Lab 4 - Surface navigation using GPS

2.S01 Introduction to Autonomous Underwater Vehicles

Spring 2026

Supun Randeni, supun@mit.edu
Department of Mechanical Engineering
MIT, Cambridge MA 02139

1 Overview and Objectives **3**

2 What is a navigation solution **3**

3 Running a mission that logs GPS and AHRS navigation data **3**
3.1 Exploring StackUxV's software configuration and launch utility (`missions-StackUxV`) . . . 3
3.2 Defining the vehicle architecture . . . 8
3.3 Defining the vehicle . . . 9
3.4 Defining the cruise . . . 10
3.5 Cleaning old vehicle log files . . . 11
3.6 Launching the mission . . . 12
3.7 Collecting navigation data . . . 13
3.8 Killing the mission . . . 13
3.9 Self checkoff assessment . . . 14

4 Copying data from the Raspberry Pi to the topside computer **14**
4.1 Self checkoff assessment . . . 15

5 Visualizing logged data **15**
5.1 Self checkoff assessment . . . 17

6 Assessment **17**

1 Overview and Objectives

- Learning the basic concepts of navigation
- Mastering command-line operations
- Learning the basic concepts of the missions-StackUxV software configuration and launch utility
- Learning how to transfer and visualize vehicle logs

2 What is a navigation solution

Similar to any other autonomous vehicle, it is essential that an AUV knows its own position in order to transit to other places to achieve mission goals. Knowing its own position is called *navigation*. The best estimate of its current position is called the *navigation solution*. There are two types of navigation solution:

1. **Absolute navigation solution**: the absolute position of the vehicle in a geographic coordinate system (e.g. latitude and longitude).
2. **Relative navigation solution**: the position of the vehicle relative to a second object; such as a buoy, mooring, ship, diver, another autonomous vehicle, etc.

In this class, we will primarily work with absolute navigation. The absolute navigation solution includes parameters such as: (1) longitude, (2) latitude, (3) local x (optional), (4) local y (optional), (5) depth, (6) roll angle, (7) pitch angle, (8) heading angle, (9) surge speed, (10) sway speed, (11) heave speed.

When the vehicle is not underwater (e.g. on the water surface), we can use the GPS and AHRS/IMU to obtain the majority of these navigation parameters. However, GPS does not work underwater because of the high attenuation of electromagnetic waves through the water; making underwater navigation extremely difficult (we will discuss this topic in our Underwater Navigation session). In this lab, we will only focus on GPS and AHRS based surface navigation.

3 Running a mission that logs GPS and AHRS navigation data

In this lab, you will launch a navigation data collection mission using your training-kit. It is a *do nothing* but logging data mission. That is, the vehicle is configured to hypothetically sit on the water surface and collect sensor data. The actuators (i.e., the control surfaces and thruster) will not engage.

3.1 Exploring StackUxV's software configuration and launch utility (`missions-StackUxV`)

As we discussed in previous lectures, the Sea Beaver III AUV class is equipped with two onboard embedded computers: a Raspberry Pi (RasPi) and a PocketBeagle. Both computers are installed with the same software stacks; however, they are configured for different functions. The PocketBeagle primarily runs low-level control and driver software that interfaces with sensors and actuators, while

RasPi handles higher-level tasks such as sensor processing, navigation, autonomy, and communication. This configuration is managed by the StackUxV configuration and launch utility, `missions-StackUxV`.

`missions-StackUxV` is a powerful utility that can configure the StackUxV software architecture for various types and classes of autonomous vehicles, not just SeaBeaver AUVs. This utility has two primary functions: (1) configuring and defining the software architecture and (2) launching software during missions. In today's lab, we will focus primarily on the latter.

To launch a mission, we first need to login to RasPi. If you do not know how to do that, please refer to the lab: *How to login to a SeaBeaver III AUV*. Once you SSH into the RasPi, use `ls -F` command to list all directories in your home directory:

```
$ ls -F
StackUxV-DEBIAN-PKG-ARCHIVE/  librobotcontrol/  missions-StackUxV/  moos-ivp/  pablo-common/
```

At a minimum, you will see the `StackUxV-DEBIAN-PKG-ARCHIVE`, `moos-ivp`, and `missions-StackUxV` directories. The source codes for `moos-ivp` and `missions-StackUxV` projects are located in their respective directories.

While the `moos-ivp` and `missions-StackUxV` projects are distributed with source code for this class, the `StackUxV`, `StackUxV-drivers`, `VECTORS`, and `AudioBox` projects are distributed as Debian packages that contain compiled binary files. The installation files for these packages (i.e., `*.deb` files) are located in `StackUxV-DEBIAN-PKG-ARCHIVE`; we will show you how they are installed and updated in upcoming labs.

Let's change the directory to `missions-StackUxV`:

```
$ cd missions-StackUxV/
```

If you use `ls -F` command to explore, you will see following directories and files:

```
$ ls -f
architecture/   clean.sh*  fleet/           logs/      release-notes.txt
build_scripts/  cruise/    global_plugs/    meta/      vehicle/
clean_logs.sh*  data/      launch_scripts/  README.md  virtual_experiment/
```

While we don't need to know every detail about each file and directory, it's helpful to have a basic understanding of their functions:

- `global_plugs/`: A directory that defines the MOOS applications and configuration parameters common to all vehicles and nodes (*note: this list is not privileged; users can overwrite or modify these applications and their configuration parameters through custom architecture, vehicle, and cruise definitions*).

- `architecture/`: A directory containing a selectable list of available vehicle architectures. Each architecture is represented by its own directory, which contains architecture-specific configuration parameters, and a "drag-and-drop" directory structure for defining additional MOOS applications associated with that architecture.

- `vehicle/`: A directory containing a selectable list of vehicle configurations. Each vehicle is

represented by its own directory, which contains vehicle-specific configuration parameters, and a "drag-and-drop" directory structure for defining additional MOOS applications associated with that vehicle.

- `cruise/`: A directory containing a selectable list of available cruise or mission configurations. Each cruise is represented by its own directory, which contains cruise-specific configuration parameters and a "drag-and-drop" directory structure for defining additional MOOS applications associated with that cruise.

- `build_scripts/`: A directory containing various scripts and programs designed to simplify operator tasks (e.g., `turn_off_seabeaver`, `ktstack`, etc.). This directory is typically included in the shell path, so users do not need to go into it in order to run these commands.

- `launch_scripts/`: A directory containing various scripts used to configure and launch missions. Because this is one of the most commonly used directories, we have created the alias `stklaunch` to allow users to quickly change into it:

- `meta/`: The scripts located here compile the final mission configuration files (i.e., target files) based on the selected fleet, architecture, vehicle, and cruise. The resulting target files are placed in `meta/targ_files`. Note that these are temporary files and are re-generated each time a mission is launched.

- `logs/`: A directory containing vehicle log files.

- `clean.sh`: This script clears the cache for the current configuration. It also removes vehicle log files from `missions-StackUxV/logs/`.

- `clean_logs.sh`: This removes vehicle log files in `missions-StackUxV/logs/` without clearing the cache.

- `README.md`: A markdown file that provides essential information about this project.

- `release-notes.txt`: Developer's release notes.

- `fleet/`: A directory containing a selectable list of fleets. Each fleet defines a group of collaborating vehicles, their communication and relative navigation patterns. This is beyond the scope of 2.S01.

- `virtual_experiment/`: A directory containing a selectable list of virtual experiment configurations This is beyond the scope of 2.S01.

3.1.1 Updating `missions-StackUxV` software to the latest version in Raspberry Pi and PocketBeagle

It is always advised to update the repository to our latest version using `git`. But first, let's make sure we are in the correct git branch; i.e. `2026_2s01`:

```
$ git branch
```

You should see an output like this, and that means that you are in `2026_2s01` branch:

```
$ git branch
* 2026_2s01
  master
```

If otherwise, switch to `2026_2s01` by executing following commands:

```
$ git fetch
$ git checkout 2026_2s01
```

Once you are in the correct branch, let's pull the latest software.

```
$ git pull origin 2026_2s01
```

Don't forget that `missions-StackUxV` repository also exists in the PocketBeagle, and you need to update that as well. Please SSH into the PocketBeagle and use the same commands to switch to `2026_2s01` branch and update `missions-StackUxV`.

3.1.2 *Quickly* updating `missions-StackUxV` on both Raspberry Pi and PocketBeagle in one go

During field operations, you will need to update the `missions-StackUxV` repository very often. We understand that it is not convenient for the operator to execute a number of commands to update both RasPi and PocketBeagle. Therefore, we have created the `update_missions_stackuxv` program that automatically updates the current branch of `missions-StackUxV` in both RasPi and PocketBeagle together. When you run this program in RasPi, it will first update RasPi and then remotely execute the update commands in the PocketBeagle (i.e. by sending remote commands via SSH). Please, make sure that you run this program in the RasPi, not in the PocketBeagle.

The source code for this program is located in `missions-StackUxV/build_scripts/`. However, this directory is already set to the RasPi's shell path. Therefore, you do not have to change the directory (cd) to this location to run this script.

```
$ update_missions_stackuxv
```

You will see an output similar to below:

```
$ update_missions_stackuxv


Updating missions-StackUxV on SeaBeaver Raspberry Pi..

missions-StackUxV is available.
missions-StackUxV is in branch:  2026_2s01

Updating missions-StackUxV..
Already up to date.
Remotely triggering git update on the PocketBeagle..
Debian GNU/Linux 10

BeagleBoard.org Debian Buster IoT Image 2020-04-06

Support: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian

default username:password is [debian:temppwd]

Being launched remotely by the RasPi..

Updating missions-StackUxV on SeaBeaver PocketBeagle..

missions-StackUxV is available.
missions-StackUxV is in branch:  2026_2s01

Updating missions-StackUxV..
Already up to date.
Connection to 192.168.7.2 closed.
```

Please note that the output also displays the current branch for each repository.

3.1.3 Selecting/configuring definition files

Before launching a mission using `missions-StackUxV`, you must select and configure three components. Although these will be covered in more detail later in the course, it is useful to have a basic understanding of them now:

1. **Select the vehicle architecture**: This defines the vehicle class. In our case, this is the SeaBeaver III architecture.

2. **Select the specific vehicle**: Individual vehicles may have slightly different parameter values, so the exact vehicle being used must be selected.

3. **Select the mission to run**: This is also referred to as the cruise. It defines the mission-specific parameters. In this lab, the mission will simply be to “do nothing” while collecting sensor data.

The configuration and launch are performed in the `launch_scripts` directory. So let us go there:

```
$ cd launch_scripts/
```

If you use `ls -F` command to explore, you will see following scripts:

```
$ ls -F
clean_configurations.sh*        configure_cruise.sh*          launch_hitl.sh*
clean_logs.sh*                  configure_fleet.sh*           launch_runtime.sh*
configure_architecture.sh*      configure_vehicle.sh*         launch_simulation.sh*
configure_batch_simulation.sh*  launch_batch_simulation.sh*   launch_topside.sh*
```

While we do not need to know every detail about each file and directory, it is helpful to have a basic understanding of their functions:

- `clean_configurations.sh`: Clears the prior configuration cache without removing vehicle log files.
- `clean_logs.sh`: Removes vehicle log files from `missions-StackUxV/logs/` without clearing the cache.
- `configure_architecture.sh`: Selects the vehicle's architecture.
- `configure_vehicle.sh`: Selects the vehicle to be used.
- `configure_cruise.sh`: Selects the mission to run.
- `configure_fleet.sh`: Selects the fleet to which this vehicle belongs (beyond the scope of 2.S01).
- `configure_virtual_experiment.sh`: Selects a batch virtual experiment to run (beyond the scope of 2.S01).
- `launch_fleet_simulation.sh`: Launches a multi-vehicle simulation (beyond the scope of 2.S01).
- `launch_hitl.sh`: Launches a hardware-in-the-loop (HITL) simulated mission.
- `launch_runtime.sh`: Launches a mission in the field.
- `launch_simulation.sh`: Launches a software-in-the-loop (SITL) simulated mission.
- `launch_topside.sh`: Launches the topside command and control system.
- `launch_virtual_experiment.sh`: Runs a batch virtual experiment (beyond the scope of 2.S01).

3.2 Defining the vehicle architecture

First, let us configure/select the vehicle architecture. Run the following command to list all the available architectures:

```
$ ./configure_architecture.sh
```

It will give you an output similar to this:

```
$ ./configure_architecture.sh
usage: ./configure_architecture.sh {architecture-name}

current architecture is set to:
(not set)

available architectures include:
dart
dart_buoy
eel
eel_sonar
ros_example
saab_ematt
sb2_dart
sb2_dart_deep
sb2_hydronet
seabeaver_iii
stackuxv_arch_cray
template_architecture
```

As you can see, there are several available architectures. They are for different robotic systems that utilize StackUxV. The architecture for the Sea Beaver III base AUV is `seabeaver_iii`. Let's select it:

```
$ ./configure_architecture.sh seabeaver_iii
```

In general, all configuration and launch scripts must be executed on both the Raspberry Pi and the PocketBeagle. However, this process has been simplified. When you run this script on the Raspberry Pi to configure the vehicle architecture, it automatically sends a remote SSH command to execute the same configuration on the PocketBeagle. As a result, you only need to configure the architecture on the Raspberry Pi. If successful, you'll see an output similar to the following:

```
$ ./configure_architecture.sh seabeaver_iii
Successfully configured the architecture to seabeaver_iii in seabeaver-raspi!
Configuring the architecture on the PocketBeagle. Architecture name: seabeaver_iii
Debian GNU/Linux 10

BeagleBoard.org Debian Buster IoT Image 2020-04-06

Support: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian

default username:password is [debian:temppwd]

Successfully configured the architecture to seabeaver_iii in debian!
```

3.3 Defining the vehicle

Now let us configure the vehicle name. Run the following command to list all the available vehicles:

```
$ ./configure_vehicle.sh
```

It will give you an output similar to this:

```
$ ./configure_vehicle.sh
usage: ./configure_vehicle.sh {vehicle-name}

current vehicle is set to:
(not set)

available vehicles include:
buoy_one
buoy_three
buoy_two
cap
ematt_prototype_1
fay
glider_one
ham
ivy
jan
kay
sb2_alpha
template_uuv
```

In later sessions, you will create a definition file for your own vehicle. But for now, let's use `cap` AUV's definition. Configure the vehicle:

```
$ ./configure_vehicle.sh cap
```

"Similar to the architecture configuration, this script first selects the vehicle on the Raspberry Pi, then automatically sends a remote SSH command to apply the same configuration on the PocketBeagle. If successful, you'll see an output similar to the following:

```
$ ./configure_vehicle.sh cap

Successfully configured the vehicle to cap in seabeaver-raspi!
Configuring the vehicle on the PocketBeagle. Vehicle name: cap
Debian GNU/Linux 10

BeagleBoard.org Debian Buster IoT Image 2020-04-06

Support: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian

default username:password is [debian:temppwd]

Successfully configured the vehicle to cap in debian!
```

3.4 Defining the cruise

Finally, we'll select the cruise (i.e., the mission) to run. Use the following command to list all available cruises:

```
$ ./configure_cruise.sh
```

At a minimum, it will show following cruises:

```
$ ./configure_cruise.sh
usage: ./configure_cruise.sh {cruisename}

current cruise definition is set to:
(not set)

available cruise names include:
a_confidence
b_zigzag
bench_test
bi_directional
bi_directional_east
bi_directional_west
c_depth_profiling
calibrate_fins
dart_sp
dart_voe_range_1000
dart_voe_range_3000
dart_voe_range_500
dive_test
ematt_altitude_test
glide_down
irminger
lab_4_do_nothing_log_data
lab_7_control
lab_9_navigation
mbat_towed
shipwreck_search
template_cruise
u_clock
u_counterclock
yoyo
zig_counter
```

Now let's select the correct cruise for this lab, `lab_4_do_nothing_log_data`:

```
$ ./configure_cruise.sh lab_4_do_nothing_log_data
```

3.5 Cleaning old vehicle log files

Each mission generates a log file containing all MOOS messages exchanged between different processes (i.e., MOOS applications), including all collected data. In this lab, we will post-process and analyze this data after the mission. However, your training kit may contain old log files left behind by a previous operator. It's a good idea to clear these logs before starting a new experiment. To do so, use the `./clean_logs.sh` script to remove any leftover log files from past missions.

```
$ ./clean_logs.sh
```

Please note that `./clean_logs.sh` performs a non-reversible operation. Make sure to back up any log files you wish to preserve; e.g. by copying them to your topside computer, before running the script.

3.6 Launching the mission

Finally, we can now launch the mission using the following command:

```
$ ./launch_runtime.sh
```

You will see an output similar to the one shown below. This command first remotely launches the necessary software on the PocketBeagle, followed by the software on the Raspberry Pi. Once the launch is complete, control will return to the terminal.

```
$ ./launch_runtime.sh

Launching Frontseat-Backseat mode from the backseat computer..
Launching Frontseat MOOS community on the PocketBeagle
Launching Backseat MOOS community on the RasPi..
RUNTIME launch in process..
Launching mode is: FRONTBACK_BACK
Current HydroMAN version: lite
Assembling targ_Backseat.moos ... Done
Assembling targ_Backseat.bhv ... Done
Launching Backseat MOOS Community. WARP is 1
Done
Runtime launch is complete..
Use 'screen -ls' command to list all screen sessions running on this computer.
```

The `launch_runtime.sh` script runs the software within a `screen` session, allowing it to operate as a background process. You can learn more about `screen` in the official manual: https://www.gnu.org/software/screen/manual/screen.html). To view all running screen sessions, use the `screen -ls` command.

```
$ screen -ls
There is a screen on:
        2066.targ_CAP_Backseat        (04/07/25 16:18:00)        (Detached)
1 Socket in /run/screen/S-seabeaver-raspi.
```

In this example, there is one screen session: `2066.targ_CAP_Backseat`. The number before the process name is the process ID (for more information: https://sourceware.org/glibc/manual/2.35/html_node/Process-Identification.html).

You can connect/attach to the screen session by using the following command:

```
$ screen -r targ_CAP_Backseat
```

You will now see real-time output from the backseat software running on the Raspberry Pi. To detach from this screen session, press `Ctrl+A`, release both keys, then press `D` (i.e., `Ctrl+A`, then `D`).

3.7 Collecting navigation data

Congratulations! You are now running the data collection mission on your training kit. You should turn the training-kit around to record AHRS data, and walk around with the kit to collect some GPS data.
Please make sure to collect data for at least 1–2 minutes while moving and rotating the kit. Otherwise, there may not be enough data for meaningful visualization later in the lab.

3.8 Killing the mission

Since the software is running in the background on both the Raspberry Pi and PocketBeagle, you cannot use `Ctrl+C` to stop it. Instead, you must use the `ktstack` command (short for 'kill-the-stack') to terminate the processes. It will terminate the software on both the Raspberry Pi and PocketBeagle. You will see an output similar to this:

```
$ ktstack

Killing MOOS and Goby processes in seabeaver-raspi..
sending self-exit instruction to 224.1.1.3:4000
pMITFS_MissionManager pid 2117    on 10.42.0.5 is self-exiting
uMAC_7458             pid 2067    on 10.42.0.5 is self-exiting
pMITFS_MissionScript  pid 2127    on 10.42.0.5 is self-exiting
pHelmIvP              pid 2107    on 10.42.0.5 is self-exiting
pShare                pid 2148    on 10.42.0.5 is self-exiting
pMITFS_HydroMANLite   pid 2138    on 10.42.0.5 is self-exiting
pLogger               pid 2097    on 10.42.0.5 is self-exiting
uProcessWatch         pid 2087    on 10.42.0.5 is self-exiting
MOOSDB_Backseat       pid 2068    on 10.42.0.5 is self-exiting
the moos is dead
Killing MOOS and Goby processes in the Frontseat computer..
Debian GNU/Linux 10

BeagleBoard.org Debian Buster IoT Image 2020-04-06

Support: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian

default username:password is [debian:temppwd]

sending self-exit instruction to 224.1.1.3:4000
uMAC_7665             pid 2427    on 192.168.7.2 is self-exiting
pMOOSLink             pid 2477    on 192.168.7.2 is self-exiting
iXsensMTi3            pid 2489    on 192.168.7.2 is self-exiting
iActuatorsPWM         pid 2499    on 192.168.7.2 is self-exiting
MOOSDB_Frontseat      pid 2431    on 192.168.7.2 is self-exiting
uProcessWatch         pid 2447    on 192.168.7.2 is self-exiting
pMITFS_ControlEngine  pid 2467    on 192.168.7.2 is self-exiting
pLogger               pid 2457    on 192.168.7.2 is self-exiting
iMITFS_Daemon         pid 2526    on 192.168.7.2 is self-exiting
/home/debian/missions-StackUxV/build_scripts/km: line 19: kill: (2427) - No such process
the moos is dead
```

3.9 Self checkoff assessment

- Are you able to successfully select the vehicle architecture?
- Are you able to successfully select the vehicle?
- Are you able to successfully select the cruise?
- Are you able to launch the mission?
- Are you able to terminate the mission?

4 Copying data from the Raspberry Pi to the topside computer

Once you've completed the mission, you'll need to copy the log files that contain all the collected data. These logs are stored on the Raspberry Pi at `/home/seabeaver-raspi/missions-StackUxV/logs`.

To visualize the data, you'll need to transfer the logs to your topside laptop.

On your topside laptop, create a directory where you'd like to store the data logs, and navigate to it. We recommend creating a directory called `2S01_data` in your home directory, with a subdirectory named after the date of the experiment. For example:

```
$ mkdir ~/2S01_data
$ cd ~/2S01_data
$ mkdir 20260407
$ cd 20260407/
```

Now we can use the secure copy protocol (SCP) to copy logs from the Raspberry Pi to the topside computer over SSH:

```
$ scp -r seabeaver-raspi@10.42.0.5:/home/seabeaver-raspi/missions-StackUxV/logs/LOG* .
```

You will be prompted to enter the Raspberry Pi's password. After entering it, the file transfer may take a few minutes to complete. Once finished, use the `cd` and `ls` commands to navigate and verify that the logs have been successfully copied.

4.1 Self checkoff assessment

Ensure that:

- Are you able to successfully copy logs from the Raspberry Pi to your topside computer?

5 Visualizing logged data

To visualize the collected data, we'll use the `alogview` utility that comes with MOOS-IvP. Depending on how many missions you've run, there may be multiple directories containing log files. Navigate to the directory corresponding to the mission you want to visualize. Note that log directories are named using the date and time of the mission. Inside, you'll find several files, including:

```
$ ls
LOG_CAP_Backseat_7_4_2026_____16_18_01.alog   LOG_CAP_Backseat_7_4_2026_____16_18_01.slog
LOG_CAP_Backseat_7_4_2026_____16_18_01.blog   LOG_CAP_Backseat_7_4_2026_____16_18_01.ylog
LOG_CAP_Backseat_7_4_2026_____16_18_01._moos  targ_CAP_Backseat._bhv
```

In later sessions, we will go into more details on these files. For now, we are interested in the *.alog file. Let's open it with `alogview` tool; for example:

```
$ alogview LOG_SB2_ALPHA_Frontseat_27_3_2024_____16_42_04.alog
```

You will see a window similar to Figure 1.

Figure 1: Alogview screen when you load the logs

Explore various options such as trail size, trail length, stream speed and visualize your navigation data. Try to create a plot similar to Figure 2. More details regarding `alogview` is given here: `https://oceanai.mit.edu/ivpman/apps/alogview`.

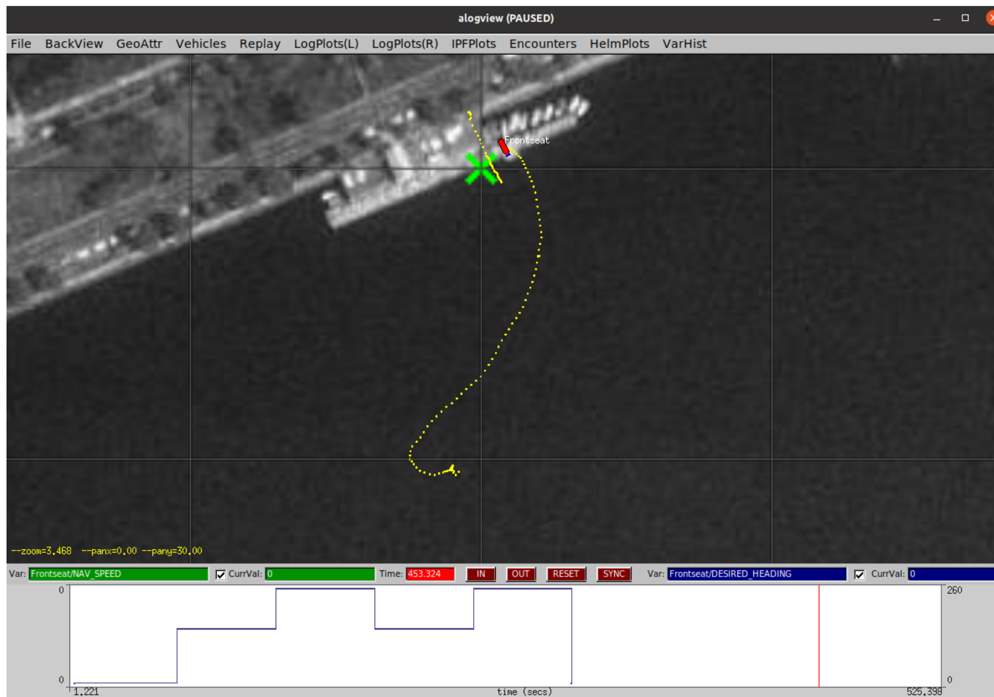

Figure 2: Alogview screen when you load the logs

If the background of the alogview does not show the sailing pavilion, you can try the following command instead:

```
$ alogview --bg=MIT_SP.tif LOG_SB2_ALPHA_Frontseat_27_3_2024_____16_42_04.alog
```

5.1 Self checkoff assessment

- Are you able to visualize the surface navigation data using the alogview tool?

6 Assessment

The lab assessments are structured to ensure that you stay on track to complete your AUV by the end of the semester for in-water trials. You are expected to demonstrate to one of the instructors that you can complete the following tasks. More importantly, please don't hesitate to communicate with the instructors if you encounter any issues.

1. Visualize the surface navigation data that you collected using alogview (similar to Figure 2), and send a screenshot to the instructors using Piazza .