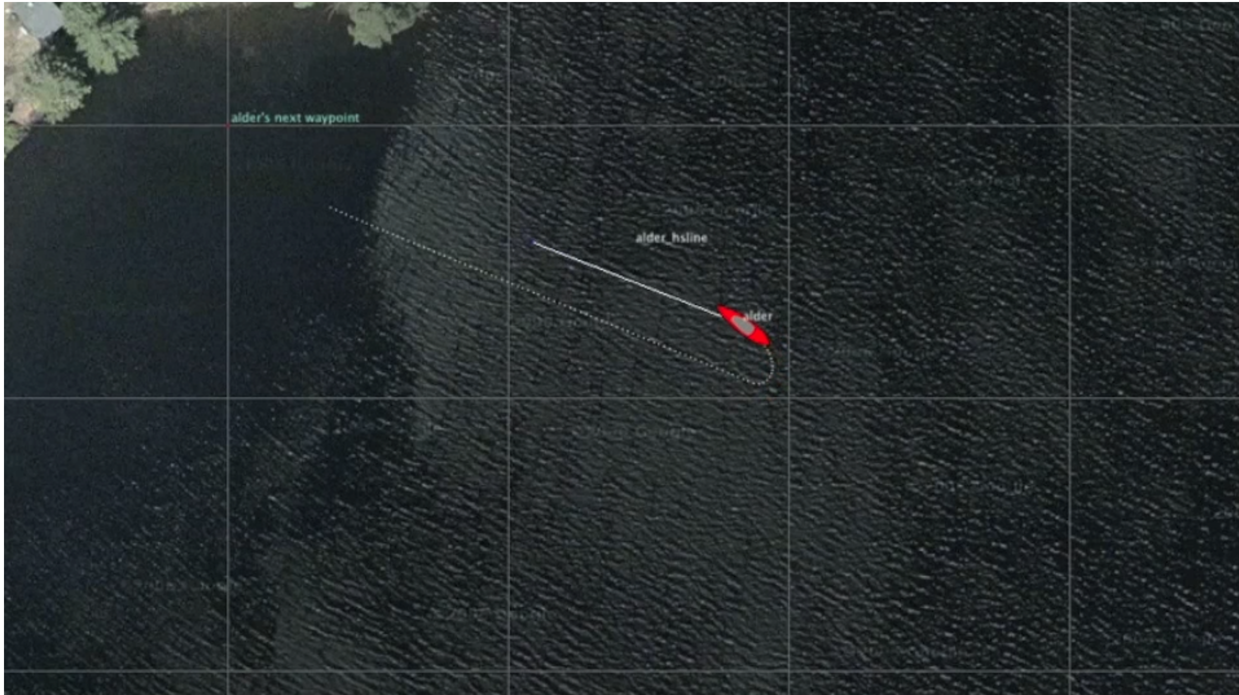

Figure 3: The Alder mission.

video:(0:13): https://vimeo.com/80730664

3.4.2 Run the Alder Mission with pOdometry

Next we'll add the pOdometry application to the mission and just verify it is generating the right output. Do the following steps:

- Modify the Alder mission (`moos-ivp-extend/missions/alder/alder.moos`) to include pOdometry at launch time by editing the Antler block in the alder.moos file.
- Add ODOMETRY_DIST to the scope list in pMarineViewer. See the pMarineViewer documentation to see how to do this. It should look something like:

```
ProcessConfig = pMarineViewer
{
  AppTick    = 4
  CommsTick  = 4

  ...
  scope = ODOMETRY_DIST    // <-- Add this line
  ...
}
```

- Re-launch the mission. Now you should be able to see the odometry distance in the scope field at the bottom of pMarineViewer. Does it look correct?

3.4.3 Modify the Alder Mission Using pOdometry

In the next step, we will modify the alder mission as follows. Instead of transiting to the waypoint specified in the waypoint behavior, the vehicle will return home after it has transited 50 meters. We haven't covered the Helm yet in class, so some of this we'll take on faith for now.

Take a look inside `alder.bhv`. This is where the helm is configured. The mission is comprised of two instances of a waypoint behavior. The first transits to a given point and is called `waypt_to_point`. It has two conditions, we'll add a third:

```
//----------------------------------------------
Behavior = BHV_SimpleWaypoint
{
  name      = waypt_to_point
  pwt       = 100
  condition = RETURN = false
  condition = DEPLOY = true
  condition = (ODOMETRY_DIST < 50)         // <-- Add this line

  endflag   = RETURN = true

  speed      = 2.0   // meters per second
  radius     = 8.0
  ptx        = 100
  pty        = -50
}
```

The second waypoint behavior is designed to return after the first one completes. It also has two conditions:

```
 condition = RETURN = true
 condition = DEPLOY = true
```

We'll change that to:

```
condition = (RETURN = true) or (ODOMETRY_DIST >= 50)
condition = DEPLOY = true
```

3.5 Assignment 3 (check off) Verify Your pOdometry Works

After the above modifications, re-launch the mission. Demonstrate to one of the TAs that your `pOdometry` works. It should look something like:

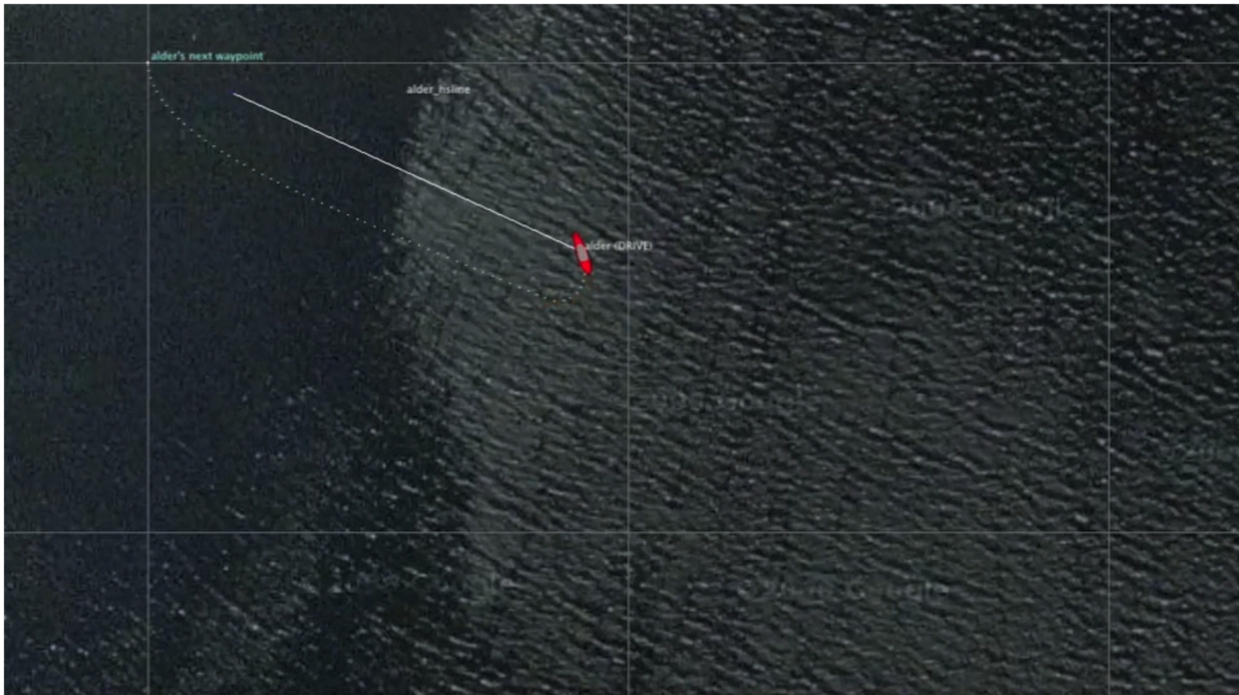

Figure 4: The Alder mission running with the pOdometry app.
video:(0:06): https://vimeo.com/80755512

3.6 Assignment 4 (check off) Convert pOdometry to an AppCasting MOOSApp

The last step will be to convert your pOdometry application to be appcast enabled. Follow the instructions from today's lecture notes, or read the documentation for enabling appcasting:

http://oceanai.mit.edu/ivpman/appcast_enable

If you used the GenMOOSApp_AppCasting script to make your initial folder for pOdometry, then your app is already mostly AppCast enabled. You just need to implement the AppCasting output. The output of your appcast report (in the buildReport() function) should simply show the total distance traveled.

4 Some Hints for Debugging

There are a few common methods for debugging a MOOS App, and we'll have more to say about this during the semester. But here are a few methods. The third method described below is a new feature added only in the last six months and is probably you most powerful option.

4.1 Run the Logger

If your Alder mission is not yet including pLogger, you will need to add it to your mission for these steps.

To add pLogger to your mission, add it the Antler config block with the additional line:

```
  Run = pLogger @ NewConsole = false
```

You will also need a configuration block for pLogger in the same mission file, e.g., alder.moos. The below block should work fine:

```
ProcessConfig = pLogger
{
  AppTick   = 8
  CommsTick = 8

  AsyncLog = true
  WildCardLogging = true
}
```

4.2 Debugging with Additional MOOS Publications

In this lab, the primary output of your app is the variable ODOMETRY_DIST. You are free to publish as many other variables as you like, to perhaps reveal any intermediate calculations, or other information. For example you can publish LEG_DIST if you are calculating the distance one leg at a time.

To utilize this method, you will need to either (a) scope on this additional variable while the mission is running, or (b) examine the variable from the log file after the mission is complete. The latter method gives you more time to ponder the values. To examine the log file, you can use aloggrep, along the lines of:

```
$ aloggrep logfile.alog ODOMETRY_DIST LEG_DIST
```

Or you can open alogview and scope on the above two variables using the VarHist pull-down menu. See the documentation for alogview. Either point your browser to:

https://oceanai.mit.edu/ivpman/apps/alogview

Or, on the command line, just type the below, and your browser should take you there.

```
$ alogview --web
```

4.3 Debugging with Terminal Output, Method 1

Another powerful method for debugging is to add cout statements to your code, to show intermediate values. So, for example, instead of publishing LEG_DIST as a MOOS variable, you can instead add something like this to your code:

```
double leg_dist = ...
cout << "Leg dist: " << leg_dist << endl;
```

To see this output, you would need to launch your app with a terminal window open, OR remove your app from the Antler block and run it separately from another terminal window. So in the first terminal window:

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

In the second terminal window:

```
$ pOdometry file.moos
```

Make sure you use the same .moos file in both launches. And note that if you launch your mission with a time warp, Antler will generate a file.moos++ mission file with the proper time warp. You would then need to launch your app with the same .moos++ file to get the same time warp.

4.4 Debugging with Terminal Output, Method 2 (Recommended)

If you have successfully created your app as an AppCasting MOOS App, then there is another simpler option for debugging with terminal output.

As above, inject your code with cout statements to produce what you would like. But rather than running your app in a separate Terminal window, simply add the below line to the configuration block for your app:

```
app_logging = log
```

This will log all the terminal output from your app in the alog file. Then you can view your terminal output by launching alogview. Once this is launched, select AppLog from the pull-down menu, select the vehicle name, and select your app from the list. You should then be able to see all your terminal output.

If you are trying this debugging method and have issues, please come find me. This is a new feature and not yet documented in the alogview documentation.

5 Additional Exercises

5.1 Adding Terminal Output to your App for Debugging

At some point you will need to debug a faulty app. A powerful tool for doing this is to generate terminal output (`cout` commands). In this exercise we will generate some simple output and show how we can view this output *after* the mission has completed using the `alogview` tool.

- Make sure your app is AppCast enabled if it is not already.
- Add terminal output in your app to show each new `NAV_X` and `NAV_Y` received, e.g.,
  `cout << "nav_x:  " << value << endl;`
- Add terminal output in your app to show each new value of `ODOMETRY_DIST` calculated.
- Enable app logging by setting `app_logging=log` in the `pOdometry` configuration block of your misssion file.
- Run your mission to completion.
- Find the alog file and launch `alogview` on this alog file.
- Select `AppLog` from the pull-down menu and confirm that you can see your terminal output.
- Show a TA.

Note that some of this functionality could also be achieved by generating AppCasting output. This is not a replacement for that. One advantage of AppLogging is that terminal output can be captured regardless of the level of code being executed. The `cout` call may be used in code that is called by your app several levels down.

5.2 Adding Run Time Checks for Steady Nav Information

Another powerful tool for developing robust apps is the use of run time err checking. AppCasting MOOS Apps support this with the `reportRunWarning()` function. In this exercise:

- Add a timestamp (a new member variable) to note when the latest `NAV_X` or `NAV_Y` was received.
- If an interval of 10 seconds or more passes, without receiving new NAV info, generate a run warning.
- Retract this warning when or if the NAV info resumes.
- To test this, run the Alder mission as normal. In another terminal window, type `killall uSimMarineV22`. This will halt the NAV info. Verify the run warning appears in your Odometry app.
- Then, also in a separate terminal window in your Alder mission, restart the simulator with `uSimMarineV22 alder.moos`.
- Show a TA.

5.3 Bonus: Configure the Staleness Threshold

To gain some experience with configuration parameters:

- Add a configurtion parameter to set the staleness threshold to something else besides the above default of 10 seconds.

- You will need to make this value a variable in your MOOS App instead of the the hard-coded `"10"` as before.
- Perform a check on the configuration parameter that it is a number greater than zero. If a user provides a faulty number in the mission file, produce a configuration warning.
- Show a TA.

5.4 Bonus: Configurable Units

Parameters that may be configured at *launch-time* are sometimes also configurable at *run-time*. There are many reasons for this. The two most-common reasons are (1) a parameter setting *augments* a set of parameters given at launch-time, with additional components that become known at run-time. (2) a parameter setting may involve the verbosity of output, perhaps in toggling on/off the publication of a MOOS variable providing debugging output. The need for debugging verbosity may change during run-time.

In this last bonus exercise, the task is to augment the `pOdometry` app with the ability to configure an additional publication of odometry distance in any arbitrary units provided either at configuration time or at run-time.

6 Due Date and Grading Criteria

6.1 Due Date

All MOOS apps described in this lab are due during lab Tuesday February 27th, 2024. Apps will be tested during lab, and source code is due to the graders at the end of the lab.

6.2 Grading Criteria

Grading will be based on

1. Implementation Credit (60%)
2. Code Organization (40%)

Full credit will given be based on whether or not you have met the given specs.

Code organization includes the issue of commenting your code, and choosing a robust code structure, e.g., properly initializing variables etc. Beginners to C++ programming can expect a healthy amount of constructive criticism in the first assignments. See the 9th and 10th C++ labs on the website: "C++ Coding Guidelines" and "C++ Coding Style Guidelines".

Unfinished assignments will be tested and accepted in the next lab. A 5% deduction will be applied for each lab that goes by unfinished.

6.3 Submitting Instructions for Lab 04

Lab 04 assignments should be submitted using the version control mechanism described earlier. Your directory format should have the naming and file organization as indicated below:

```
moos-ivp-yourname/
   src/
     pOdometry

   missions/
     alder/
       alder_lab4.moos
       alder_lab4.bhv

   lib/                (should be empty)
   bin/                (should be empty)
   build/              (should be empty)
   build.sh            (should be unchanged)
   CMakeLists.txt      (should be unchanged)
)
```