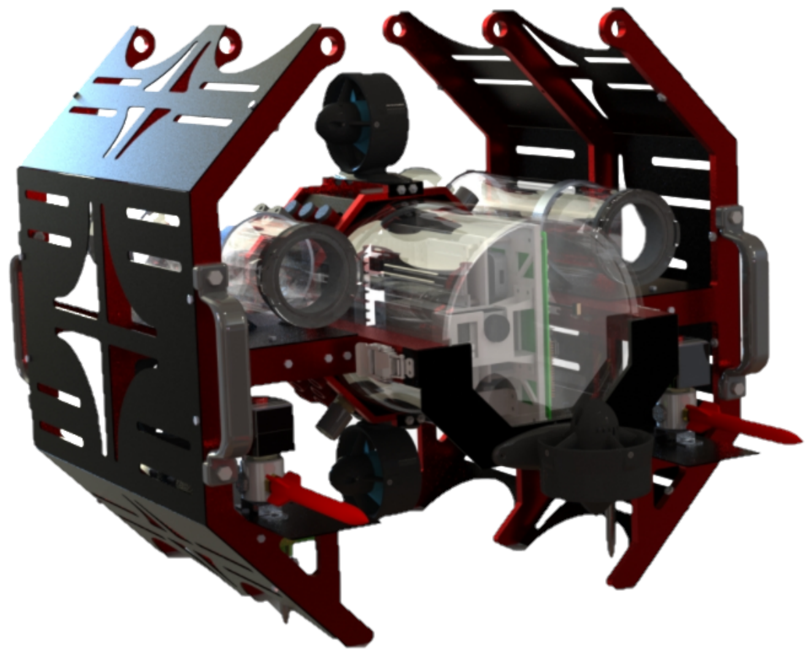
ARVP

autonomous robotic vehicle project

Software Team Onboarding 2019 Mission Task



Welcome to ARVP!

Your first task is to program our robot, Auri, to navigate through a qualification course. Completing this task will also qualify you to gain full access to ARVP's software repository on GitHub, following which you'll be assigned a task suitable to your interests and skills.

This is to be considered an introduction, not a skill test, so feel free to reach out for help as you need it. We understand that not everyone has an extensive background in programming, which makes ARVP the perfect opportunity for development.

Before you start

Please take a moment to fill out **this form** to help us get to know you better. Ensure that the GitHub information you submit is correct, as we will use it to give you access to our software repository.

Communication

ARVP uses Slack to facilitate communication. Here's an **invitation to join**. This gives you access to the following channels:

- #onboarding
    - For discussing, collaborating on the mission task, or for general help in regards to software onboarding.
- #random
    - For memes, trying to sell stuff that people don't want, etc.
- #software-team
    - For discussing topics related to the software team.
- #general
    - For discussing topics related to ARVP as a whole.

The co-leads for ARVP's 2019 software team are Nicholas Wengel and Willard Farmer. If you need to contact us, it is preferred if you use Slack instead of email.

RoboSub

The goal of ARVP is to program an autonomous robot for the purpose of completing a competition course: Robosub. For a bit of context, or for curiosity, you might want to check out their site to see what they're all about, and what past competitions have been like. This will be helpful for understanding our mission planner when competition props are referenced, for example.

Here are the **rules for RoboSub 2019**.

System Requirements

It’s recommended that you have the following:

- x64 computer with at least 4GB of RAM (8GB preferred)
  - Though you may use a desktop for development, a laptop is required for connecting to Auri at pool tests.
  - An NVIDIA GPU is required for running our computer vision code. If you do not have one, the applicable software will simply be disabled.
  - Our simulator will run very slowly if your computer does not have at least very basic integrated graphics.
- USB drive (for installing Ubuntu 18.04)
- Ethernet cable (for connecting to Auri)

While it is possible to install Ubuntu on a Surface or Mac, the experience will not be great, and likely not worth the effort. Whatever machine you decide to use should at least have an ethernet port (or a USB3.0/USB-C port and an ethernet dongle).

If you have a Mac, you can follow this tutorial. It’s unlikely that the built-in keyboard and trackpad on a Mac laptop will be supported out of the box on Ubuntu. Hence, you’ll need an external keyboard (and maybe a mouse). Since you’ll need connection slots for the Ubuntu install USB as well, you’ll probably need a splitter (and/or adapter(s) for USB-C).

Installing Ubuntu 18.04

This is the OS we use for development. Using a virtual machine is not recommended. Thus, here follows instructions on how to dual boot. What this means is that Ubuntu will be installed alongside Windows so that you can select which OS to use whenever you power on your computer. If you don’t have or want Windows, there’s nothing wrong with just installing Ubuntu by itself either.

Head to Ubuntu’s website and download the latest Ubuntu 18.04 ISO.

If your network is spotty, you probably want to torrent the ISO so that you can resume your download in the event that your connection fails. This will require a bittorrent client such as µTorrent.

Once you have the Ubuntu 18.04 ISO you need to write it to your USB. Linux Live USB Creator is a Windows program that can do this for you. Here is a guide on how to use it.

Before you can use your new Linux Live USB you'll need to partition your computer's storage. This basically just means reserving space for Ubuntu.

Windows will not likely allow you to shrink a partition sufficiently without also performing the following steps:

- Clean / defragment your disk
- Disable recovery
- Disable hibernation
- Disable pagefile

You can re-enable recovery and the pagefile once you've shrunk a partition to the desired size. Do not re-enable hibernation as it will make your Windows partition inaccessible from within Ubuntu.

It is recommended that you use the Ubuntu installer to further subdivide the newly reserved partition into three smaller partitions: / (root), /home (where your personal files go), and swap. By separating your root directory from your files, you'll be able to reinstall Ubuntu effortlessly in the future if somehow it becomes unusable.

Ideal partition sizes:

- / - 40GB or more
- /home - 10GB or more
- swap - 8GB or more

Now you can proceed to install Ubuntu.

The above procedure can be daunting, so don't hesitate to ask for help.

Development Environment

Now that you have a fresh installation of Ubuntu, you'll need a code editor. A complete IDE is unnecessary since most of our tools, including compilation, are run in a terminal.

Visual Studio Code is an excellent editor, but there are others. CLion is also a great option. You can use a student license for CLion and all Jetbrains software, instructions for using Clion can be found here.

To contribute to our software repository, you'll need to use Git.

Once Git is installed, you need to link it to your GitHub account.

Due to the multi-window nature of our development environment, you may find Ubuntu workspaces to be invaluable. I personally use a 4x4 grid, placing VS Code in one window, a terminal for compiling in another window, a web browser in another, etc.

Our software repository

Is located **here**. This is a private repository so if you haven't been given access rights yet, you will be unable to see it. If you have only just filled out the onboarding form, it may take a day or so for us to grant access. If a day or so has passed and you still do not have access, don't hesitate to contact Nicholas Wengel of Willard Farmer on Slack.

Note that until you have completed the onboarding, you will not have write access to the repository. We have **copied the repository for your use**.

Clone the repository into your home directory (ideally, but anywhere should work).

```
cd ~
git clone git@github.com:arvpUofA/au_everything_onboarding.git
```

Rename the cloned folder from "au_everything_onboarding" to just "au_everything".

Follow the instructions on the repository's README.md to set things up. Note the section "Fresh Install?" that instructs on how to run a script that will automate everything.

That is:

```
# Go to parent directory of au_everything and run setup.sh
cd au_everything/..
bash -i ./au_everything/scripts/setup.sh
```

Programming Languages

Our repository is written primarily in C++ with a bit of Python on the side. The mission task will exclusively involve C++. If you are unfamiliar with this language, here is a good tutorial. Since C++ is a compiled language, it needs to be compiled every time its source code is changed. Python is an interpreted language, which is why the code can be run immediately once edited. You may want to research the difference for your own knowledge (one explanation).

You may want to briefly skim the tutorial given above, but don't worry about trying to cram all the information at once. You'll learn better if you try features one at a time, which means just starting to use the language. You can actually copy and paste a lot of code for the mission task, using pre-existing missions, which will help you develop a basic understanding of the syntax.

Compiling

Editing the source code will not be enough for your changes to take effect when you run our software stack. Every time you make a change, the high-level code (C++ in the case of this task) needs to be compiled into machine code that can be directly executed by your machine's CPU. To compile, do the following:

```
# catkin_make may only be called from the catkin_ws folder
cd au_everything/catkin_ws
catkin_make
```

Common Issues

In the past, our software stack depended on ROS Kinetic, but last year we upgraded to ROS Melodic. However, not all packages that we depended on have been upgraded to ROS Melodic yet, so we have to compile them ourselves. In the au_everything folder, you'll notice another folder called uwsim_ws, which contains packages related to our simulator.

If you get an error during compilation, which references osg_… do the following:

```
cd au_everything/uwsim_ws
catkin_make_isolated --install
```

To register the newly compiled packages, close your current terminal and open a new one.

If you get errors when running the arvp_sim_start command (which you'll read about later), check your .bashrc for two repeated lines of "source /opt/ros/melodic/setup.bash". Delete them. The bottom of your .bashrc should now have the line "source /home/ninja/au_everything/scripts/arvp_bashrc.sh" (assuming you haven't modified it). The .bashrc can be found in your home directory. By default it is hidden, but can be shown by pressing ctrl-h. This file is run every time you open a new terminal, so if you change it, open and close any relevant terminals.

Making changes

When developing a new feature, it's important to create a branch to work on. A branch is a copy of the repository at the time that you make it. It won't be affected by changes other developers make to the repository over time. For example, suppose you're working on a computer vision algorithm, and someone overhauls the way we organize and execute them, you'll be able to finish your work before integrating it with the overhaul.

This is how you make a new branch on your local machine:

```
git checkout -b "mybranch"
```

The portions of your branch that you're not working on will become outdated, but that's OK because you'll update it when you're finished. The **master** branch is where production code that's been reviewed and tested lives. When you want to update the latest code from **master** onto your branch, run this:

```
git checkout master
git pull
git checkout mybranch
git merge master
```

This will switch to the **master** branch, pull changes from the remote, switch back to your branch, and merge the **master** branch into it. If you develop using multiple computers, you'll probably use `git pull` on your own branch to update changes from the remote whenever you switch machines.

Sometimes you'll want to work on multiple projects at the same time, or checkout code that someone else has written. Before you can change branches though, git requires that you don’t have any outstanding changes that need to be saved. You can stash your changes, switch branches, switch back, and pop your changes back onto your branch. This is like stuffing everything into a bag, carrying it, and dropping it again. In fact, you can pop your changes onto any branch, not just the one you stashed from. First, let’s see if there’s any changes to stash:

```
git status
```

This will print the names of files that have been changed. Red ones indicate that they haven’t been added to the **index** yet. Green ones have been. The **index** is like a shipping box that you slowly fill up before actually sending it. When you seal the **index**, or make it ready to go, this is called **committing.** We’ll talk more about this later. Here’s how to stash:

```
git stash
```

Now you’re carrying your changes around. Let’s switch to a new branch:

```
git checkout “otherbranch”
```

Notice that the -b flag is missing. That’s only for making a new branch. You can check what branch you’re on, and what branches are available:

```
git branch
```

You’ll notice that not all the branches on the remote repository are available. This command will only display the branches that are local, or cached. You can update the list of known branches like so:

```
git fetch
```

When you’re ready to drop your stashed changes onto the current branch, do this:

```
git stash pop
```

If something is horribly wrong with your branch, you can reset it to the latest **commit**, which is like a checkpoint. This cannot be reverted:

```
git reset --hard
```

You’ll probably rarely need to use stashing, so let’s try making a commit. The git workflow is like this: unsaved changes -> index -> commit -> remote. The **remote** is just another name for our repository, to differentiate it from the one you’ve downloaded locally. You can add files to the index like so:

```
git add *
```

This will add all files in and below the working directory. You could also choose all files in a specific directory like:

```
git add mydir/*
```

Or add a single file:

```
git add myfile
```

You can again use `git status` to verify that your files have been added to the index (they should be green). When you’re ready to ship things, or want to make a checkpoint, you should make a commit. Often it’s a good idea to break your work into multiple commits so that you can reset things if you need to undo a step in your development process:

```
git commit -m “add my feature”
```

If you accidentally commit something that shouldn’t have been, you can undo the last N commits, leaving your branch in the state before you committed:

```
git reset HEAD~N
```

When you’re ready to push your changes to our repository, do this:

```
git push
```

If you haven’t pushed your branch to a remote before, git will probably complain about setting an upstream. This just means to set which branch on the remote to push to. You can copy and paste the command git recommends.

If your branch is done, it's time to make a pull request. Go onto GitHub, select your branch, and make a PR. Feel free to look at past PRs for exemplars.

Forks (optional)

Sometimes it's nice to backup incomplete work online without cluttering the main repository. This is what "forks" are for. A fork is a personal copy of a repository.

To fork our repository go to **URL** and click the "Fork" button near the top right of the page. Once your fork is created, open a terminal and type the following to see your current remotes:

```
git remote -v
```

You should see something like this:

```
origin     git@github.com:arvpUofA/au_everything.git (fetch)
origin     git@github.com:arvpUofA/au_everything.git (push)
```

This indicates the remote locations that Git is currently aware of. Now you want to make it aware of your new fork. Run (inserting your GitHub username):

```
git remote add <username> git@github.com:<username>/au_everything.git
```

If you run `git remote -v` again you'll see:

```
<username> git@github.com:<username>/au_everything.git (fetch)
<username> git@github.com:<username>/au_everything.git (push)
origin     git@github.com:arvpUofA/au_everything.git (fetch)
origin     git@github.com:arvpUofA/au_everything.git (push)
```

To fetch the contents of the remotely forked repository, run `git fetch <username>`. To update all your remotes, run `git fetch --all`.

Now you can push and pull code to your fork. For example, if you are writing code in your private branch `fix123` and you want to share it with another member, but you don't want to push it to the main repository yet, you can do:

```
git push <username> fix123
```

This pushes the code in the current local branch to the fix123 remote branch in the <username> remote.

Checking out another member's branch (optional)

To access another person's fork, you have to add their remote. You might want to do this, for example, if you want to see what someone else did for the onboarding task.

```
git remote add <username> git@github.com:<username>/au_everything.git
```

Then run `git fetch <username> branchName` to download their remote branch. To checkout the code under a local branch, run:

```
git checkout -b test1234 <username>/branchName
```

This creates a new local branch called test1234 which tracks the remote branch <username>/branchName, and then checks it out.

This copying is necessary because remote branches are read-only. To modify them, they must first be pulled, edited locally, and then changes must be committed.

Further on Git

A comprehensive tutorial on Git is beyond the scope of this document, so here's a guide. Avoid trying to learn Git all at once, just pick out parts on a need to know basis. You'll probably, for example, want to figure out how to push code to a branch right away.

Mission Task

Now that you've setup our software repository, it's time to use it!

Your task is to create a mission for Auri, guiding her through a course we've built for our simulator. This task is meant to reinforce a few basic points:

- The ability to confidently run our software stack end-to-end
- A basic understanding of C++

Following the completion of this task, there are a number of other exercises that you should complete at your own leisure. These involve using ROS, as well as overviews of some of Auri's software components such as our mapper, computer vision, etc.

The Workspace

Our repository is organized into a number of subrepos. These are located in the au_everything/catkin_ws/src folder. Our computer vision, for example, is isolated to the au_vision folder, sonar is in au_sonar, and so on. Your task is isolated to the au_planner subrepo which houses our mission planner.

We've created a template for you to work within. Look for a file called sm_onboarding.cpp within the au_everything/catkin_ws/src/au_planner/src/missions folder. Your job is to write the code for completing a simulated course. This involves writing the sequence of actions, as well as placing the appropriate #includes at the top of the file.

You're encouraged to look at the code for existing missions, and can even run them if you're not sure what some functions do.

Running a mission

Before you can run a mission, you need to start our software stack with the following command:

```
arvp_sim_start <course_name>
```

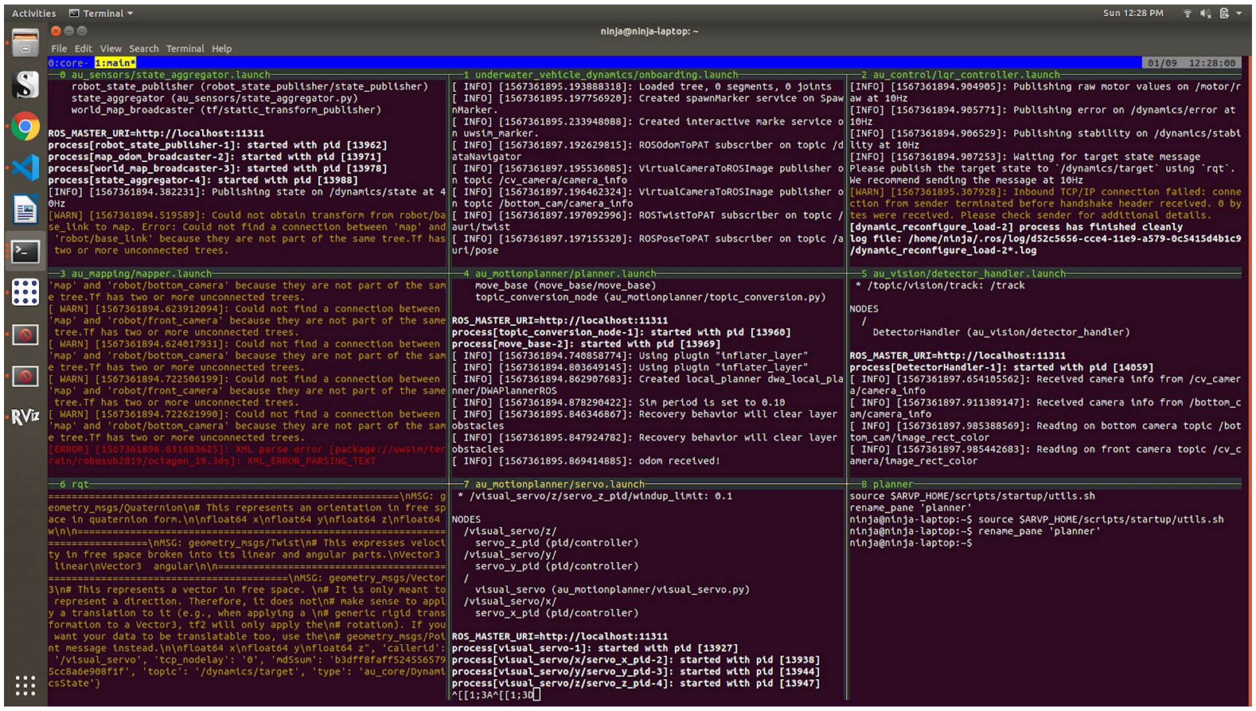
This will start the simulator, as well as a few other nodes. For running the onboarding mission, you'll want to use the "onboarding" course (**there are others though**).

Be aware that some missions are designed to run on specific courses. As well, some rely on computer vision. If you do not have a GPU, or don't want to set up computer vision, there's an option to enable the mock detector.

Running the above command will split your terminal into a grid of smaller ones. You can switch between windows by clicking on them, and can scroll up and down a select window by pressing ctrl+b followed by PgUp/PgDown. To end any of the running nodes (especially to restart them), select a terminal and press ctrl+c. Exiting the overall window will not kill the software stack. To kill it, use any terminal to run the following:

```
tmux kill-server
```

If this is your first time running our software stack, the simulator may seem to hang. Click the window labelled "1 underwater_vehicle_dynamic/<course_name>.launch" and press enter. It just needs a confirmation to download some assets.



```
0:core- 1:main*                                                    01/09  12:28:00
```

```
0 au_sensors/state_aggregator.launch
    robot_state_publisher (robot_state_publisher/state_publisher)
    state_aggregator (au_sensors/state_aggregator.py)
    world_map_broadcaster (tf/static_transform_publisher)

ROS_MASTER_URI=http://localhost:11311
process[robot_state_publisher-1]: started with pid [13962]
process[map_odom_broadcaster-2]: started with pid [13971]
process[world_map_broadcaster-3]: started with pid [13978]
process[state_aggregator-4]: started with pid [13988]
[INFO] [1567361894.382231]: Publishing state on /dynamics/state at 4
0Hz
[WARN] [1567361894.519589]: Could not obtain transform from robot/ba
se_link to map. Error: Could not find a connection between 'map' and
 'robot/base_link' because they are not part of the same tree.Tf has
 two or more unconnected trees.
```

```
1 underwater_vehicle_dynamics/onboarding.launch
[ INFO] [1567361895.193888318]: Loaded tree, 0 segments, 0 joints
[ INFO] [1567361895.197756920]: Created spawnMarker service on Spaw
nMarker.
[ INFO] [1567361895.233948088]: Created interactive marke service o
n uwsim_marker.
[ INFO] [1567361897.192629815]: ROSOdomToPAT subscriber on topic /d
ataNavigator
[ INFO] [1567361897.195536085]: VirtualCameraToROSImage publisher o
n topic /cv_camera/camera_info
[ INFO] [1567361897.196462324]: VirtualCameraToROSImage publisher o
n topic /bottom_cam/camera_info
[ INFO] [1567361897.197092996]: ROSTwistToPAT subscriber on topic /
auri/twist
[ INFO] [1567361897.197155320]: ROSPoseToPAT subscriber on topic /a
uri/pose
```

```
2 au_control/lqr_controller.launch
[INFO] [1567361894.904905]: Publishing raw motor values on /motor/r
aw at 10Hz
[INFO] [1567361894.905771]: Publishing error on /dynamics/error at
10Hz
[INFO] [1567361894.906529]: Publishing stability on /dynamics/stabi
lity at 10Hz
[INFO] [1567361894.907253]: Waiting for target state message
Please publish the target state to `/dynamics/target` using `rqt`.
We recommend sending the message at 10Hz
[WARN] [1567361895.307928]: Inbound TCP/IP connection failed: conne
ction from sender terminated before handshake header received. 0 by
tes were received. Please check sender for additional details.
[dynamic_reconfigure_load-2] process has finished cleanly
log file: /home/ninja/.ros/log/d52c5656-cce4-11e9-a579-0c5415d4b1c9
/dynamic_reconfigure_load-2*.log
```

```
3 au_mapping/mapper.launch
'map' and 'robot/bottom_camera' because they are not part of the sam
e tree.Tf has two or more unconnected trees.
[ WARN] [1567361894.623912094]: Could not find a connection between
'map' and 'robot/front_camera' because they are not part of the same
 tree.Tf has two or more unconnected trees.
[ WARN] [1567361894.624017931]: Could not find a connection between
'map' and 'robot/bottom_camera' because they are not part of the sam
e tree.Tf has two or more unconnected trees.
[ WARN] [1567361894.722506199]: Could not find a connection between
'map' and 'robot/front_camera' because they are not part of the same
 tree.Tf has two or more unconnected trees.
[ WARN] [1567361894.722621990]: Could not find a connection between
'map' and 'robot/bottom_camera' because they are not part of the sam
e tree.Tf has two or more unconnected trees.
[ERROR] [1567361896.631683625]: XML parse error [package://uwsim/ter
rain/robosub2019/octagon_19.3ds]: XML_ERROR_PARSING_TEXT
```

```
4 au_motionplanner/planner.launch
    move_base (move_base/move_base)
    topic_conversion_node (au_motionplanner/topic_conversion.py)

ROS_MASTER_URI=http://localhost:11311
process[topic_conversion_node-1]: started with pid [13960]
process[move_base-2]: started with pid [13969]
[ INFO] [1567361894.740858774]: Using plugin "inflater_layer"
[ INFO] [1567361894.803649145]: Using plugin "inflater_layer"
[ INFO] [1567361894.862907683]: Created local_planner dwa_local_pla
nner/DWAPlannerROS
[ INFO] [1567361894.878290422]: Sim period is set to 0.10
[ INFO] [1567361895.846346867]: Recovery behavior will clear layer
obstacles
[ INFO] [1567361895.847924782]: Recovery behavior will clear layer
obstacles
[ INFO] [1567361895.869414885]: odom received!
```

```
5 au_vision/detector_handler.launch
 * /topic/vision/track: /track

NODES
  /
    DetectorHandler (au_vision/detector_handler)

ROS_MASTER_URI=http://localhost:11311
process[DetectorHandler-1]: started with pid [14059]
[ INFO] [1567361897.654105562]: Received camera info from /cv_camer
a/camera_info
[ INFO] [1567361897.911389147]: Received camera info from /bottom_c
am/camera_info
[ INFO] [1567361897.985388569]: Reading on bottom camera topic /bot
tom_cam/image_rect_color
[ INFO] [1567361897.985442683]: Reading on front camera topic /cv_c
amera/image_rect_color
```

```
6 rqt
==============================================================\nMSG: g
eometry_msgs/Quaternion\n# This represents an orientation in free sp
ace in quaternion form.\n\nfloat64 x\nfloat64 y\nfloat64 z\nfloat64
w\n\n================================================================
=================\nMSG: geometry_msgs/Twist\n# This expresses veloci
ty in free space broken into its linear and angular parts.\nVector3
 linear\nVector3  angular\n\n========================================
========================================\nMSG: geometry_msgs/Vector
3\n# This represents a vector in free space. \n# It is only meant to
 represent a direction. Therefore, it does not\n# make sense to appl
y a translation to it (e.g., when applying a \n# generic rigid trans
formation to a Vector3, tf2 will only apply the\n# rotation). If you
 want your data to be translatable too, use the\n# geometry_msgs/Poi
nt message instead.\n\nfloat64 x\nfloat64 y\nfloat64 z", 'callerid':
 '/visual_servo', 'tcp_nodelay': '0', 'md5sum': 'b3dff8faff524556579
5cc8a6e908f1f', 'topic': '/dynamics/target', 'type': 'au_core/Dynami
csState'}
```

```
7 au_motionplanner/servo.launch
 * /visual_servo/z/servo_z_pid/windup_limit: 0.1

NODES
  /visual_servo/z/
    servo_z_pid (pid/controller)
  /visual_servo/y/
    servo_y_pid (pid/controller)
  /
    visual_servo (au_motionplanner/visual_servo.py)
  /visual_servo/x/
    servo_x_pid (pid/controller)

ROS_MASTER_URI=http://localhost:11311
process[visual_servo-1]: started with pid [13927]
process[visual_servo/x/servo_x_pid-2]: started with pid [13938]
process[visual_servo/y/servo_y_pid-3]: started with pid [13944]
process[visual_servo/z/servo_z_pid-4]: started with pid [13947]
^[[1;3A^[[1;3D
```

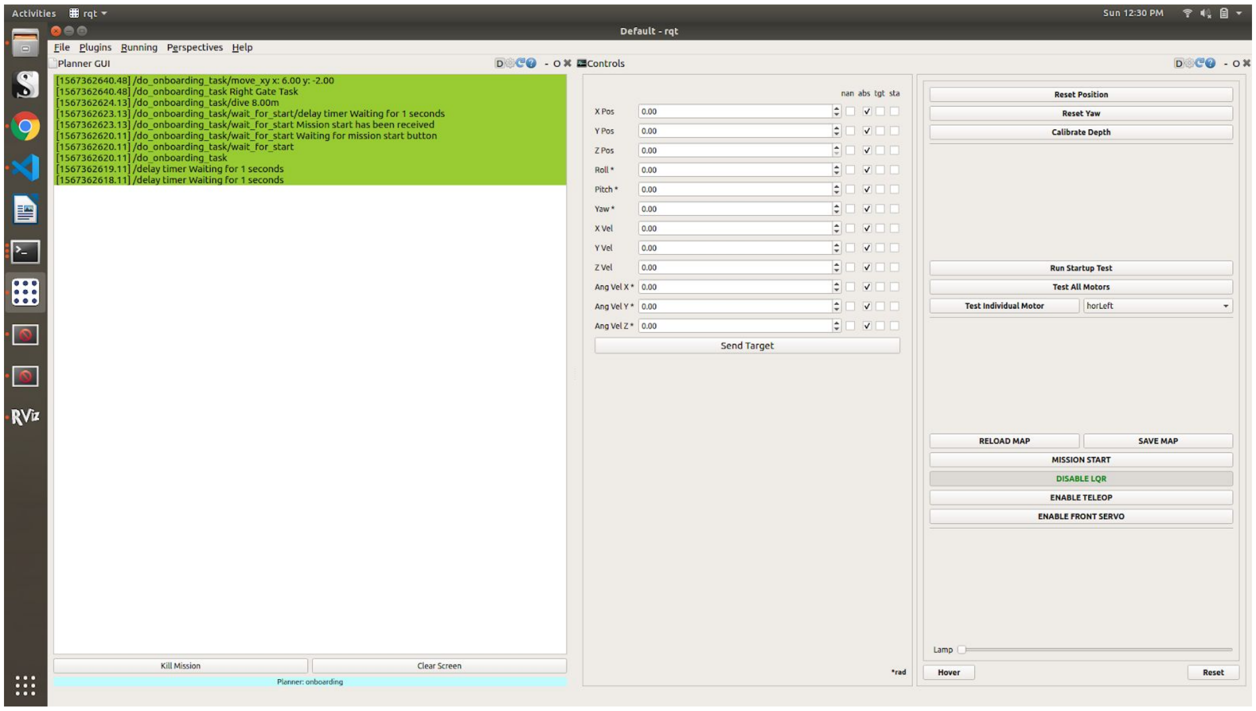
```
8 planner
source $ARVP_HOME/scripts/startup/utils.sh
rename_pane 'planner'
ninja@ninja-laptop:~$ source $ARVP_HOME/scripts/startup/utils.sh
ninja@ninja-laptop:~$ rename_pane 'planner'
ninja@ninja-laptop:~$
```

A few windows will have opened at this point:

- rqt
- RViz
- Simulator + keyboard control

We use rqt to display GUIs with helpful information about Auri. By selecting Plugins->ARVP from the top-left menu, you can open and dock our GUIs:

- Planner GUI
    - This will display status information ("diving", "moving to X", etc.) about a mission if one is running. This information is the same as in the terminal, but is more convenient to dock alongside other plugins.
- Controller GUI
    - This is the primary GUI for controlling the robot. The only thing you need to know about at this point is the "MISSION START" button near the bottom. Pressing this will signal the mission planner to begin if it's waiting.

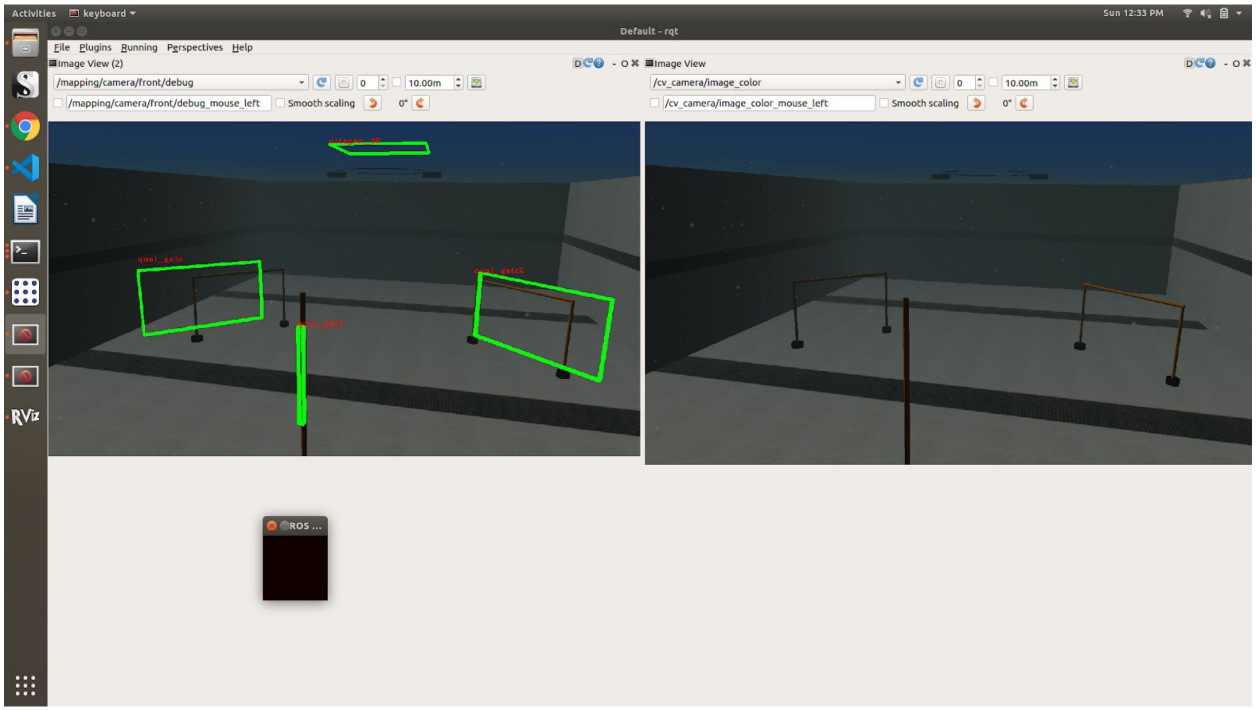


```
[1567362640.48] /do_onboarding_task/move_xy x: 6.00 y: -2.00
[1567362640.48] /do_onboarding_task Right Gate Task
[1567362624.13] /do_onboarding_task/dive 8.00m
[1567362623.13] /do_onboarding_task/wait_for_start/delay timer Waiting for 1 seconds
[1567362623.13] /do_onboarding_task/wait_for_start Mission start has been received
[1567362620.11] /do_onboarding_task/wait_for_start Waiting for mission start button
[1567362620.11] /do_onboarding_task/wait_for_start
[1567362620.11] /do_onboarding_task
[1567362619.11] /delay timer Waiting for 1 seconds
[1567362618.11] /delay timer Waiting for 1 seconds
```

Planner: onboarding

By selecting Plugins->Visualization->Image View, you can select a video feed to observe in rqt. The important ones for you are:

- “/cv_camera/image_rect”
    - This will display what Auri’s front camera sees
- “/mapping/front/camera/debug”
    - This will display the front camera with an overlay. The overlay projects cuboids over where Auri’s mapper thinks objects are. For you, this is mostly useful for remembering the names of objects, as their tags will be displayed.



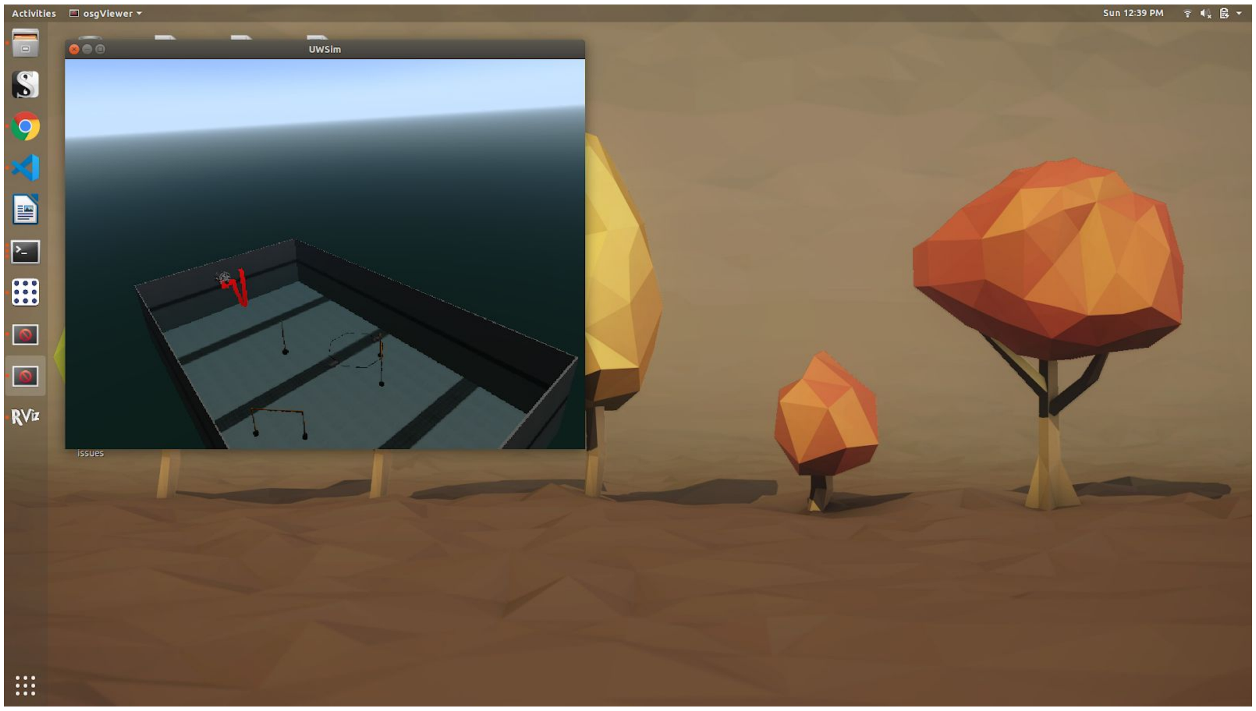
Notice the small square window on the above image. By hovering the mouse over this, Auri can be **controlled with your keyboard**. This window opens as part of the simulator. I like to place this on rqt along with my visual feeds for ease of access.

If there isn’t enough space on one instance of rqt for all of the plugins you want, you can open another instance of rqt in a new terminal with the following command:

```
rqt
```

This can be extremely helpful if used in conjunction with workspaces.

The simulator will open a medium sized window with an aerial view of the pool. It can be annoying to use for close range observation, so I tend to just use the image views in rqt.

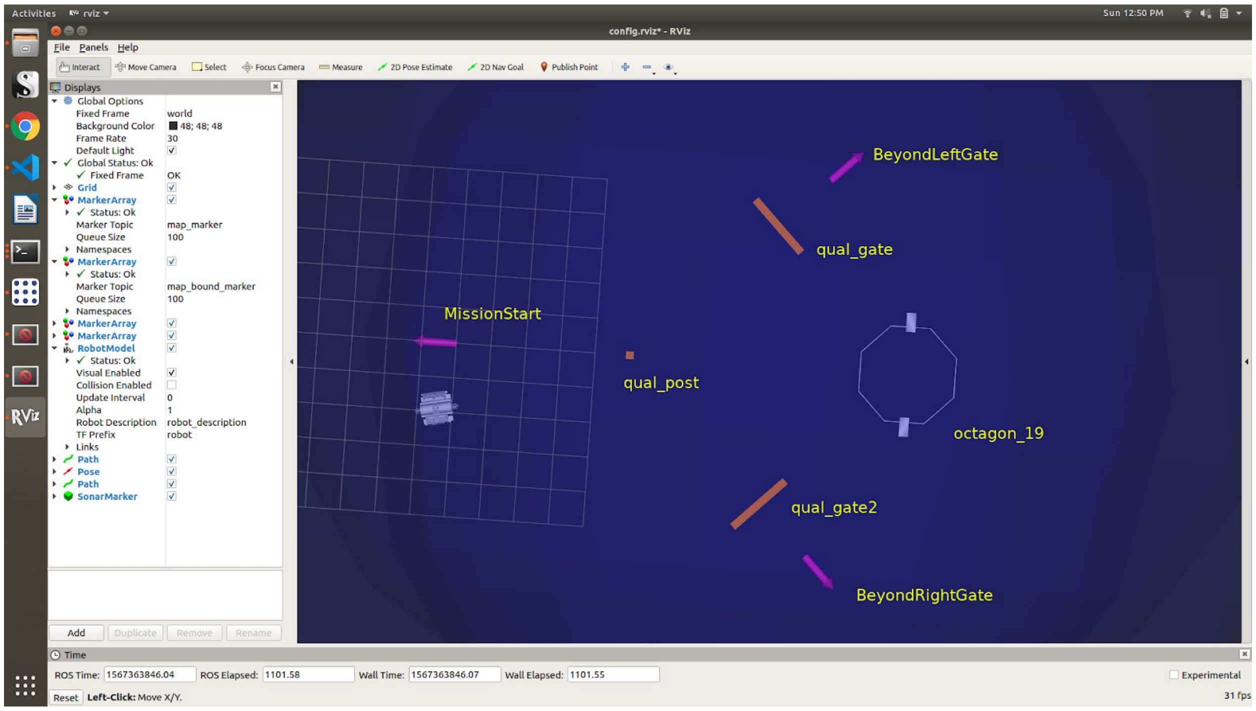


By clicking in the window, the view can be rotated. Press ‘h’ for an on screen help. One of the useful things about this window is that it displays a trail of Auri’s movement. This means that you could run your mission, walk away, and return to see how things went.

RViz is used to visualize Auri’s mapper. This is a node that keeps track of incoming messages and aggregates them into an understanding of where Auri is and where objects are. Since you do not need to use any computer vision detectors for your task, you will not be able to see objects move as their estimates change. The rendered props are only estimated positions, while the blue ellipsoids represent the total area where an object could be. If detectors are run, the blue ellipsoids will shrink as Auri becomes more confident about her surroundings.

What is important are the purple arrows. These are waypoints. They are direction vectors anchored to objects so that programming Auri to move relative to an object is simple. This means that if an object moves in the mapper, its associated waypoints will also move. You should avoid specifying absolute coordinates in your mission code, instead using offsets from object locations and waypoints.

In the mission planner, waypoints are treated transparently as actual objects. If “BeyondGate” represents the location in front of a gate, and MoveTo(“gate”) is valid, then MoveTo(“BeyondGate”) is equally valid.



Displays

Global Options	
Fixed Frame	world
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
MarkerArray	✓
✓ Status: Ok	
Marker Topic	map_marker
Queue Size	100
Namespaces	
MarkerArray	✓
✓ Status: Ok	
Marker Topic	map_bound_marker
Queue Size	100
Namespaces	
MarkerArray	✓
MarkerArray	✓
RobotModel	✓
✓ Status: Ok	
Visual Enabled	✓
Collision Enabled	
Update Interval	0
Alpha	1
Robot Description	robot_description
TF Prefix	robot
Links	
Path	✓
Pose	✓
Path	✓
SonarMarker	✓

Time

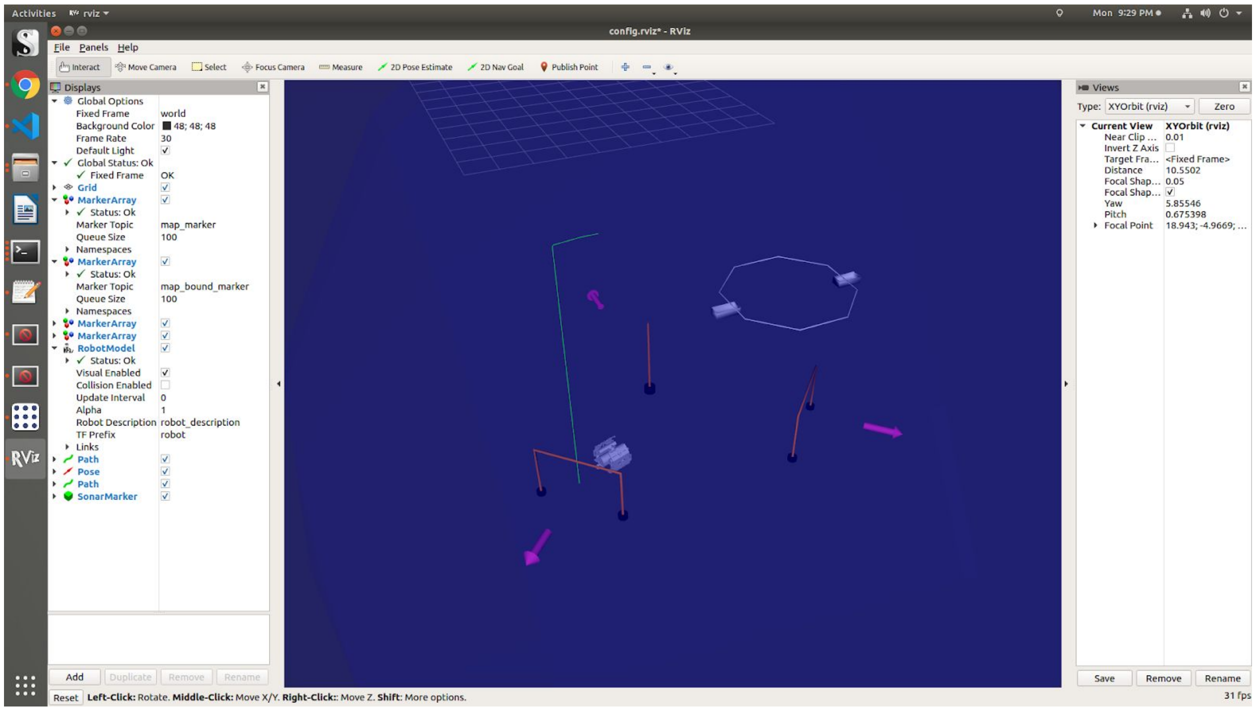
ROS Time: 1567363846.04 ROS Elapsed: 1101.58 Wall Time: 1567363846.07 Wall Elapsed: 1101.55

Left-Click: Move X/Y.

Notice that waypoints use CamelCase while objects use snake_case for their naming scheme.

RViz is also useful for monitoring Auri’s path if a mission has given a movement goal. The calculated route will be displayed as a line.

You can change the perspective in RViz by editing the “views” panel.



Global Options	
Fixed Frame	world
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
MarkerArray	✓
✓ Status: Ok	
Marker Topic	map_marker
Queue Size	100
Namespaces	
MarkerArray	✓
✓ Status: Ok	
Marker Topic	map_bound_marker
Queue Size	100
Namespaces	
MarkerArray	✓
MarkerArray	✓
RobotModel	✓
✓ Status: Ok	
Visual Enabled	✓
Collision Enabled	
Update Interval	0
Alpha	1
Robot Description	robot_description
TF Prefix	robot
Links	
Path	✓
Pose	✓
Path	✓
SonarMarker	✓

Type: XYOrbit (rviz) Zero

Current View	XYOrbit (rviz)
Near Clip ...	0.01
Invert Z Axis	
Target Fra...	<Fixed Frame>
Distance	10.5502
Focal Shap...	0.05
Focal Shap...	✓
Yaw	5.85546
Pitch	0.675398
Focal Point	18.943; -4.9669; ...

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Move Z. **Shift**: More options.

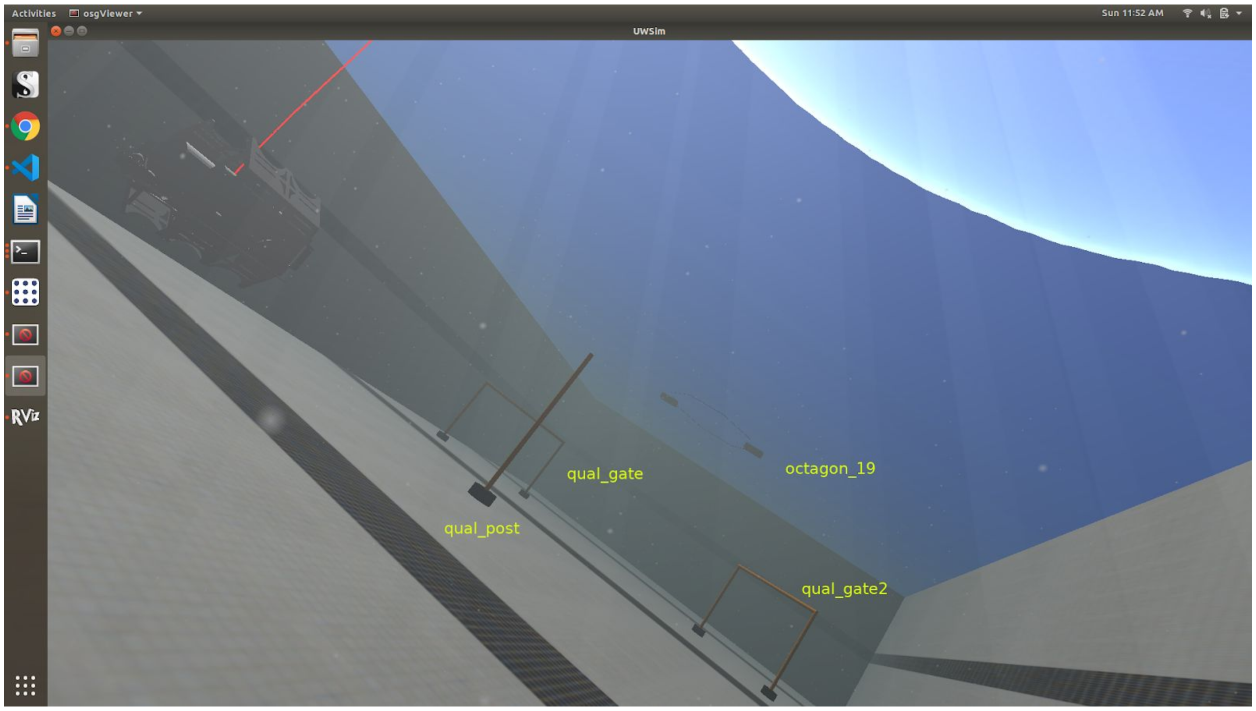
Now that you know the GUIs, it's time to run a mission. In the terminal, navigate to the window labelled "8 planner" and type the following command:

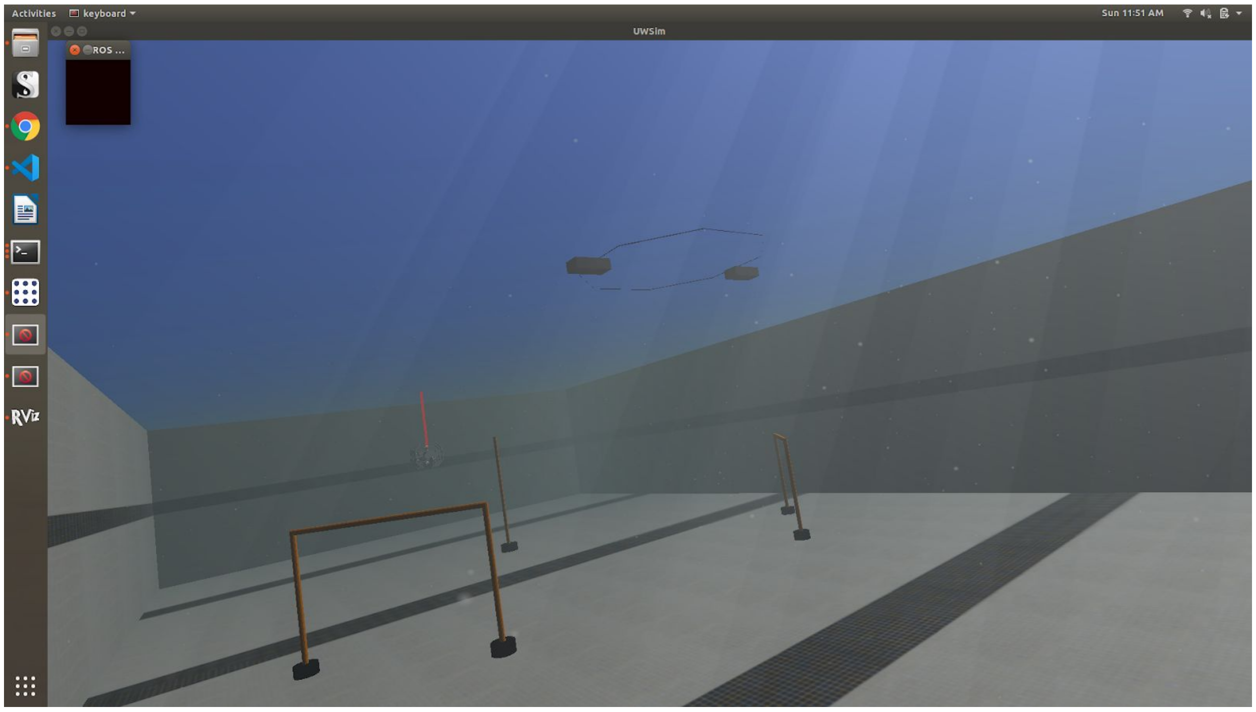
```
roslaunch au_planner <mission_name>.launch
```

For a complete list of missions that can be launched, see the folder au_planner/launch. To run the onboarding task, simply use "onboarding.launch". Give it a shot before you edit the onboarding.cpp file. This way you'll know that if something doesn't work at a later time, it's definitely your own mistake.

The Onboarding Course

A small pool is setup with a post, two gates, and an octagon.





Here’s what you need to program the robot to do:

1. Wait for the mission start button.
2. Dive.
3. Right Gate Task
    a. Move around the left side of the post, pass through the right gate, stop at the BeyondRightGate waypoint.
    b. Do a barrel roll.
4. Left Gate Task
    a. Move back through the right gate, move around the post, move through the left gate, stop at the BeyondLeftGate waypoint.
    b. Do a spin.
5. Octagon Task
    a. Move back through the left gate, stop between the post and octagon.
    b. Move to the octagon, surface within it, dive again.
6. Return Task
    a. Move to the left side of the post, move in a circle around it, rotating towards the octagon, stopping at the side closest to the starting point.
    b. Move to the MissionStart waypoint.
7. Success! The robot will automatically surface when the mission exits.

Rules:

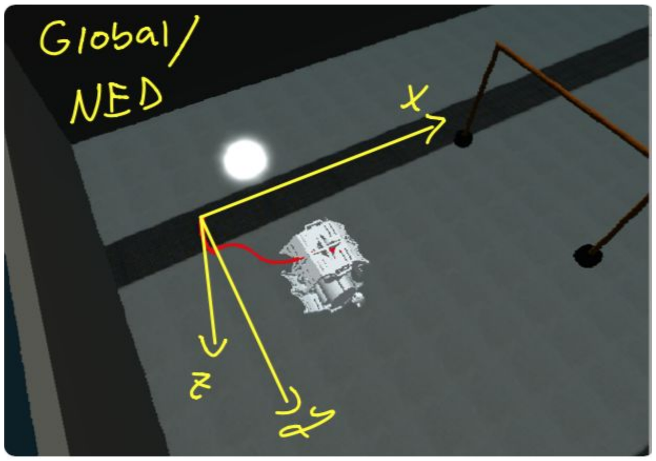
- Auri cannot touch any of the props / pool floor. If LQR is unstable, Auri may move quite a bit, so slight clipping may be tolerated. This unstable movement is not concerning, we probably just haven’t recently tuned our control algorithm.
- Auri cannot surface outside of the octagon or unless ending the mission.
- You must use the command STATE_TRACE(nctx, "MyComment"); to log Auri’s progress to the planning GUI. It is sufficient to log when each of the main tasks commence:
    - Right Gate Task
    - Left Gate Task
    - Octagon Task
    - Return Task
- You must use waypoints and object locations. You may not use absolute positioning. I.e. you cannot manually enter a series of (x,y) pairs to create a route. You may use locations in the spirit of (gate_x - 3, gate_y + 5), etc.
- You must use the gen_circle_waypoints() function in conjunction with a loop when circling the post on step 6a. You may move more granularly around the post on other steps.

- You must follow the style guide for C++. Just look at some existing code and try to mimic it. You should run au_everything/scripts/reformat.sh to make your code conform, but this will not fix some issues such as incorrect naming conventions.
- State machine functions must be wrapped in OUTCOME_TRY() like in existing missions.
- All of the necessary functions are provided so you are not permitted to make additional functions.
- Please comment your code and handle errors.

Also **please note** collaboration is absolutely 100% **PERMITTED**. The point of this challenge is not as much of a test of your skill as it is an exercise in familiarizing you with working with our systems. If you get stuck please ask any member of the software team and they should be able to help. One other note, it goes without saying that you cannot just copy someone else's solution to this challenge.

A few points to help you speed along:

- It can take a bit for the robot to move, so it'll be faster to test certain maneuvers at the start point rather than waiting. E.g. try rolling at the start point.
- A depth of 9m is adequate.
- We use North-East-Down (NED) as our world coordinate system such that:
    - Positive x is towards the back of the pool.
    - Positive y is towards the right of the pool.



- It's not necessary to have a long delay after the mission start button is pressed.
- You are permitted to copy code from existing missions. However, it is expected that you understand what you are using. There should be no redundant code.
- Time units in the planner are in seconds, angular measures are in radians.

- Use constants for values that are used repeatedly. E.g. assign a value to a timeout variable and use it instead of manually typing (and potentially having to change) a single number in many places.

When you have completed the mission task, please push it to your own fork of ARVP's software repository. Contact Nicholas Wengel or Willard Farmer when you've done so and we can take a look. We look forward to working with you!

Submitting your Completed Task / Your First Pull Request

Once you've completed the mission task you need to push it to the onboarding copy of the software repository (au_everything_onboarding). This will give you an opportunity to emulate the normal development cycle:

- Write code
- Push to remote and create a PR
- Request a review from other developers most familiar with your feature's subject
    - E.g. If you're working on sonar and Bob is the senior sonar guy, you probably want to ask him instead of someone who's never looked at the sonar code.
- Make changes as requested by reviewers, until they approve your PR
- Squash and merge your branch
- Delete it (this will actually archive it)

For the mission task, Nicholas Wengel and Willard Farmer should be your reviewers.

Submissions made to your own repositories and linked will not be accepted since the purpose of this is to emulate the end-to-end development cycle.

Additional Information About the Planner

Making you read over the entire au_planner subrepo would take a while and wouldn't be all that helpful to the immediate goal of completing the mission task. Though you're perfectly welcome to inspect things for a deeper understanding of the code, here's a complete list of all the functions you might need. You may use other existing functions if you desire to do so.

Note that the following code excerpts are not meant to be copied and pasted. These are the function declarations, not example usages. If you want to use one of these functions, try searching through existing missions.

For additional documentation, see the README.md inside of the au_planner subrepo.

Logging Messages

```
auto nctx = ctx.with_trace("my message");
STATE_TRACE(nctx);
```

The planner uses a context to keep track of timers and such, which is why it's passed to most of the state machine's functions. Part of our design is that the context is never modified in-place, and can only be copied.

The .with_trace() function creates a copy of the context, while recording that a new state is being entered. The STATE_TRACE() function performs the actual logging to the terminal.

Outcome

```
OUTCOME_TRY(sm_func())
```

You'll notice that we wrap all of our state machine calls in OUTCOME_TRY(). This is a third party function that we use to ensure that errors are handled properly.

Status Results

```
#include <au_planner/sm_util_actuator.h>
```

```
template <typename T>
outcome::result<T> show_status(const MContext& ctx, outcome::result<T>
res)
```

This function is used to wrap state machine calls so that their return result will be printed to the terminal. The normal behavior is for state machine calls to return silently. You don't need to print return values, but this might be useful for debugging.

Mission Start Button

```
#include <au_planner/sm_util_wait_for_start.h>
```

```
outcome::result<void> sm_wait_for_start(const SMContext &ctx, int
delay, bool use_button)
```

This function will block until the mission start button is pressed. For untethered tests, Auri has a physical button, but you can use the one on the controller GUI. Once the button is pressed, Auri will delay her next action for the specified number of seconds.

Waypoint / Object Locations

```
#include <au_planner/sm_util_waypoints.h>
```

```
auto qual_post = nctx.ros.in->get_object("qual_post");
```

The ->get_object() function is used to return data about either a waypoint or an object. The position can be extracted from the returned data like so:

```
qual_post->position.x()
qual_post->position.y()
```

Timeouts

Many state machine calls have a timeout parameter. If an action takes too long to complete, it can timeout, allowing the next action to initiate. This is helpful for things like movement, where the action may not register as complete unless the robot reaches within a certain error tolerance of the target. If LQR is unstable, it could take a while to stabilize to the exact target. This is very useful in competition when time is precious and we don’t want the robot to get stuck on one component of any mission.

The default behavior for timed out states is to return a failure code. This means that, by default, the entire mission will fail if a single state doesn’t complete in time. E.g. if a movement takes too long. To prevent this behavior, you can pass an IGNORE_TIMEOUT flag to states.

```
OUTCOME_TRY(sm_move_path(nctx, timeout, width, 0, depth, M_PI_2,
nctx.config.map_frame, IGNORE_TIMEOUT));
```

Movement

There are a number of movement commands, but the easiest one to use is:

```
#include <au_planner/sm_util_path.h>
```

```
outcome::result<void> sm_move_path(const SMContext &ctx, int timeout,
double x, double y, double z, double yaw, std::string frame_id,
uint32_t flags)
```

```
ctx.ros.in->robot_pose().z()
ctx.config.map_frame
```

This will move Auri to a location specified by an (x,y,z) pair, using a specified yaw. The exact path is handled automatically in a movement planning node. You'll need to specify the depth as well as the coordinate frame. The map frame and the current depth can be acquired with the above expressions. The yaw angle is in radians.

```
outcome::result<void> sm_move_to_object(const SMContext& ctx, int
timeout, const std::string& obj_name, double threshold, double
epsilon, uint32_t flags)
```

This will make Auri move to a specified object or waypoint.

```
#include <au_planner/sm_util_control.h>
```

```
outcome::result<void> sm_dive(const SMContext &ctx, int timeout,
double depth, uint32_t flags = 0)
```

This will make Auri dive to specified depth.

```
#include <au_planner/sm_util_rotations.h>
```

```
outcome::result<void> sm_barrel_roll(const SMContext& ctx, int
timeout, double ang_vx, double lin_vx, uint32_t flags)
```

This will make Auri do a barrel roll (roll 360 degrees). See au_planner/src/sm_gate_roll.cpp for example parameters.

```
#include <au_planner/sm_util_rotations.h>
```

```
outcome::result<void> sm_spin(const SMContext& ctx, int timeout,
double ang_vz, uint32_t flags)
```

This will make Auri do a spin (yaw 360 degrees). See au_planner/src/sm_gate_roll.cpp for example parameters.

```
#include <au_planner/sm_util_waypoints.h>
```

```
std::vector<Eigen::Vector3d> gen_circle_waypoints(const
Eigen::Vector3d& centre, double radius, double step_angle, double
start_angle, double end_angle)
```

This will generate a sequence of points representing a circular path around the specified center. These points can then be used in the typical sm_move_path() function using a loop. See the qualification mission for an example of how to use this.

Course Waypoint Names and Object Names

Here are the names to use when referencing positions in the course:

- qual_post
    - The orange post
- qual_gate
    - The orange gate on the left side of the pool
- qual_gate2
    - The orange gate on the right side of the pool
- octagon_19
    - The black octagon floating on the pool surface
- MissionStart
    - A waypoint representing Auri's initial position
- BeyondLeftGate
    - A waypoint representing the area behind the left gate
- BeyondRightGate
    - A waypoint representing the area behind the right gate

Example Video

URL

This video shows a few things:

- Where the sm_onboarding.cpp file is located. This is the source file that needs to be modified to complete the mission task.
- How to compile the repository with the catkin_make command. Note that this must be called within the au_everything/catkin_ws/ directory.
- How to launch the software stack, which appears as a grid of small terminals in a tmux session.
- How to position windows within Ubuntu workspaces for an easier workflow.
- How to open rqt, the ARVP plugins, and visualization plugin. As well as which visualization topics are relevant, resizing plugins, moving plugins.
- How to start the planner in the terminal.
- Where the mission start button is within the rqt GUI.
- How to use RViz
  - Orthographics and perspective views
  - Hiding/showing information
- How to use the simulator window
  - Display help
  - Toggle lighting
  - The keyboard window
- An exemplar of the mission task