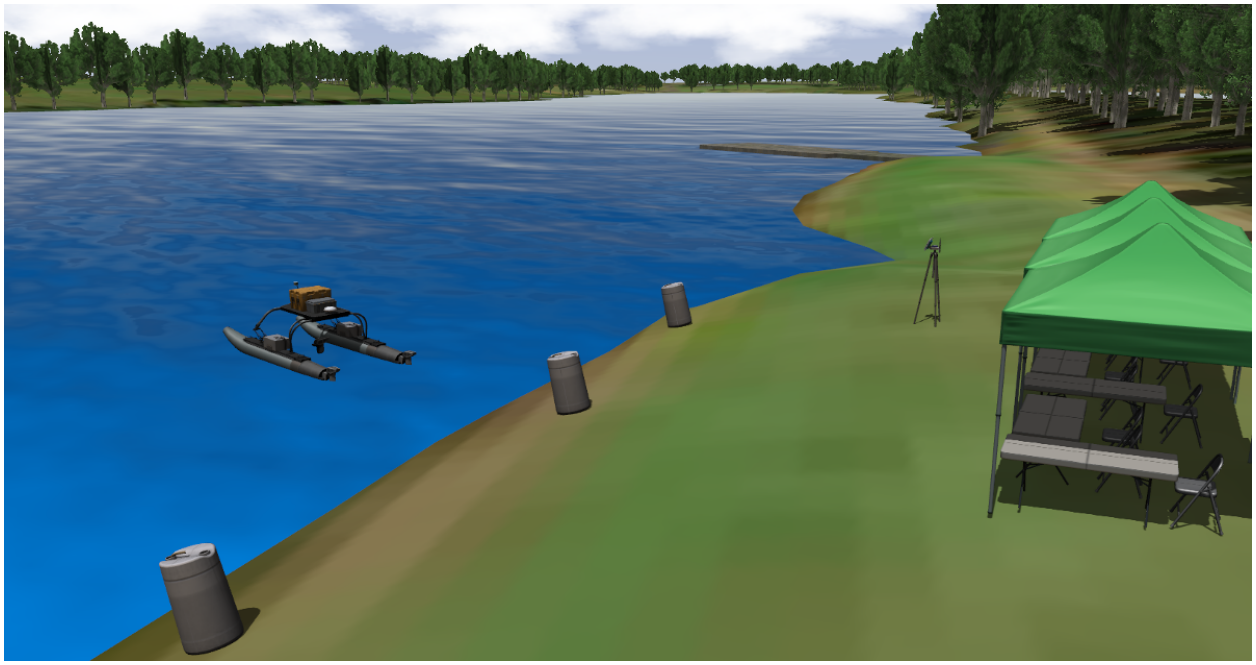
# RBOT 280/281: Capstone 1/2

## Introduction:

For my capstone project I will be working in the simulation environment of Gazebo and with the ROS robotic middleware for message communication as well as programmatic algorithmic development.

The project will follow the setup of the modules with regards to the syllabus, such that the scope of the project represents the stated goals of the capstone project. The final report will incorporate all pertinent information.

The Open Source Robotics Foundation (OSRF) has developed a simulated marine environment and robotic marine vehicle, the WAM-V, for the Virtual Robot-X competition. I will be using this simulated marine environment and robotic vehicle within the Gazebo simulation software. I will be interfacing with the robot and through the Robot Operating System (ROS), which is a robotic middleware used for inter-robot communications.



# Module 1: Basic Setup

## Module Goals:

- Download and compile software
    - Gazebo 11
    - ROS Noetic
    - VRX environment and robot model
- Run basic simulation environment
- Configure the vessel
- Interface with the robot
    - Keyboard control
    - Programatically
- Make video clip of work

## Module Notes:

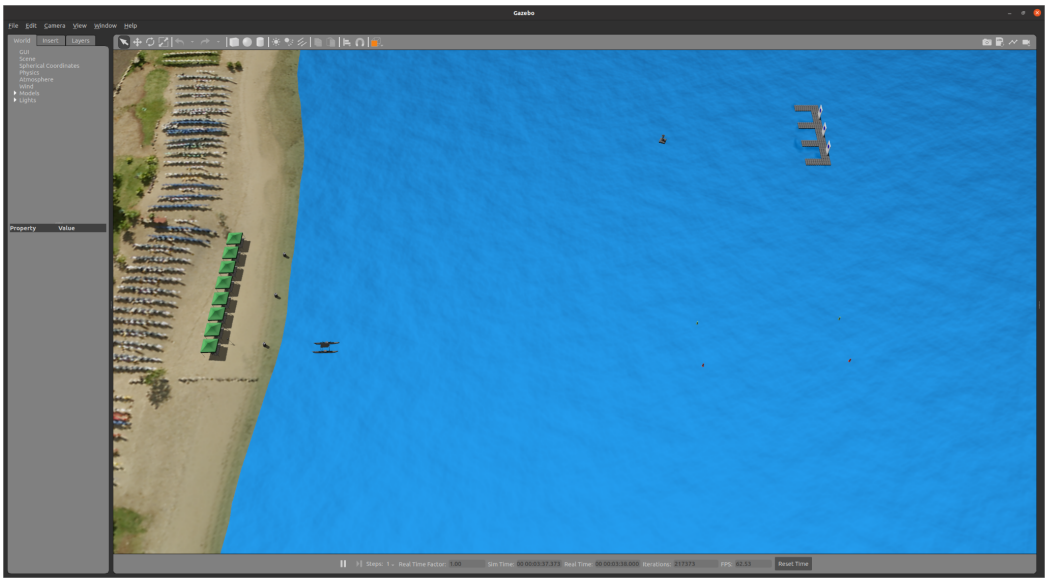
### Download and compile software:

The first step I took was to upgrade my desktop to Ubuntu 20.04 and download the messaging framework, ROS Noetic, as well as the simulation environment, Gazebo 11. I then downloaded the Virtual Robot-X (VRX) simulation environment and built the packages in ROS.

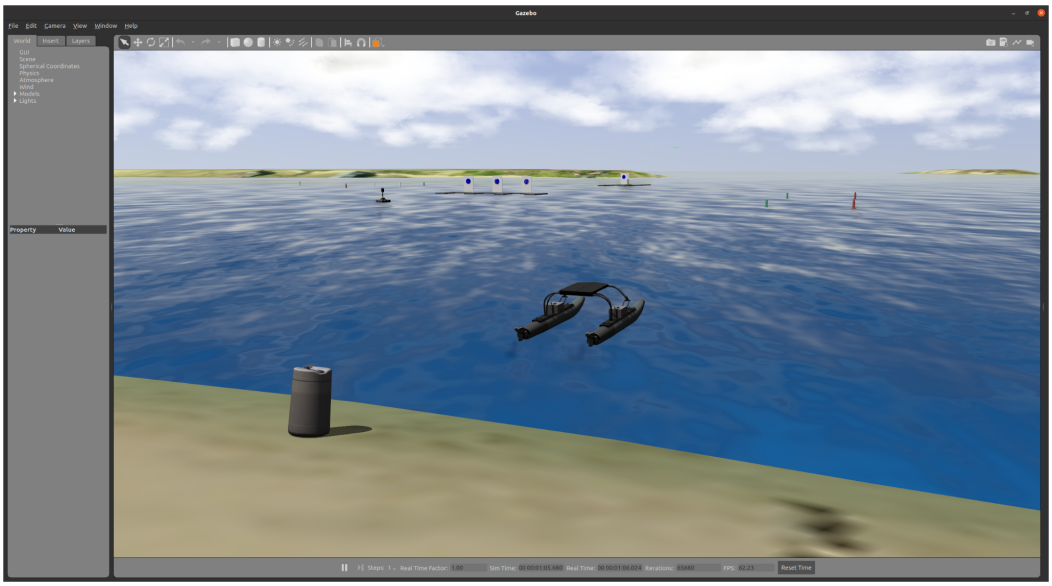
### Run basic simulation environment:

I tested the basic “hello world” tutorial [1] and got the following output.

- $ roslaunch vrx_2019 sandisland.launch



Birds eye view of a portion of the test environment



Hello world of the WAM-V (marine vessel) and simulation environment

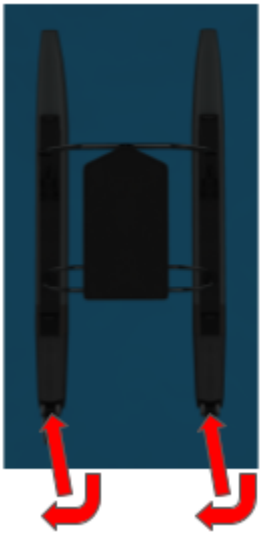
Close up of the WAM-V robot in the simulation environment

## Configure the vessel:

First I created a custom URDF file for my robot using a script designed to import settings for the thrusters and components (sensors) [2]. A URDF file is a format to describe a robot and will be used by Gazebo, rviz, and other ROS packages.

### Thruster configuration:

The robot will be a differentially controlled vehicle with two single fixed thrusters at the end of each pontoon. This is described as the “H” Differential Thrust setting [3].



Differential thrust of the robot

### Adding sensors to the vessel:

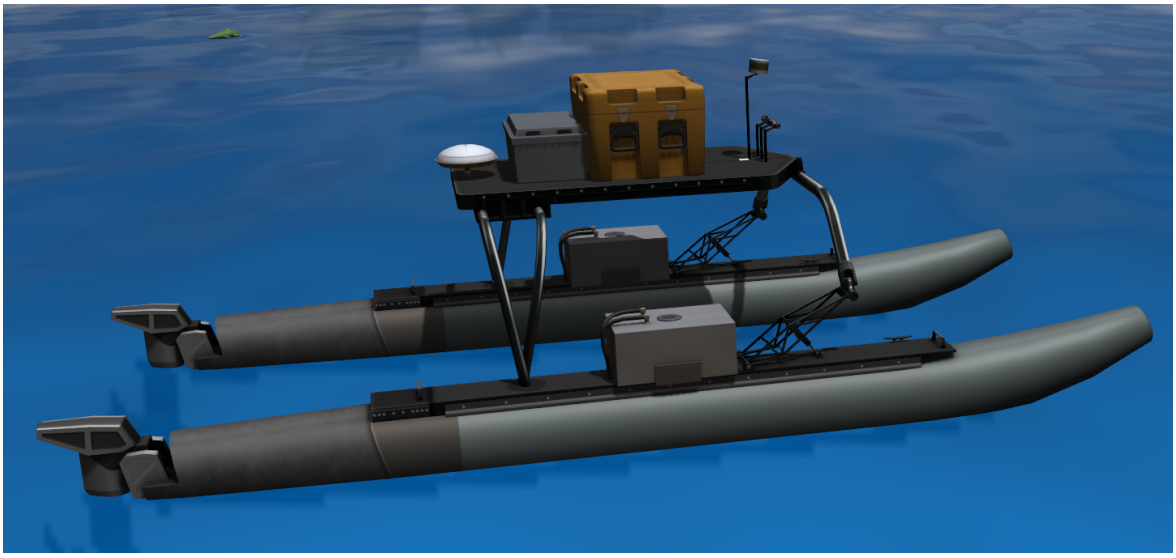
I am configuring the robot to have a lidar for 360° HFoV with a range of 130 meters, three forward looking cameras with a total horizontal field of view (HFoV) of 200°, an IMU and GPS for the robots pose and position. HFoV for each camera is 80° defined in vrx/wamv_gazebo/urdf/components/wamv_camera.xacro

- $ roslaunch vrx_gazebo generate_wamv.launch thruster_yaml:=$PWD/my_wamv/thruster_config.yaml component_yaml:=$PWD/my_wamv/component_config.yaml wamv_target:=$PWD/my_wamv/my_wamv.urdf

Launch the simulation with the custom robot configuration

- $ roslaunch vrx_gazebo sydneyregatta.launch urdf:=$PWD/my_wamv/my_wamv.urdf

Custom robot front view



Custom robot side view

## Interface and control the robot:

The plan is to control the robot differentially. I do not plan on directly controlling the angle of thrust, only the amount of thrust to each motor. The following ROS topics will control each motor.

```
/wamv/thrusters/left_thrust_angle
/wamv/thrusters/left_thrust_cmd
/wamv/thrusters/right_thrust_angle
/wamv/thrusters/right_thrust_cmd
```

ROS thruster topics

Teleoperation: Controlling the vehicle using the keyboard [4]

The ROS teleop_twoist_keyboard package, along with a custom twist2thrust.py node are used to convert Twist messages to two Float32 messages for the left and right thrusters [5].

- right_thrust_cmd = Twist.linear.x + Twist.angular.z
- left_thrust_cmd  = Twist.linear.x - Twist.angular.z

Launch the teleoperation scripts

- $ roslaunch vrx_gazebo sydneyregatta.launch urdf:=$PWD/my_wamv/my_wamv.urdf
- $ roslaunch vrx_gazebo usv_keydrive.launch

Keys to move the robot:

- The i /, keys command forward/reverse velocity (twist.linear.x) which is mapped to axial thrust (right+left)
- The j / l keys command counter-clockwise/clockwise rotational velocity (twist.linear.z) which is mapped to differential thrust (usvdrive.right-usvdrive.left)

Data published to left/right thruster to command the robot

- Forward: 0.5 / 0.5
- Reverse: -0.5 / -0.5
- Left: -1.0 / 1.0
- Right: 1.0 / -1.0

Programmatic operation: Controlling the vehicle by publishing messages to the appropriate ROS topics [6]

ROS messages of type std_msgs/Float32 are used to describe each thruster input. Each command is a normalized value between -1.0 to 1.0, where 1.0 is maximum forward force and -1.0 is maximum reverse force. Thrust commands are published on the following ROS topics

- Left thrust: /wamv/thrusters/left_thrust_cmd
- Right thrust: /wamv/thrusters/right_thrust_cmd

Publishing to one of these topics in a positive value will rotate the robot about the z-axis. This will be the method of publishing data when the robot will be moving autonomously. Below is an example to move the robot for a single second publishing to the right thruster for 1 Hz.

- $ rostopic pub --once /wamv/thrusters/right_thrust_cmd std_msgs/Float32 "data: 1.0"

A second method of programmatically thrust the robot is to publish to the selected topic through the rqt_publisher package. This provides a convenient graphical tool for testing commands.

## Video Links:

**Please note that sometimes the recording software does not work with Mac. Please view directly in the browser if that is an issue.**

1. Teleoperation of the robot: https://www.dropbox.com/s/cob9zbf5g1tzv8n/2022-01-16%20_robot-teleop.mp4?dl=0
2. Programmatic operation of the robot:
    a. Publishing to the thrust command topic via the command line interface https://www.dropbox.com/s/78fnio53w0x8oh9/2022-01-16_programatic-operation.mp4?dl=0
    b. Publishing to the thrust command topic via rqt_publisher https://www.dropbox.com/s/tgsm4778px7vznz/2022-01-16_rqt-publisher-operation.mp4?dl=0

# References (Module 1)


1. https://github.com/osrf/vrx/wiki/sand_island_tutorial
2. https://github.com/osrf/vrx/wiki/tutorials-Creating%20a%20custom%20WAM-V%20Thruster%20and%20Component%20Configuration%20For%20Competition
3. https://github.com/osrf/vrx/wiki/tutorials-PropulsionConfiguration
4. https://github.com/osrf/vrx/wiki/driving_teleop_tutorial
5. https://github.com/osrf/vrx/wiki/tutorials-keydrive_details

# Module 2: SLAM Mapping

## Module Goals:

- Use joystick app to steer the robot and create maps of environment
- Create rviz file to visualize the robot and the sensors that are simulated with Gazebo
- Make video clip of work

## Module Notes:

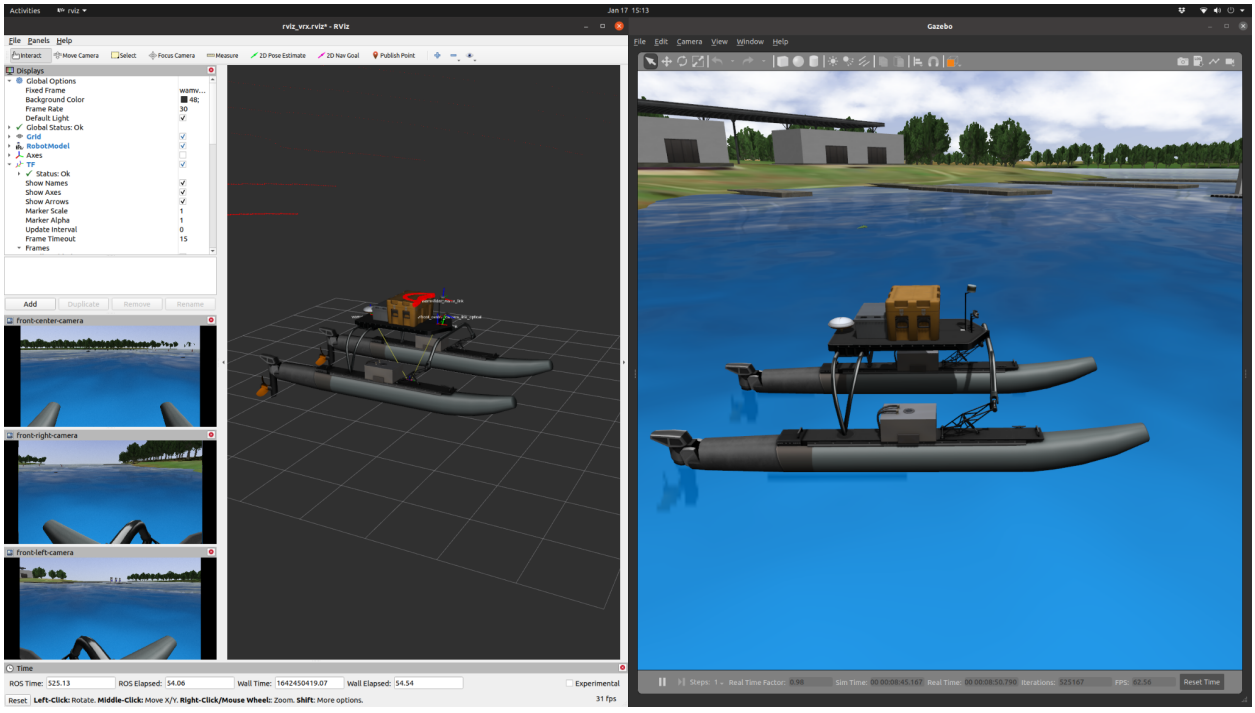
Continuing with my simulated water environment and unmanned surface vessel I will attempt to map the part of the environment that has obstacles in it. This turns out to be a very large area, but is not the whole simulated world as the rest of the environment I am not interested in exploring.

### RVIZ

Rviz is used to visualize the robot and the output of the sensors being simulated with Gazebo. I have created a launch file to launch a custom rviz gui with the output of the sensors enabled.

I created a saved .rviz configuration file to load into RVIZ to visualize the robot model and sensor (three cameras and lidar) outputs  of the Gazebo simulation.

- $ roslaunch rbot280 wamv_vrx.launch



Left: RVIZ output, Right: Gazebo simulation

## Create rbot280.launch script

Next I combined all relevant launch files into a single launch script. Now the rbot280.launch file launches gazebo, the custom robot model, rviz with the custom configuration, as well static joint transforms to /tf, revolute joint static transforms to /tf. I also added the robot_localization package to the launch file. The robot_localization package is used to fuse the GPS and IMU into a Kalman filter odometry message. The pointcloud_to_laserscan package is added to take the 3D pointcloud and create a 2D laserscan message for SLAM mapping.

### Created a new ROS package to manage rbot280 work called… rbot280

- rbot280/launch/:
    - Gazebo simulation with the example_course.world and custom robot (rbot280_bringup.launch)
    - Custom robot (rbot280/config/my_wamv.urdf)
    - Custom rviz config (wamv_vrx.launch)
    - Static joint transforms
    - Revolute joint static transforms
    - Custom robot_localization setup (wamv_localization.launch)
    - Pointcloud_to_laserscan (pointcloud-to-laserscan.launch)

## Map_Server ROS Package

Maps manipulated by the tools in this package are stored in a pair of files. The YAML file describes the map meta-data, and names the image file. The image file encodes the occupancy data.

### Image

The image describes the occupancy state of each cell of the world in the color of the corresponding pixel. In the standard configuration, whiter pixels are free, blacker pixels are occupied, and pixels in between are unknown

### YAML

Required parameters describing the image in the yaml file.

- **image** : Path to the image file containing the occupancy data; can be absolute, or relative to the location of the YAML file
- **resolution** : Resolution of the map, meters / pixel
- **origin** : The 2-D pose of the lower-left pixel in the map, as (x, y, yaw), with yaw as counterclockwise rotation (yaw=0 means no rotation). Many parts of the system currently ignore yaw.

- **occupied_thresh** : Pixels with occupancy probability greater than this threshold are considered completely occupied.
- **free_thresh** : Pixels with occupancy probability less than this threshold are considered completely free.
- **negate** : Whether the white/black free/occupied semantics should be reversed (interpretation of thresholds is unaffected)

## SLAM

My goal for SLAM is to use the GMapping ROS package to create a map of the local area with the Velodyne Laser. Another alternative is the Hector-SLAM package which doesn't require odometry data. SLAM (Simultaneous Localization and Mapping) is used to create an occupancy grid map. The overall goal of this map is to use it to help the robot localize, or to find its pose. A pose is a representation of the robot in free space, consisting of position and orientation.

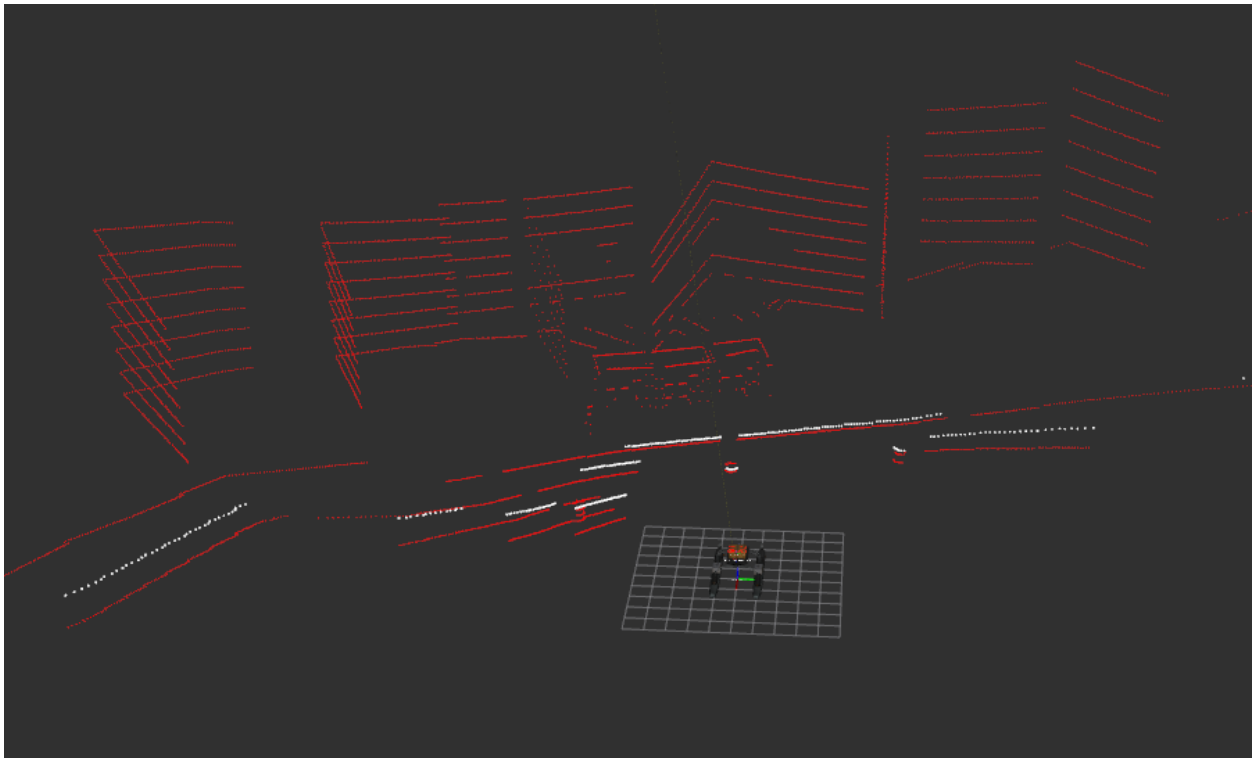
Gmapping is an efficient particle filter to learn grid maps from laser range data. The ROS GMapping package is a tool that uses the OpenSlam software library. To use slam_gmapping, you need a mobile robot that provides odometry data and is equipped with a horizontally-mounted, fixed, laser range-finder. The slam_gmapping node will attempt to transform each incoming scan into the odom (odometry) tf frame.

### ROS Packages to Install:

1. gmapping [1]
    a. $ sudo apt-get install ros-noetic-gmapping
2. pointcloud_to_laserscan [2]
    a. $ sudo apt-get install ros-noetic-pointcloud-to-laserscan
3. navigation [3] (note note actually needed until module 4)
    a. $ sudo apt-get install ros-noetic-navigation
4. robot_localization [4]
    a. $ sudo apt-get install ros-noetic-robot-localization

### pointcloud_to_laserscan ROS package

Gmapping subscribes to the transform topic (tf/tfMessage) and the laser scan topic (sensor_msgs/LaserScan). The LaserScan message [/scan] is a 2D laser message that is not supported currently by the Velodyne lidar in use onboard the robot. There is a ROS package 'pointcloud_to_laserscan' that takes the pointcloud message of the Velodyne lidar and converts that to the required LaserScan message. The pointcloud_to_laserscan package has many parameters to be set. I based my parameters off the lidar (Velodyne VLP-16) to make sure the node worked as needed.

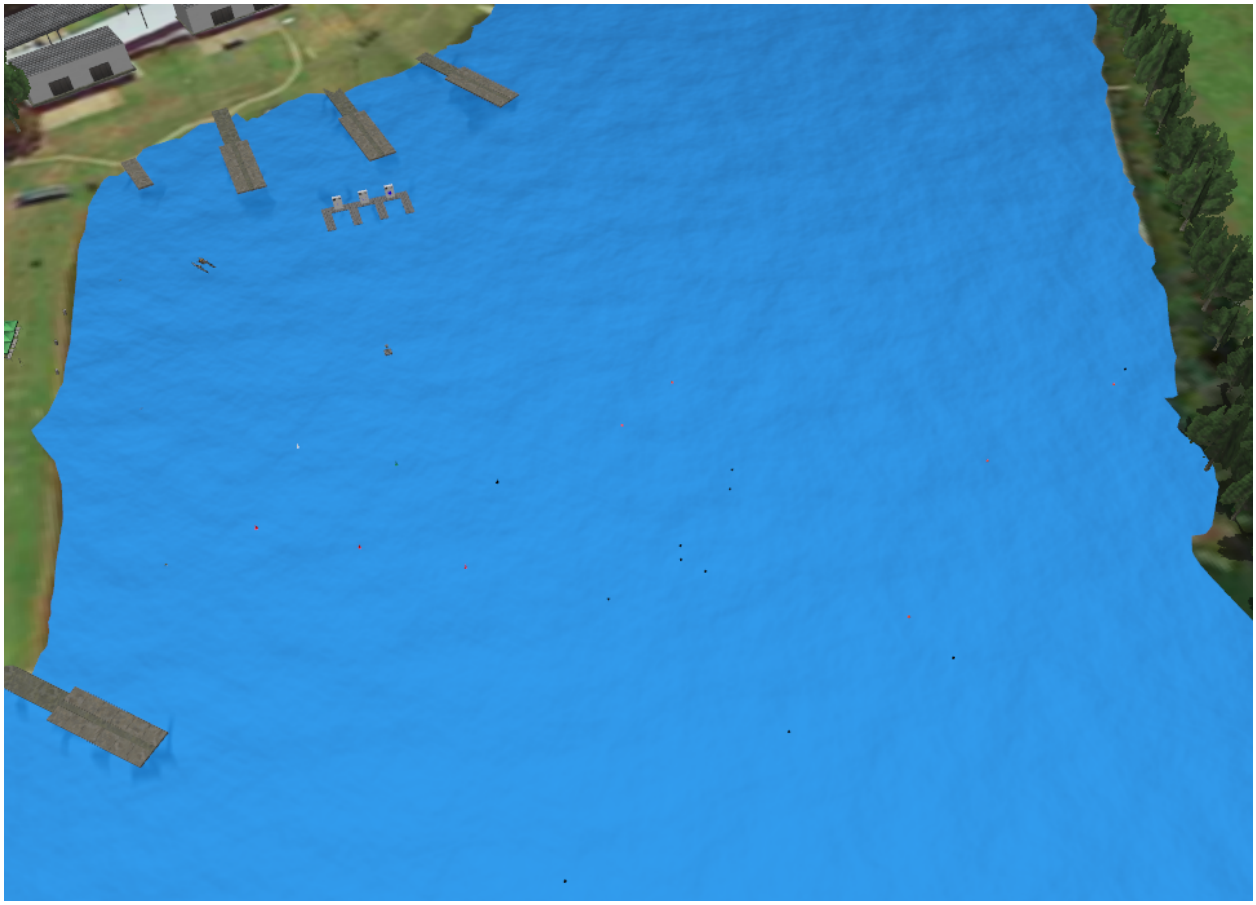


Red is the raw pointcloud output from the Velodyne Lidar and white is the 2d laserscan output from the pointcloud_to_laserscan package used in Gmapping

### gmapping command line steps:

1. Launch simulation and rviz gui, robot localization, joint state publishers
    a. $ roslaunch rbot280 rbot280_bringup.launch
2. Launch gmapping
    a. $ roslaunch rbot280 wamv_slam.launch slam_methods:=gmapping
    b. This launches all of the joint state transforms needed to determine the transforms from each coordinate frame to each component on the robot. It also internally launches the pointcloud_to_laserscan node as well as the gmapping node. The gmapping node has a configuration file that is set separately to set each parameter for this node.
3. Launch Localization
    a. $ roslaunch rbot280 wamv_localization.launch
4. Launch Teleoperation
    a. $ roslaunch vrx_gazebo usv_keydrive.launch
5. Save map using map_server [6]
    a. $ rosrun map_server map_saver -f <map-name>

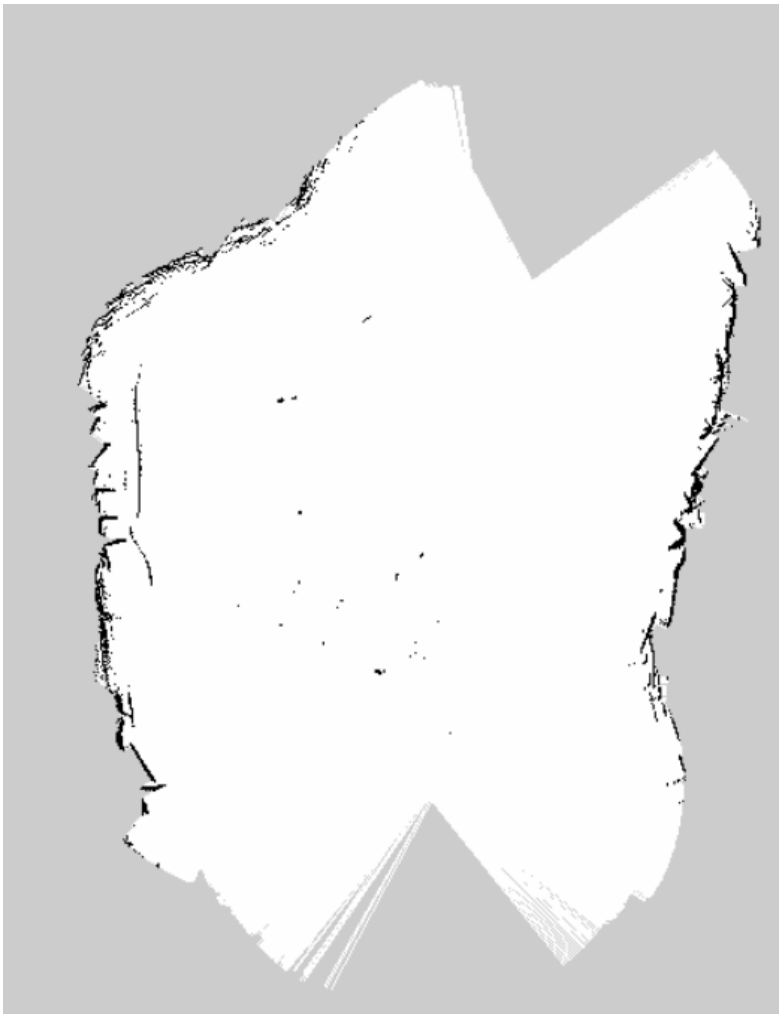
Area of interest to be mapped



Sand Island Birds Eye View

I had to change a lot of the default parameters in the gmapping_params.yaml file to get any sort of map to output to the screen. Some of the notable parameters to change were:

- delta: Resolution of the map (in meters per occupancy grid block)
    - The default is set way to low and cause zero obstacles to show up during the mapping process
- minimumScore: Minimum score for considering the outcome of the scan matching good.
    - The default (0.0) is not sufficient to make a map in an open area environment.
    - I found that when the vehicle is moving to fast the scan matching ability was not very accurate
- LinearUpdate: Process a scan each time the robot translates this far
    - The default (1.0) was too low for the robot's linear speed. I turned this up to 2.0.
- angularUpdate: Process a scan each time the robot rotates this far
    - The default (0.5) was too low for the robot’s angular speed and I turned this up to 0.9.



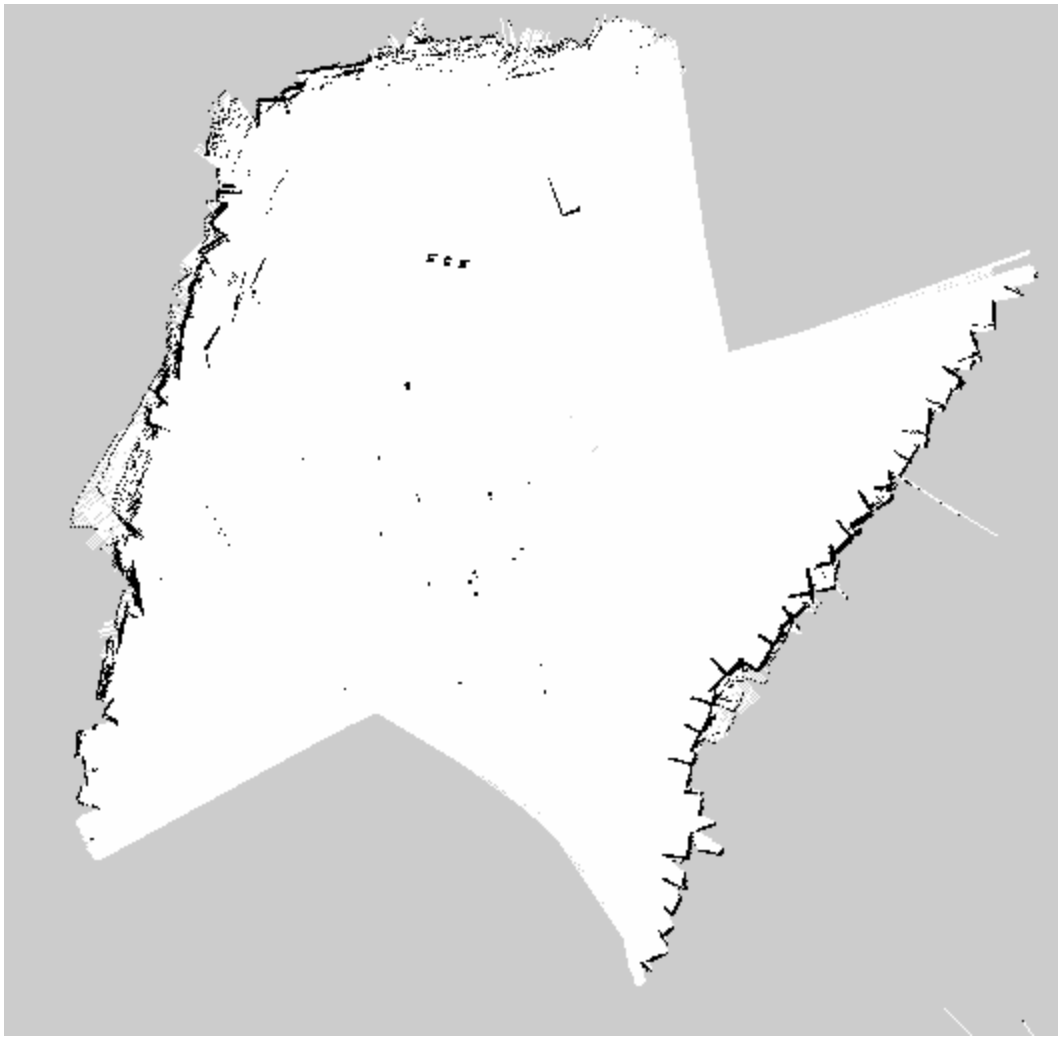
Gmapping output map

While the above map was not so bad, as I was driving around you could see some obstacles winking in and out.

Sometimes the odometry and laser output are not working and you get a message like this:

```
************************************************************************
** The Odometry has a big jump here. This is probably a bug in the   **
** odometry/laser input. We continue now, but the result is probably **
** crap or can lead to a core dump since the map doesn't fit.... C&G **
************************************************************************
```

I was not aware of this issue as I wasn't watching the launch of the gmapping node. In the end I think this was causing a bad map to be produced. I ended up waiting for the transforms to launch in a script earlier than the localization package which relies on the transform to give an odometry reference.



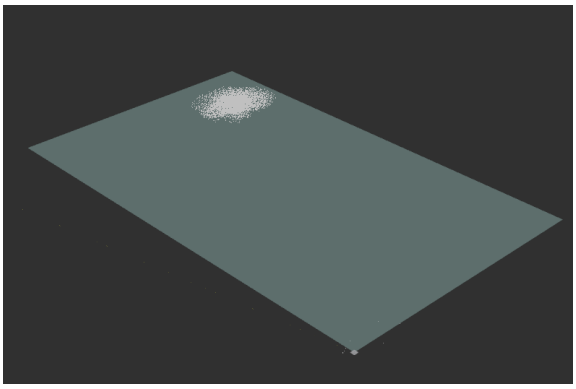
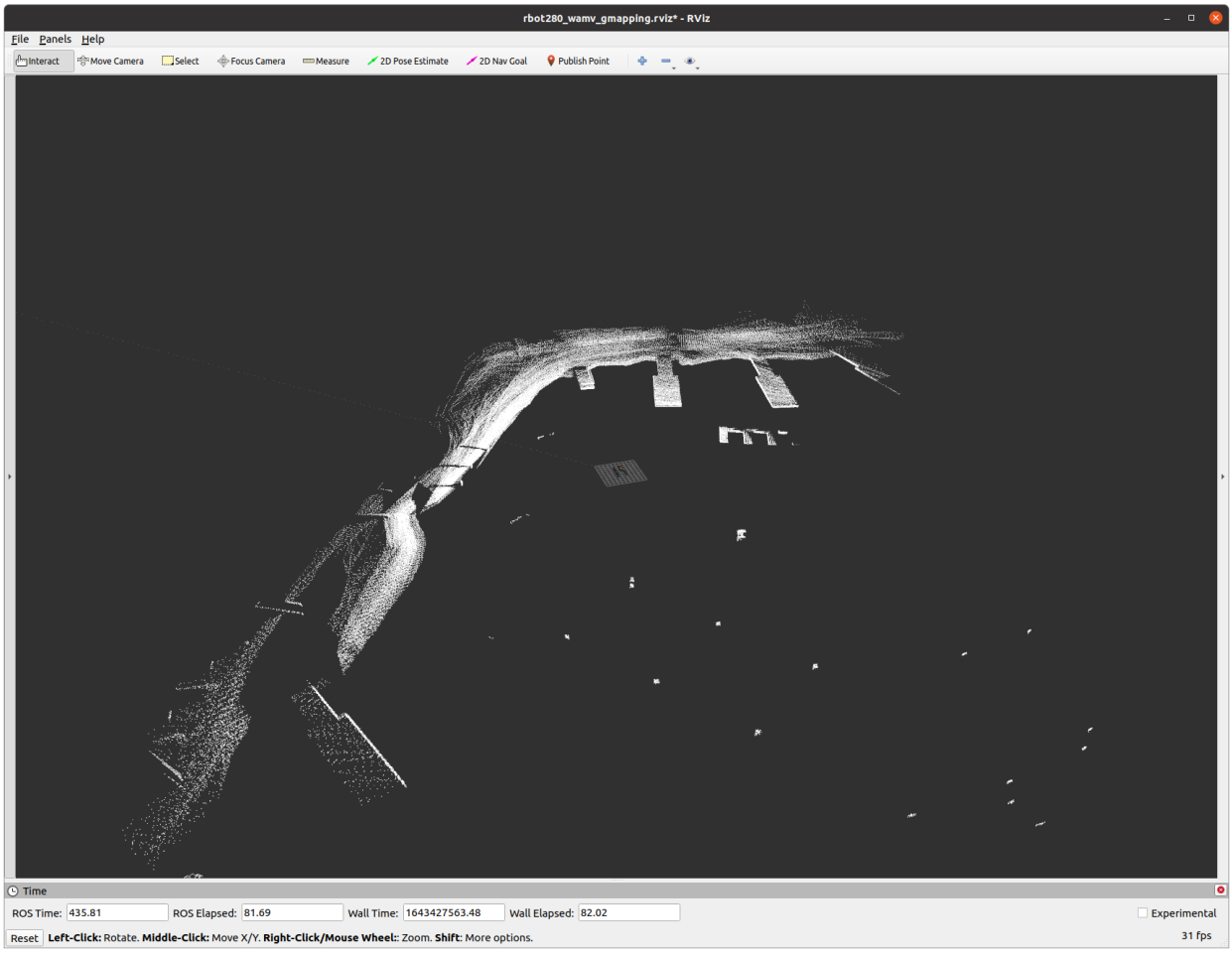
Updated map after parameter tuning and fixing some transform issues that were occurring due to how I was launching the nodes

The updated map is a little more clear than the first few that I made. The outline of the shoreline is a little more clear and the three docking bays in the top middle of the image are pretty clearly defined.

## Issues during Module 2 process

- Gmapping:
    - Setting up transforms
        - map->odom
        - odom->base_link
    - Laser needs to be at a Z = 1, aka horizontal to the robot
        - Laser was originally angled downward
    - Gmapping picks up a lot of the reflections from the laser from the pitch and roll caused by the waves. Also the map starts up in the wrong coordinate frame when

I use the previously saved map. This is an issue I still need to work through for the path planning module.

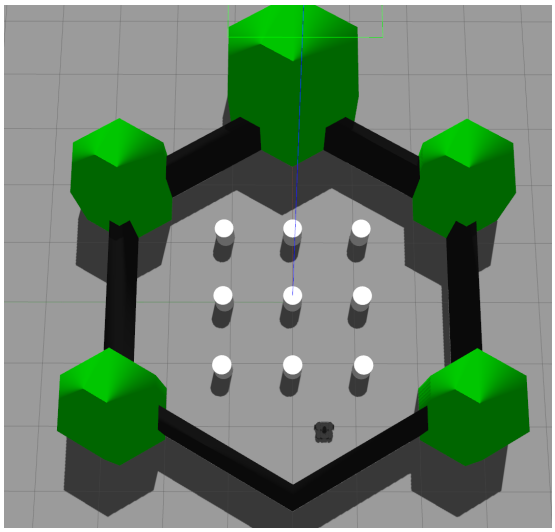
- 
    - Figure: Map output in RVIZ. Map is in the wrong coordinate frame. Also the occupancy grid is all white with no obstacles created.
- Checking that odometry to laser frame looks reasonable
    - Setting decay time on laserscan to 600 seconds and driving the robot around gives me a pretty nice, if not crude map (although looks better than anything I can reproduce with gmapping). I would take this as a sign that my odometry frame is mapped correctly.
- 

- Figure: Odometry to laserscan frame test. Looks good.

## Testing SLAM mapping with Turtlebot

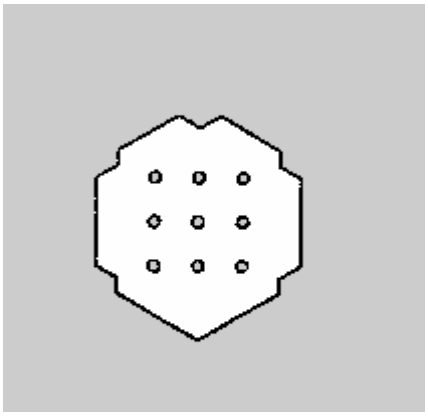
To try to figure out how to properly set up the parameters correctly I am going to test Gmapping and Hector slam ROS packages out in the simpler turtlebot indoor environment. This will give me a more intuitive understanding (I hope) to understand what is happening with my mapping tests with the surface vessel in a more dynamic environment. Unfortunately both the GMapping and Hector-SLAM ROS packages worked out of the box without needing to tune the parameters with these toy problems. So I did not gain much insight here.

- Install
    - turtlebot3
    - Turtlebot3-gazebo
- Launch Simulation World
    - $ export TURTLEBOT3_MODEL=waffle
    - $ roslaunch turtlebot3_gazebo turtlebot3_world.launch



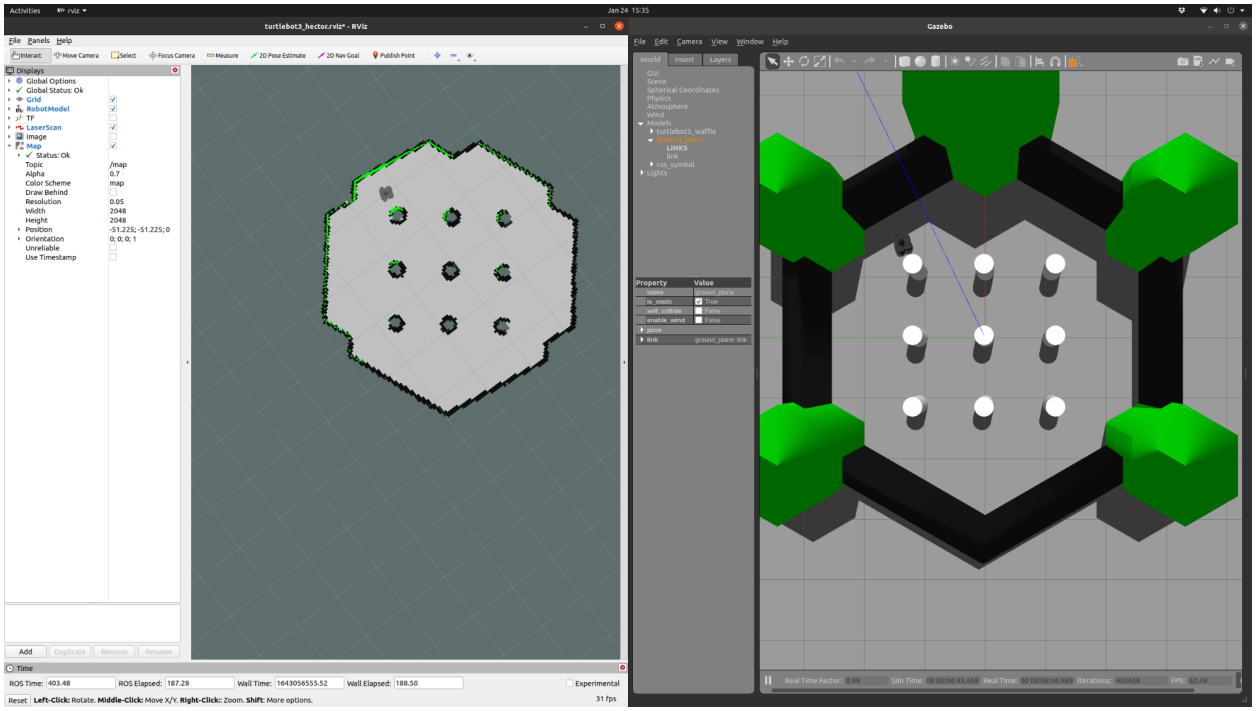
Sample turtlebot3_world in Gazebo

- Gmapping
    - $ export TURTLEBOT3_MODEL=waffle
    - $ roslaunch turtlebot3_gazebo turtlebot3_world.launch
    - $ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
    - $ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
    - Drive around to create a map
    - Save map: $ rosrun map_server map_saver -f gmapping



Turtlebot3_world map created using Gmapping

- Hector-SLAM
    - $ export TURTLEBOT3_MODEL=waffle
    - $ roslaunch turtlebot3_gazebo turtlebot3_world.launch
    - $ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=hector
    - $ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
    - Drive around to create a map
    - Save map: $ rosrun map_server map_saver -f hector



Turtlebot3_world map creation using Hector-Slam. Left: Rviz, Right: Gazebo

-

Video Links:

1. Gmapping video: https://www.dropbox.com/s/5y34ud1ewq92xl9/2022-01-29_gmappping_wamv_1.mp4?dl=0
    a. Note: The area that is mapped is very large compared to that of a room so the video is quite long. Feel free to forward through it.

# References (Module 2)


1. http://wiki.ros.org/gmapping
2. http://wiki.ros.org/pointcloud_to_laserscan
3. http://wiki.ros.org/navigation/Tutorials
4. http://docs.ros.org/en/noetic/api/robot_localization/html/state_estimation_nodes.html#ekf-localization-node
5. http://docs.ros.org/en/api/sensor_msgs/html/msg/LaserScan.html
6. http://wiki.ros.org/map_server

# Module 3: Object Detection

## Module Goals:

- Find obstacles using lidar
- Fine obstacles using camera
- Translate coordinates of objects into the Map frame of the ROS system
- Make video clip of work

## Module Notes:

### Record Bagfile

I am recording a small bag file of driving the robot up to the dock and into one of the docking bays, driving through the navigation buoys, and driving through the obstacle buoy field. This will give me lidar data and camera data to work on the object detection algorithms without having to run a full simulation every time I make a tweak in the algorithm.

#### Launch Order

- $ roslaunch rbot280 rbot280_bringup.launch
- $ roslaunch vrx_gazebo usv_keydrive.launch
- $ rosbag record -a

#### Bags Recorded

- 2022-02-09-21-55-05_driving-to-dock.bag
- 2022-02-09-21-58-15_dock.bag
- 2022-02-09-22-03-39_navigation-buoys.bag
- 2022-02-09-22-05-16_obstacle-buoys.bag

#### Testing with a bag file

$ roslaunch rbot280_detect wamv_detect_bag.launch

### Object Detection

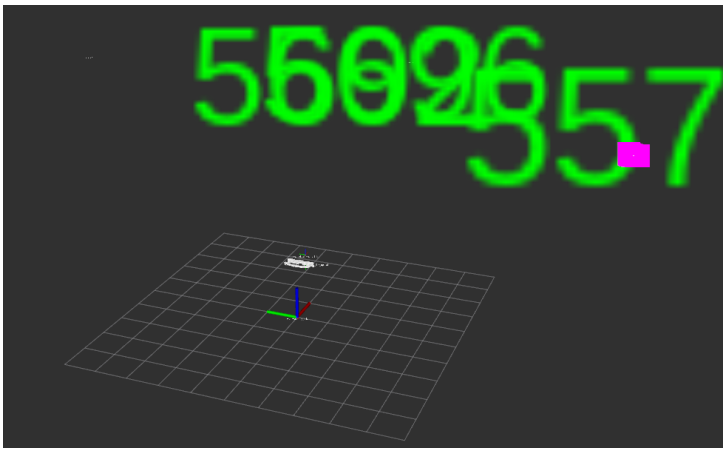
Creating a new package for object detection called rbot280_detect. This package will consist of PCL and CV detection algorithms to identify objects on the water. Additionally creating the package rbot_msgs to create custom message types for object detection and clustering.

## Object Detection: Lidar

The goal with the lidar output is to be able to determine where objects are located relative to the robot. This will be useful when using the robot in an autonomous mode for navigation to interesting points and to avoid objects as well. The Point Cloud Library (PCL) is an open sourced software project for 2D/3D image and pointcloud processing. I did not have to build PCL on my Linux system as it comes pre-installed in ROS, otherwise PCL is fairly simple to build from the command line and very painless. PCL is broken up into a series of modular libraries. Some of the important ones are filters, features, keypoints, registration, kdtree, octree, segmentation, sample_consensus, surface, recognition, io, and visualization.

Using the above PCL functions I was able to extract the clusters of objects that the laser detected on the water. I created a Cluster class to define clustered objects from the pointcloud and a few custom ROS msgs (CloudCluster) to package up some common parameters and pass the data around as a ROS topic.

I also created a filter for culling points that are not in a specified camera FOV. This was an attempt to match clusters from the Lidar to the camera, but have not been successful in that part of the sensor fusion process.



Clustered objects are shown in pink with an ID number associated with it

### Custom Messages:

#### Centroid.msg

std_msgs/Header header
geometry_msgs/Point[] points

#### CloudClusterArray.msg

std_msgs/Header header
CloudCluster[] clusters

CloudCluster.msg

std_msgs/Header header

uint32 id
uint32 last_seen  # sweep count since cluster last seen
string label

sensor_msgs/PointCloud2 cloud
geometry_msgs/PointStamped min_point
geometry_msgs/PointStamped max_point
geometry_msgs/PointStamped avg_point
geometry_msgs/PointStamped centroid_point
geometry_msgs/PolygonStamped convex_hull

pointcloud_cluster_detect package

```
### Subs
Input point cloud from lidar

* `pc_topic`
([sensor_msgs/PointCloud2](http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/PointCloud2.html))
Input point cloud from lidar

### Pubs

* `pc_topic/cluster_cloud`
([sensor_msgs/PointCloud2](http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/PointCloud2.html))
Point Cloud of clustered objects
* `pc_topic/cluster_centroids` (auvlab_msgs/Centroids)
Array of cluster centroids
* `pc_topic/detection/raw_cloud_clusters` (auvlab_msgs/CloudClusterArray)
Array of cloud clusters
* `pc_topic/chull_markers`
([visualization_msgs/MarkerArray](http://docs.ros.org/en/noetic/api/visualization_msgs/html/msg/MarkerArray.html))
Marker array for visualization of clusters in RVIZ
* `pc_topic/text_markers`
([visualization_msgs/MarkerArray](http://docs.ros.org/en/noetic/api/visualization_msgs/html/msg/MarkerArray.html))
Marker array for visualization of cluster IDs in RVIZ
```

## Object Detection: Camera

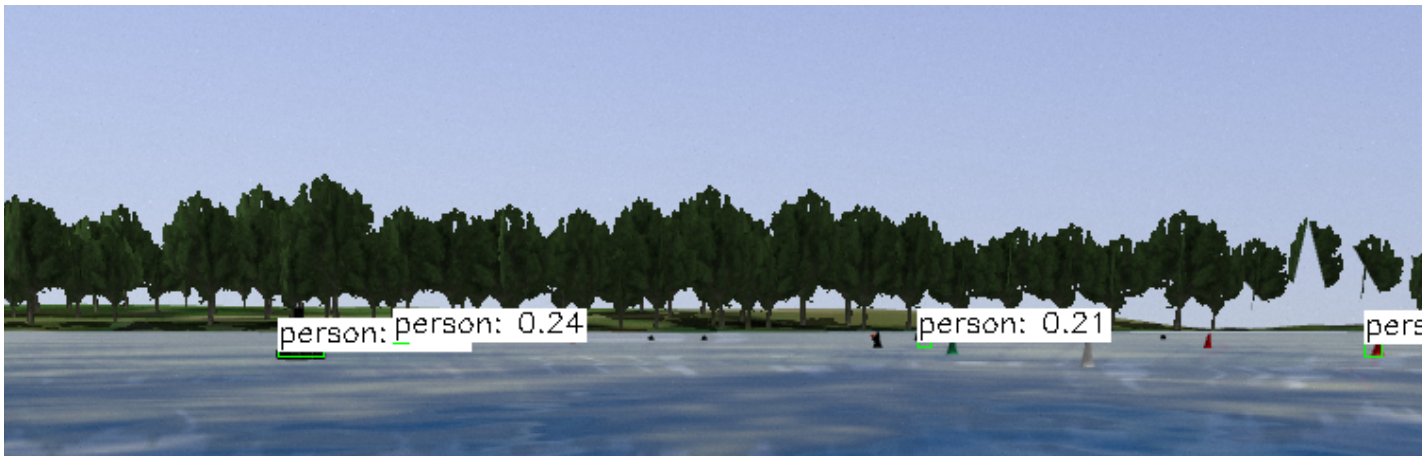
The goal with the camera is to identify the markers that indicate a docking bay and to report via topic what marker and color is assigned to each bay.

### YOLO: A Neural Network to classify objects and output a bounding box

Convolutional Neural Nets (CNNs) are specifically designed deep learning algorithms and models that expect as an input an image. The CNN essentially takes in an image and outputs an image of labeled objects. Of course this is an oversimplification but that is the end goal or result in most cases. Before CNNs computer vision was not very efficient for real time operations. To be honest, CNNs are not super efficient for real time operations without some serious computer and GPU power, but given the right components. CNNs perform better than anything in the computer vision world prior to them. Whole datasets of various image classes are trained with a CNN to create a model that can detect hundreds of objects at a very high accuracy.

I created a ROS node based in OpenCV to use a YOLO [1] machine learning model to identify objects on the water and assign a bounding box around them. YOLO is a CNN that is trained on the COCO dataset which is a large scale dataset that has 200k plus images that are labeled as many categories. Not going to take into consideration classification labels at the moment. All I am interested in is if there is a detection of something that is not water at the moment.

Using YoloV3, I attempted to use YoloV3-tiny for better performance on the CPU but it doesn't work well with the scene due to the robot's pontoon being in the view of the camera.
Outputs the detections of the buoys as a person if the buoy is elongated or a sport ball if the buoy is round.




Output image of the YOLO model through the center camera stream

-

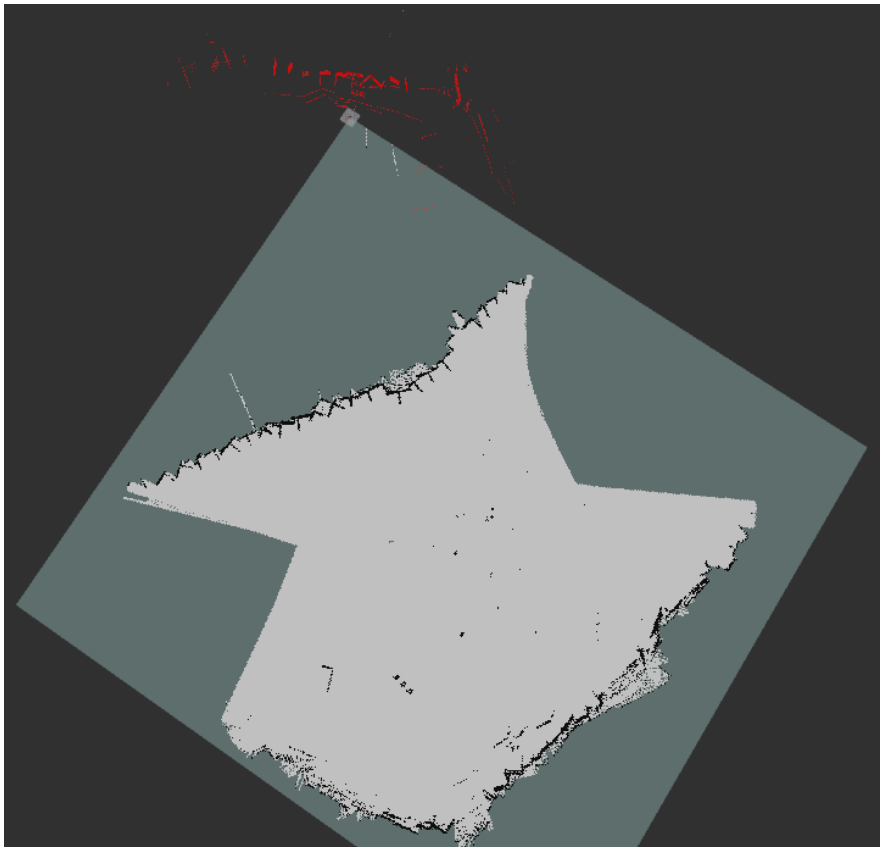
### Testing Detection on the Gazebo Simulation

- $ roslaunch rbot280 rbot280_bringup.launch
- $ roslaunch rbot280_detect wamv_detect.launch

## Converting objects (obstacles) to map frame coordinates

### Issues: Running tally of issues that I am running into

1. Map is not properly aligned with the robot in the world. Map origin is 0,0,0 and this is not correct as the map is



Origin of the map is 0,0,0 so when the map server starts up the robot appears to be at the left hand corner of the map. In the upper section of the robot you can see the red laser that should be aligned with the shore line of the map and it is not.

2. Localization package is not set up correctly
    a. Need to properly set up this package so that there is a proper transform from the map to odom to robot
3. Not able to use CUDA with the ROS OpenCV version which in turn doesn't allow for using a GPU for object detection with the camera. I am able to use a GPU outside of ROS so there must be a work around that I am not aware of at the moment.

# Module 3 Summary

## Robot to World Coordinates: Map Frame -> Odom Frame -> Robot Frame

As shown above I was having trouble aligning frames from the robot to the odometry frame to the world frame. I figured out that I was not properly setting the static transform [2] from the map frame to the odometry frame. I also figured out that I needed to use the robot_localization package [3] to fuse the IMU and GPS sensors via an Extended Kalman Filter (EKF) to produce a robot odometry and thus produce the odometry frame of the system. I ended up remapping the area of interest for my robot with these new settings. Now when I launch the robot with robot_localization the new map enables all of the frames (map->odom->robot) to align. Now when I report obstacles from the laser or camera, which is in the robot frame, the transform between the robot and the odom frame or map frame is available for

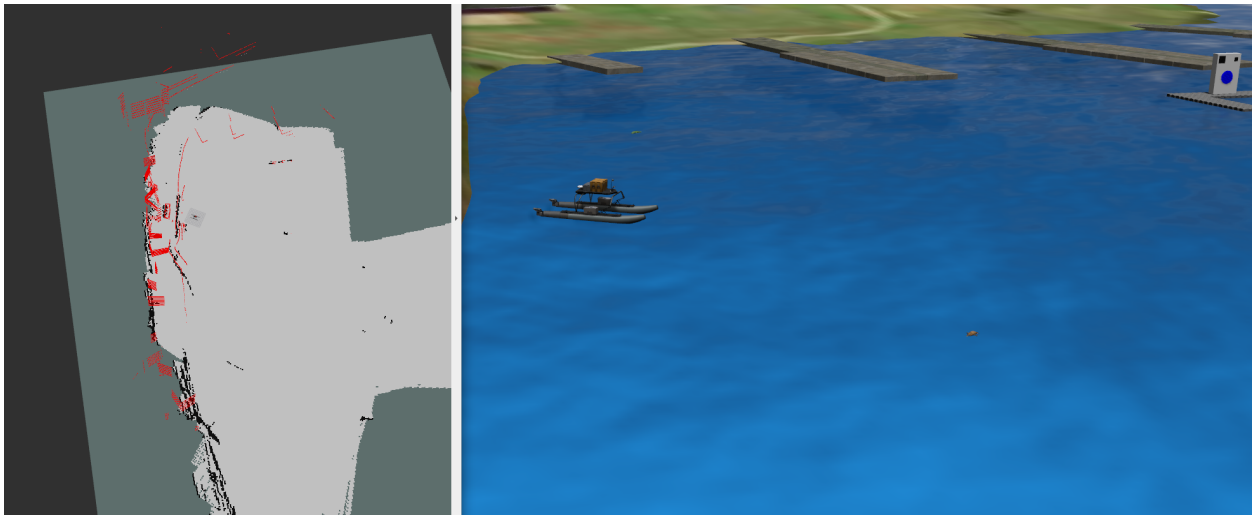
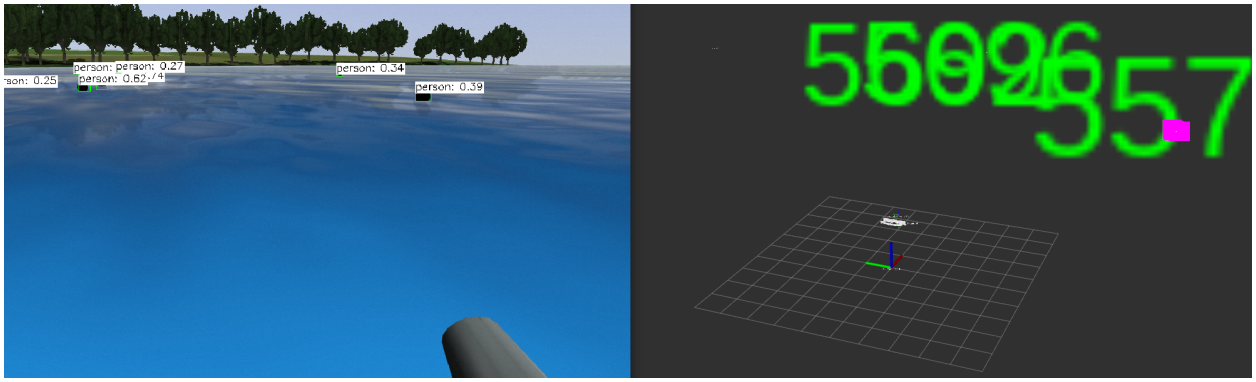


Image shows the robot laser shoreline output in red aligned with the map image shoreline

## Detecting Objects in the Pointcloud and Camera:

I was able to detect objects in the pointcloud and camera and report that back to the system in a coherent way that transforms the location of the objects to the world view of the ROS framework.




This image is a frame from the object detection process. Both the camera stream and the lidar stream are producing objects that are being detected by the robot.

## Future Work:

There are certainly more things that I want to do to make the obstacle detection of the robot more robust and have just run out of time. Below is a running list:

1. Retrain the YOLO ML model to identify objects that I am seeing on the water.
    a. Would need a larger set of images to train on
    b. Classes would include obstacle buoys, navigational buoys, and docks
2. Add a ROS msg that outputs the bounding box for each object detected
3. Figure out a work around to use DNNs with ROS and OpenCV
    a. Unfortunately OpenCV builds with ROS in such a way that you cannot change the parameters that OpenCV builds with. This means that OpenCV does not build with CUDA capabilities that would allow for use of GPUs to speed up object detections with the camera
4. Take a more traditional approach to the CV problem and use some of the techniques defined in OpenCV to figure out where objects are on the water.
5. Create a better way of associating clustered objects in PCL
    a. Current version is a simple euclidean distance and time last seen algorithm that assigns and matches IDs to clusters.
6. Fuse Laser obstacles with camera obstacles

## Video Links:

- https://www.dropbox.com/s/uvvil637i70nezq/ml_pcl.mp4?dl=0

# References (Module 3)

1. https://pjreddie.com/darknet/yolo/
2. http://wiki.ros.org/tf
3. http://docs.ros.org/en/noetic/api/robot_localization/html/index.html

# Module 4: Path Planning

## Module Goals:

- Plan path from point A to B
- Navigate robot to designated point
- Make video clip of work

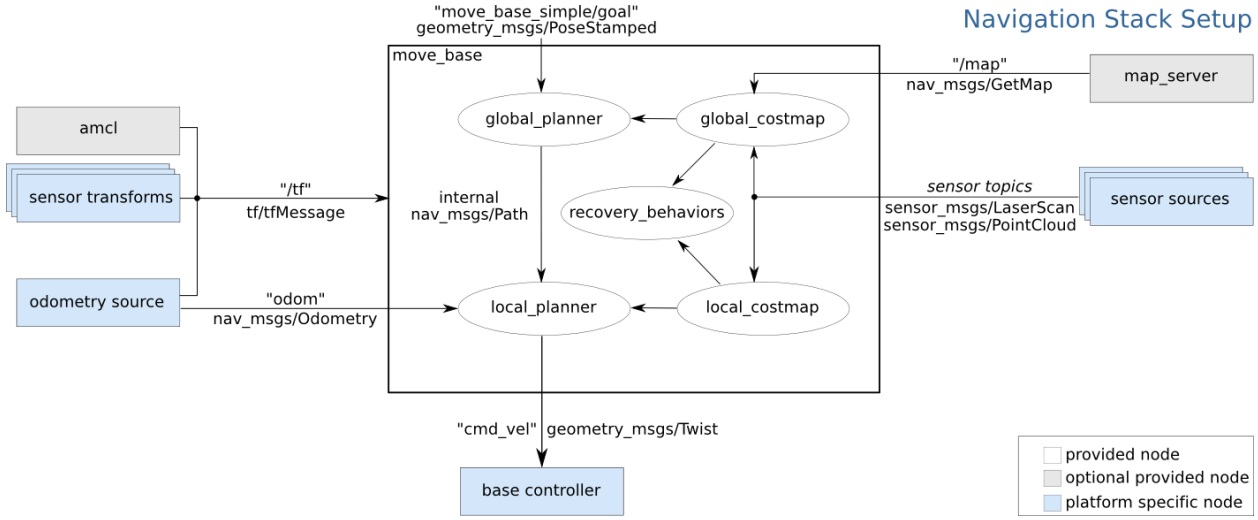
## Module Notes:

ROS Navigation stack. TEB local planner. http://wiki.ros.org/teb_local_planner
https://emanual.robotis.com/docs/en/platform/turtlebot3/nav_simulation/

## ROS Navigation Stack [4]

- The ROS Navigation Stack uses two costmaps to store information about obstacles in the world, a global costmap and a local cost map.
- The global costmap is used to generate long term plans over the entire environment
- The local costmap is used to generate the short term plans over the environment, i.e. to avoid obstacles
- Global Planning: Static paths from A to B
- Local Planning: Ability to get past unknown obstacles

## Move Base




Robot Setup with move_base

The move_base node provides a ROS interface for configuring, running, and interacting with the navigation stack on a robot. A high-level view of the move_base node and its interaction with other components is shown above. The blue vary based on the robot platform, the gray are optional but are provided for all systems, and the white nodes are required but also provided for all systems.

### move_base [5]

#### Subscribed Topics

- move_base_simple/goal (geometry_msgs/PoseStamped)
    - Provides a non-action interface to move_base for users that don't care about tracking the execution status of their goals.

#### Published Topics

- cmd_vel (geometry_msgs/Twist)
    - A stream of velocity commands meant for execution by a mobile base.

## Costmap Configuration

The navigation stack uses two costmaps to store information about obstacles in the world. One costmap is used for global planning, meaning creating long-term plans over the entire environment, and the other is used for local planning and obstacle avoidance. There are some configuration options that we'd like both costmaps to follow, and some that we'd like to set on each map individually. Therefore, there are three sections below for costmap configuration: common configuration options, global configuration options, and local configuration options.

### costmap_2d [1]

#### Subscribed Topics

- ~<name>/footprint (geometry_msgs/Polygon)
    - Specification for the footprint of the robot. This replaces the previous parameter specification of the footprint.

#### Published Topics

- ~<name>/costmap (nav_msgs/OccupancyGrid)
    - The values in the costmap
- ~<name>/costmap_updates (map_msgs/OccupancyGridUpdate)
    - The value of the updated area of the costmap
- ~<name>/voxel_grid (costmap_2d/VoxelGrid)
    - Optionally advertised when the underlying occupancy grid uses voxels and the user requests the voxel grid be published.

## Base Local Planner Configuration

The base_local_planner is responsible for computing velocity commands to send to the mobile base of the robot given a high-level plan. We'll need to set some configuration options based on the specs of our robot to get things up and running.

### base_local_planer [2]

#### Published Topics

- ~<name>/global_plan (nav_msgs/Path)
    - The portion of the global plan that the local planner is currently attempting to follow. Used primarily for visualization purposes.
- ~<name>/local_plan (nav_msgs/Path)
    - The local plan or trajectory that scored the highest on the last cycle. Used primarily for visualization purposes.
- ~<name>/cost_cloud (sensor_msgs/PointCloud2)
    - The cost grid used for planning. Used for visualization purposes. See the publish_cost_grid_pc parameter for enabling/disabling this visualization. New in navigation 1.4.0

#### Subscribed Topics

- odom (nav_msgs/Odometry)
    - Odometry information that gives the local planner the current speed of the robot. The velocity information in this message is assumed to be in the same coordinate frame as the robot_base_frame of the costmap contained within the

TrajectoryPlannerROS object. See the costmap_2d package for information about the robot_base_frame parameter.

## AMCL Configuration

### amcl [3]

#### Subscribed Topics

- scan (sensor_msgs/LaserScan)
  - Laser scans.
- tf (tf/tfMessage)
  - Transforms.
- initialpose (geometry_msgs/PoseWithCovarianceStamped)
  - Mean and covariance with which to (re-)initialize the particle filter.
- map (nav_msgs/OccupancyGrid)
  - When the use_map_topic parameter is set, AMCL subscribes to this topic to retrieve the map used for laser-based localization. New in navigation 1.4.2.

#### Published Topics

- amcl_pose (geometry_msgs/PoseWithCovarianceStamped)
  - Robot's estimated pose in the map, with covariance.
- particlecloud (geometry_msgs/PoseArray)
  - The set of pose estimates being maintained by the filter.
- tf (tf/tfMessage)
  - Publishes the transform from odom (which can be remapped via the ~odom_frame_id parameter) to map.

## Robot Setup for ROS Navigation Stack [6]

1. Transform Configuration:
   The transforms were previously set up as part of earlier modules.
2. Sensor Information:
   The navigation stack uses information from sensors to avoid obstacles in the world; it assumes that these sensors are publishing either sensor_msgs/LaserScan or sensor_msgs/PointCloud messages over ROS. This information is already set and the laser sensor is being published on the appropriate message type.
3. Odometry Information:
   The navigation stack requires that odometry information be published using tf and the nav_msgs/Odometry message. I set the odometry information over the previous modules.
4. Base Controller

The navigation stack assumes that it can send velocity commands using a geometry_msgs/Twist message assumed to be in the base coordinate frame of the robot on the "cmd_vel" topic. This means there must be a node subscribing to the "cmd_vel" topic that is capable of taking (vx, vy, vtheta) <==> (cmd_vel.linear.x, cmd_vel.linear.y, cmd_vel.angular.z) velocities and converting them into motor commands to send to a mobile base.

5. Mapping
   The navigation stack does not require a map to operate. I have set the map up in previous modules and will be using one for our purposes.

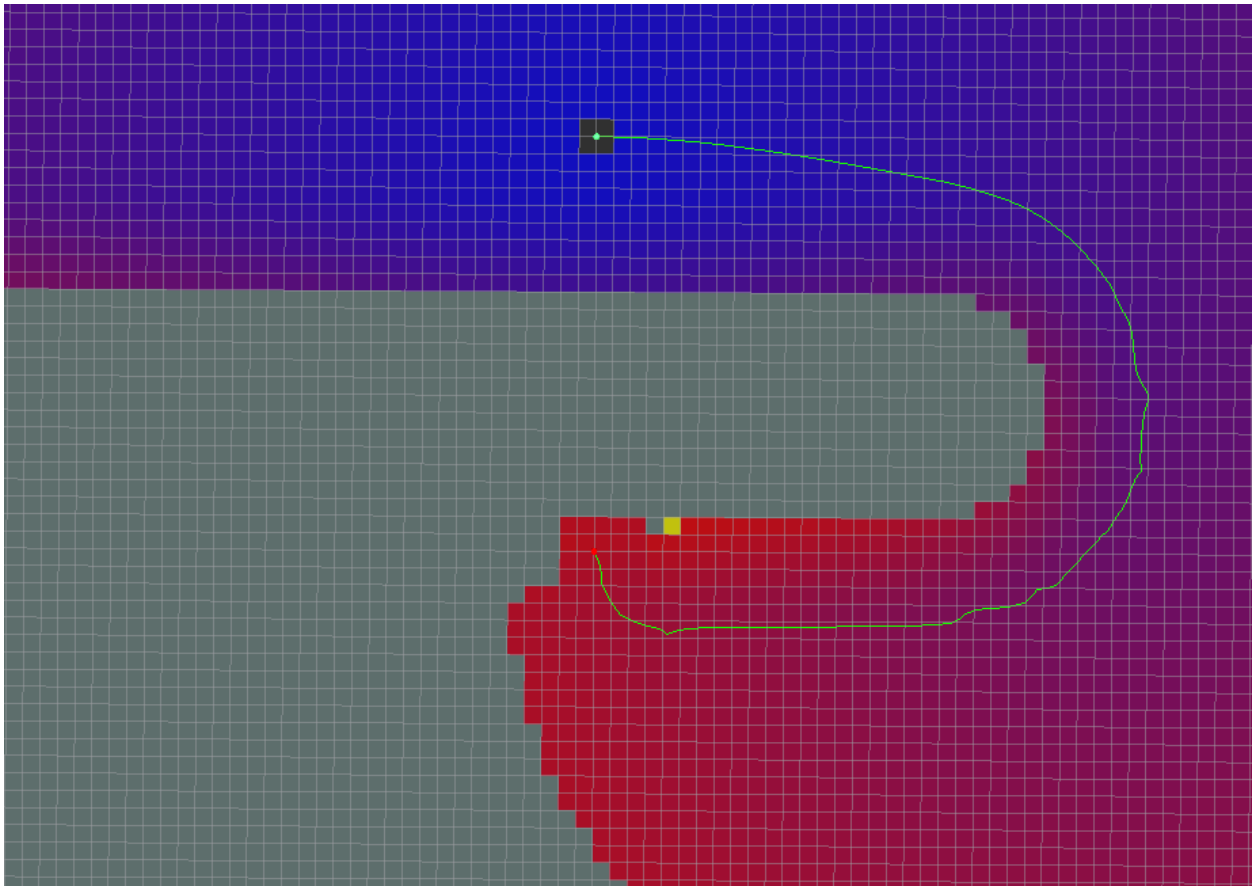
# Path Planning

## ROS base_local_planner

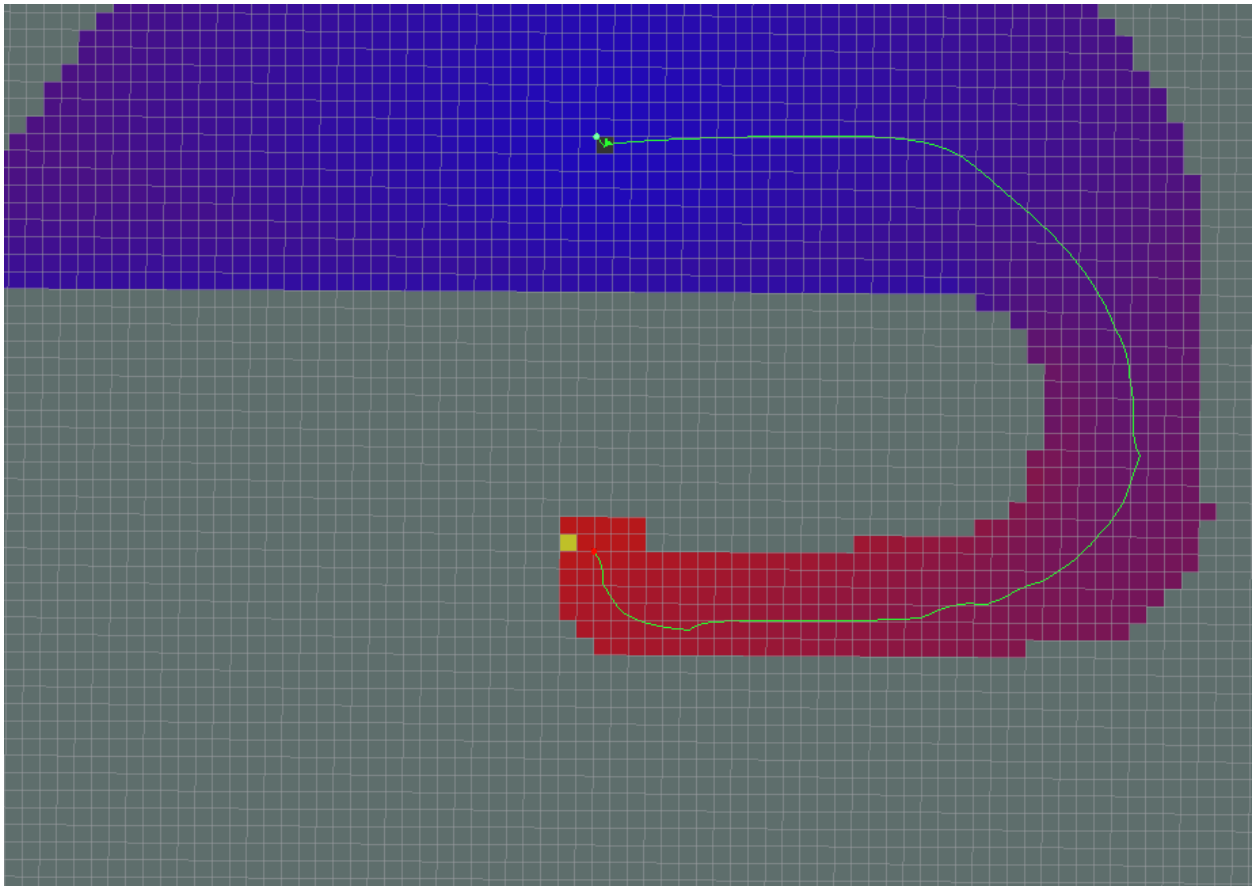
The base_local_planner package provides a controller that drives a mobile base in the plane. This controller serves to connect the path planner to the robot. Using a map, the planner creates a kinematic trajectory for the robot to get from a start to a goal location. Along the way, the planner creates, at least locally around the robot, a value function, represented as a grid map. This value function encodes the costs of traversing through the grid cells. The controller's job is to use this value function to determine dx,dy,dtheta velocities to send to the robot.

## ROS global_planner

This package provides an implementation of a fast, interpolated global planner for navigation. I ended up using the standard behavior for this planner as I saw a much slower planning rate for the A* algorithm.



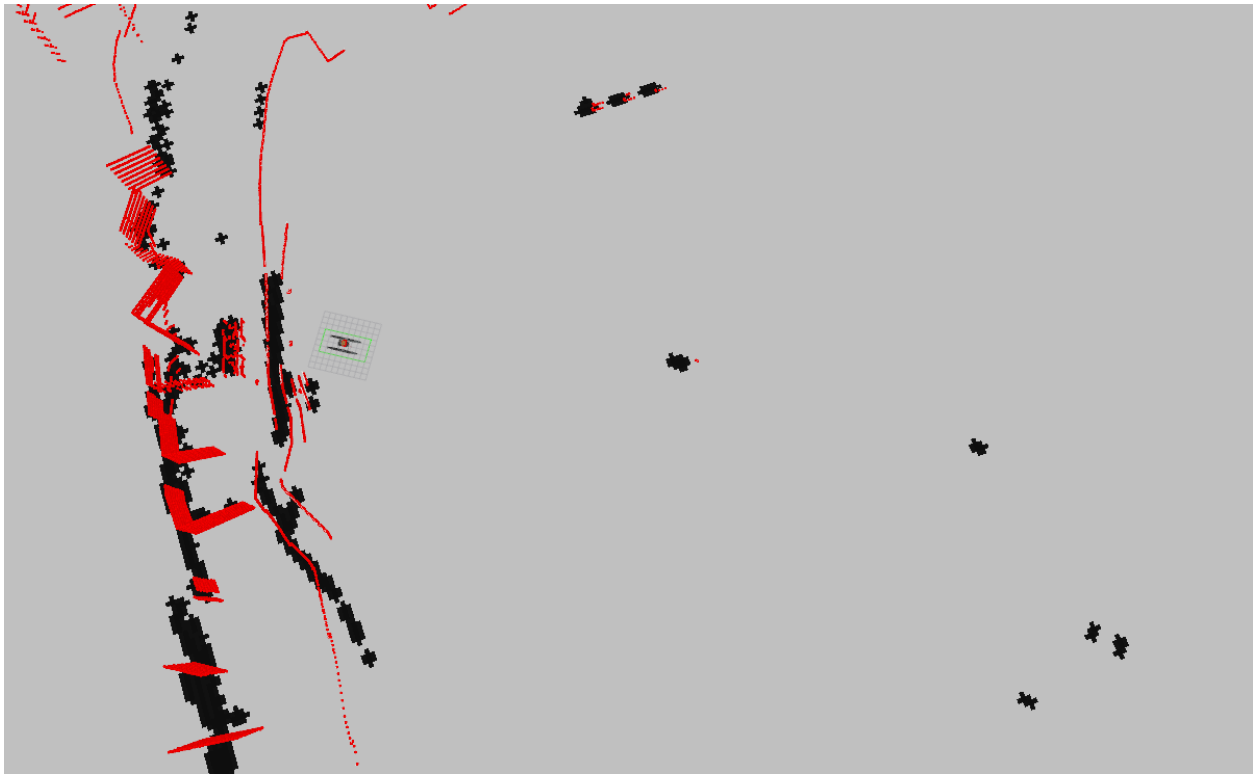
Standard Behavior



A* Path

## Path Planner in Action

With the ROS navigation package running I am able to produce a 2D global cost map and local cost map as well as a path to a goal point that I select using the 2D Nav Goal button in RVIZ.



Global Cost Map: Gray = Free Space, Black = Obstacle

Local Cost Map: Gray = Free Space, Black = Obstacle

Path To Goal in Teal

```
mikedef-home:~$ rostopic echo -n 1 /move_base_node/GlobalPlanner/plan
header:
  seq: 57
  stamp:
    secs: 1146
    nsecs: 673000000
  frame_id: "wamv/odom"
poses:
  -
    header:
      seq: 0
      stamp:
        secs: 1146
        nsecs: 673000000
      frame_id: "wamv/odom"
    pose:
      position:
        x: 11.180980682373047
        y: 16.182758331298828
        z: 0.0
      orientation:
        x: 0.0
        y: 0.0
        z: 0.8225818154657698
        w: 0.5686467768879361
  -
    header:
      seq: 0
      stamp:
        secs: 1146
        nsecs: 673000000
      frame_id: "wamv/odom"
    pose:
      position:
        x: 10.997119903564453
        y: 16.669635772705078
        z: 0.0
      orientation:
        x: 0.0
        y: 0.0
        z: 0.7964279272129011
        w: 0.6047334592656188
```

Path plan topic output

## Launch Steps:

- Rbot280_bringup

    - Launches the Gazebo simulated world and robot, rviz for sensor visualization, robot localization (sensor fusion for odometry), map server, transform between map and odom frame, pointcloud_to_laserscan
    - $ roslaunch rbot280 rbot280_bringup.launch
- Wamv_move_base
    - Launches the ROS Navigation Stack
    - $ roslaunch rbot280 wamv_move_base.launch

### Issues:

Really the main issue I ran into in the end is that the control of the robot needs to be fine tuned. I will work on that issue for the last two weeks as I think control to the goal point is a pretty important task. The other issue is that I could not really get the non-standard local path planners to work properly. The DWA approach worked with limited velocity and the TEB local planner seems to not be active at all. I will look into these a bit more as well as I think having an option for a path planner is important.

## Video Links:

- https://www.dropbox.com/s/jafw09n4igtyhhy/path_plan.mp4?dl=0

# References (Module 4)


1. https://wiki.ros.org/costmap_2d
2. https://wiki.ros.org/base_local_planner
3. https://wiki.ros.org/amcl
4. https://wiki.ros.org/navigation/Tutorials/RobotSetup
5. http://wiki.ros.org/move_base
6. http://wiki.ros.org/navigation/Tutorials/RobotSetup
7. http://docs.ros.org/en/api/geographic_msgs/html/msg/GeoPath.html
8.

# Bonus Module: Robot Control

One issue that I have run into is that the velocity topic (cmd_vel) output from move_base that is being published to move the robot to the goal location is not mapped correctly for my differential drive robot.

The control algorithm's job is to output control signals (velocity to thrusters) that reduce the error between the actual state and the desired state. For the robot in the marine environment the desired state will be the desired heading and desired speed.

## Control: Programmatic Operation of Robot Thrusters

The topic /cmd_vel will be published to drive the robot towards the goal point selected. This topic is a ROS geometry_msgs/Twist msg. Each thruster is controlled on its own separate topic /wamv/thrusters/left_thrust_cmd/ and /wamv/thrusters/right_thrust_cmd. The issue is how to correctly map to these topics such that the robot is controlled in a smooth way.

# PID Control Node (wamv_simple_contorller.py) :

## PID Background:

### Proportional Control

Goal is to steer the ship towards a desired goal heading. If the heading is slightly left, we want to correct slightly right. If the heading is significantly left or the desired heading then we want to correct the robot significantly to the left. The correction magnitude is directly proportional to the difference between the desired heading and the current heading. The further the robot is off, the harder the robot corrects.

$$Control\ Output\ =\ K_P * (Desired\ -\ Actual)$$

- Control Output = What we want to control. For instance heading or velocity
- $K_P$ = Gain term, a non-negative constant that determines how large the control signal will be for a particular error value.
- Desired = Set Point. Heading or velocity we want to reach
- Actual = Process Variable. Current measured heading or velocity
- Error = Desired - Actual

The Proportional term is all about adjusting the output based on the present error. It will consider the position of the actual versus the desired.

### Proportional Control Algorithm

```
While True:
      error = desired - actual
      Control output = Kp*error
```

### Derivative Control

The derivative term is used to slow down the correction as the set point is approached, i.e. avoid overshoot.

$$Control\ Output\ =\ K_p * Error\ + K_d * Rate\ of\ Change\ of\ the\ Error\ with\ Respect\ to\ Time$$

- $K_d$ = Derivative gain
- Error = $error_{t-1}$ - error / time step

The Derivative term is all about tracking the error over time and will help the controller look forward in time. If the error is decreasing too quickly with respect to time, this term will reduce the control output. This will help take into account the speed at which we get to the goal location. It will help dampen the oscillations.

### Derivative Control Algorithm

```
error_prev = desired - actual
While True:
      error = desired - actual
      d = (error_prev - error) / time_step
      Control output = Kp*error Kd*d
      error_prev = error
```

## Integral Control

The integral term looks for the residual error that is generated even after the proportional control is applied. The integral term seeks to get rid of the residual error by incorporating the historical cumulative value of the error. As the error decreases, the integral term will decrease.

$$Control\ Output\ =\ K_p\ *\ Error\ +\ K_i\ *\ Sum\ of\ Errors\ Over\ Time\ +$$
$$K_d\ *\ Rate\ of\ Change\ of\ the\ Error\ with\ Respect\ to\ Time$$

- $K_i$ = Integral constant

The integral term is all about adjusting the output based on the past error. This term stores up the error that the system sees over time. This term helps if the robot can't quite make it to the goal location.

### Integral Control Algorithm

```
error_prev = desired - actual
error_sum = 0
While True:
      error = desired - actual
      error_sum = error_sum + error * time_step
      d = (error_prev - error) / time_step
      Control output = Kp*error + Ki*error_sum + Kd*d
      error_prev = error
```

PID is a control method that is made up of three terms:

- Proportional gain that scales the control output based on the difference between the actual state of the system and the desired state of the system (i.e. the error).
- Integral gain that scales the control output based on the accumulated error.

- Derivative gain that scales the control output based on the rate of change of the error (i.e. how fast the error is changing).

### Controller Node

Goal is to create a linear (x) and an angular (z) PID controller loop within the node, one controller to output the forward velocity that keeps the robot moving forward until it reaches the goal, and one controller to output the rotational velocity that turns the robot toward the goal. The node will subscribe to the goal waypoint that is published by the path topic, extract the xy waypoint position, calculate the angle error and distance error, then use the PID loop to publish to the left and right motor topics that will be used to drive the robot towards the goal. A maximum speed and maximum steering angle is passed in through a parameter server.

Setting up a Dynamic reconfig file so I can adjust PID parameters on the fly while the mission is running. http://wiki.ros.org/dynamic_reconfigure/Tutorials/HowToWriteYourFirstCfgFile

## Launch Steps:

- Rbot280_bringup
    - Launches the Gazebo simulated world and robot, rviz for sensor visualization, robot localization (sensor fusion for odometry), map server, transform between map and odom frame, pointcloud_to_laserscan
    - $ roslaunch rbot280 rbot280_bringup.launch
- Wamv_move_base
    - Launches the ROS Navigation Stack
    - $ roslaunch rbot280 wamv_move_base.launch
- Wamv_control
    - Launches controller to follow goal points from path planner
    - $ roslaunch rbot280 wamv_conrol.launch simple_control:=true

## General Notes:

- Right now I have a very basic controller working. Going to stop here and focus on module 5 and sensor fusion work.
- I found the controller to be very responsive when tuning for either the angular position or the speed. When the controllers are combined into a single control loop, I have found the robot to be less than ideally responsive.
- As I work through module 5 I will circle back on better tuning of the controller.

# Module 5: Detect and Report

## Module Goals:

- Find and identify objects and coordinates in odom frame
- Plan path and navigate to an object
- Navigate to home location after reporting object
- Make video clip of work

## Module Notes:

### 1. OpenCV Machine Learning GPU workaround

Back in Module 3 I wrote a ROS node based on OpenCV's machine learning functions. These functions allow you to use any major machine learning framework and its associated model to perform machine learning tasks on an image. The problem with this node is that ROS builds with a specific version of OpenCV that does not yet have the specific functions to use the ML models with a GPU enabled. So for a work-around I wrote a ROS wrapper node (Darknet_node2.py) to use the Darknet.py program created by Alexey Bochkovskiy [1]. The node is called darknet_node2.py and now allows for the use of a GPU to speed up detections. It takes in any number of subscriptions for input video streams, in my case I am concentrating on the single center camera onboard the robot. The images are processed using OpenCV to resize the images at the size the YOLO model expects them. The images are then run through the selected YOLO based model and detections are returned based on the image. The detections are then scaled to be superimposed onto the original image as bounding boxes and detection labels.

I also had to throttle images coming into the detector as the raw camera images were coming in at 30 Hz and were creating a lag on the detected images. I used the ROS package 'topic_tools' to create a topic throttler [2]. While the model runs very smoothly with the new darknet_node it is limited to the classifications provided by the COCO dataset. While this is fine for now, it is not optimized for "on the water" autonomous vehicle driving. This is shown to be evident by the predictions coming up as sheep, sports ball, person, cow, and sometimes a boat (see figure below). If I had the time I would train a model to detect and classify items on the water such as mooring balls, navigational buoys, ships, sailboats, docks, ect. To prove that retaining is possible I will retrain a model that just labels all things on the water as "object".

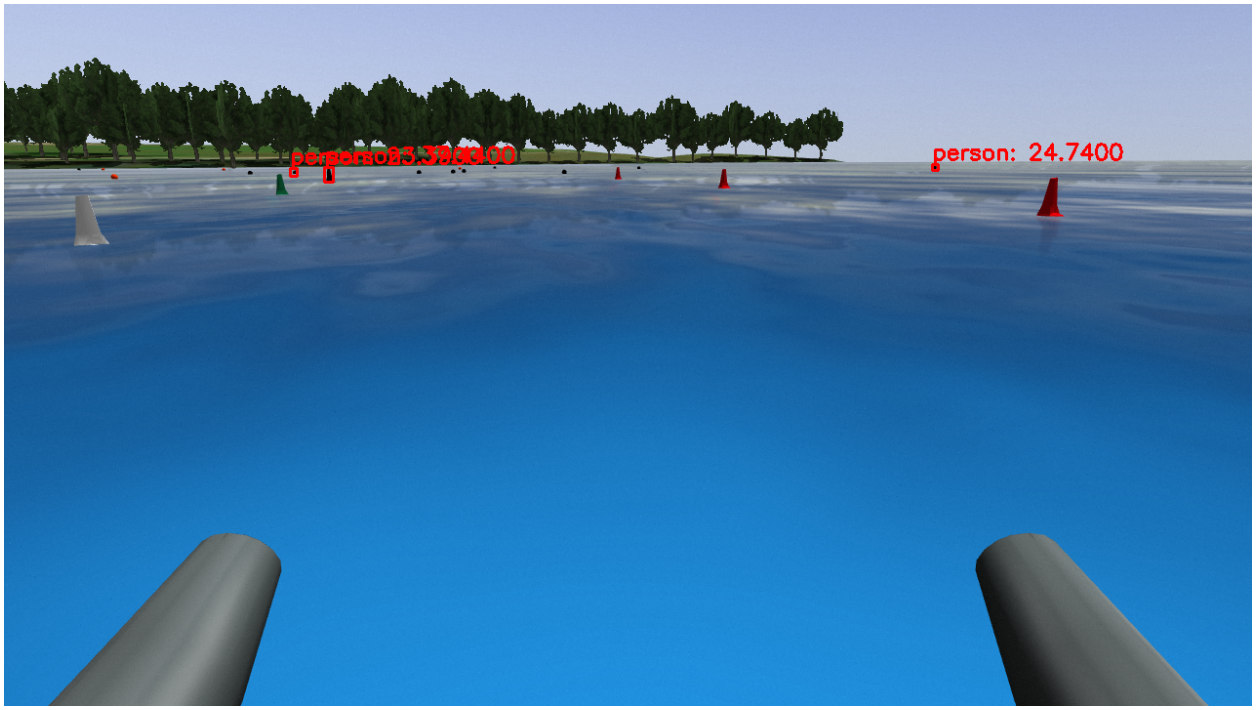



Figure: Camera YOLO ML model output when trained on the COCO dataset. You can see mislabeled and unlabeled buoys in the image.

### Launch Steps for image object detection:

- Rbot280_bringup
  - Launches the Gazebo simulated world and robot, rviz for sensor visualization, robot localization (sensor fusion for odometry), map server, transform between map and odom frame, pointcloud_to_laserscan
  - $ roslaunch rbot280 rbot280_bringup.launch
- Wamv_detect
  - Launches the refactored YOLO node that will use OpenCV natively on Linux rather than through the installed version in ROS
  - $ roslaunch rbot280_detect wamv_detect_test.launch

## 2. Marine Based YOLO Model

As mentioned in the previous section, the standard YOLOv4 model that is trained on the COCO image dataset is not adapted to the marine environment. I wanted to retrain the model instead of classifying objects in relation to the COCO classes for it to classify based on marine classes. Unfortunately after reading through Darknet README to gauge what is needed to accomplish this task I quickly realized this would be a tall order to customize based on the marine environment. Also, while the Gazebo world is marine based it is not marine in the sense of a local harbor where you see various shipping traffic. The world is a competition world used by RobotX and only has buoys and docks on the water. I decided to retrain the model by just

labeling everything as an ‘object’ class. This ended up being a very doable task. Below are a few images from the output of the retrained YOLOv4 model.

I took the following steps to retrain the YOLOv4 model:

- Gather images for training
    - Gathered a set of <200 images of buoys, docks, and other items on the water
- Label images using “LabelImg” python program [5]
    - pip3 install labelImg
    - Labeled each imaged with a bounding box over each item on the water’s surface under the classification label of “object”
- Retrain YOLOv4 with new images
    - Followed these steps from this blog [4] to retrain yolov4 using Goggle Colab. Had to make a few modifications to load my labeled images properly. Let the model train for 4+ hours
        - Saved weights and cfg file for use in custom ROS node (more on this node below)
        - Below are some test images. See Video for realtime output

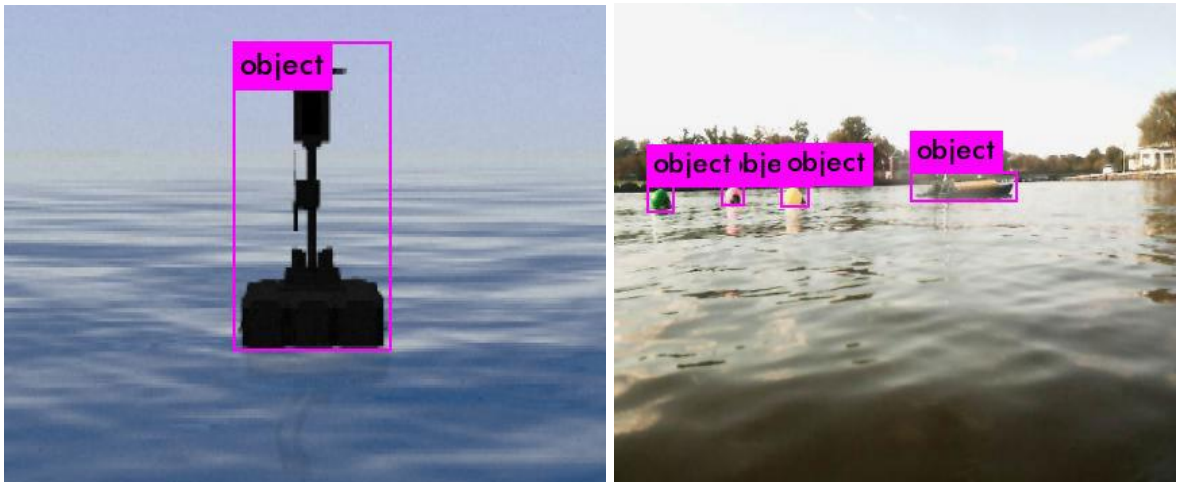



Figure: Output of retrained YOLOv4 ML model

# 3. Sensor Fusion

## Pointcloud filter by camera view node:

To simplify the amount of 3D lidar points that need to be transformed to 2D and checked to see if they overlay into the camera view I first created a node to filter out unnecessary lidar points for the overlay operation. This node is called pointcloud_camera_filter and is in my rbot280_detect ros package. The basics of this node are similar to the one I wrote to cluster points returned by the lidar into objects. The added feature is the filterCameraView() function that takes in the camera’s field of view (FOV) and the camera’s theta. Theta is a parameter used to define the camera’s horizontal static angle relative to the front of the robot. In the case of the images below

I am using the center camera so theta is set to 0 and the camera has a FOV of 80° so that parameter is set to 80. The function then filters out any points that don't fall within the camera's theta based on the bearing to that point from the front of the robot.

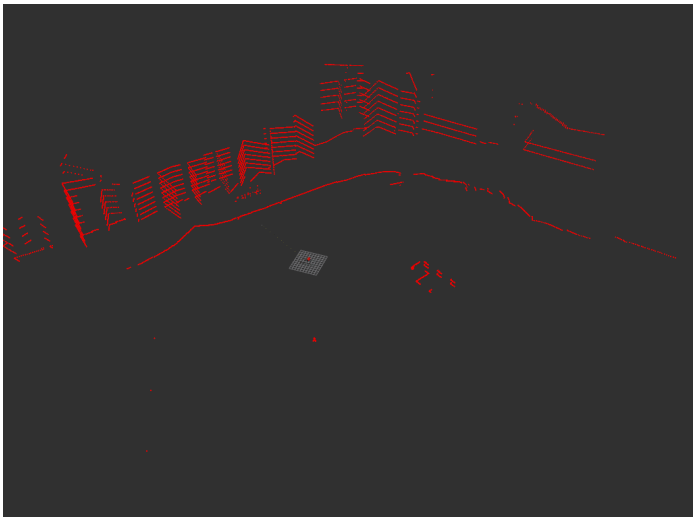


Figure: Raw pointcloud return from lidar

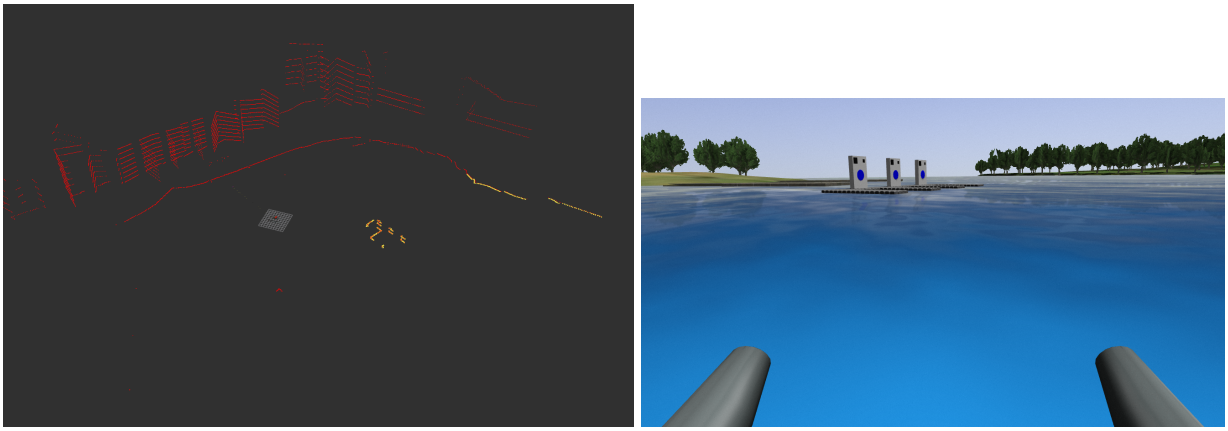


Figure: **Left:** Filtered pointcloud for the camera view displayed in yellow. Raw pointcloud returns displayed in red. **Right:** Image view of center camera that corresponds to the filtered pointcloud in yellow.

## Pointcloud Image Overlay Node:

Created ROS node that projects pointcloud into image frame. Node name is pointcloud_image_overlay in the rbot280_detect package. The node subscribes to both a video stream topic and a pointcloud topic. I implemented a message filter so that the image and the points used are approximately at the same timestamp. The node grabs information like camera intrinsics and the transform between the lidar and the camera as the node initializes. The transform consists of a translation matrix and a rotation matrix between the two sensors. The transforms are used to convert 3D points into 2D points and can be used in the opposite

direction if needed. As the two messages from each sensor come in the transform is performed and the lidar points are superimposed onto the camera image. In the images below I first filtered the pointcloud based on the horizontal FOV of the center camera, then projected the points into the 2D image space and overlaid the points into the image as shown below.

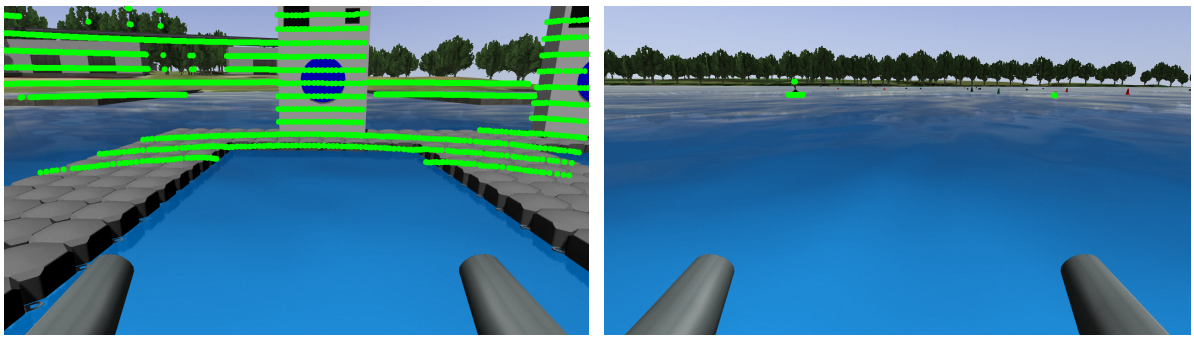


Figure: Projected point cloud returns from the camera POV.

Next I modified the pipeline that adds the ability to project points from the Clustered Lidar into the image frame that is overlaid into the returns from the machine learning model to fused the two outputs together as shown below. You can see in the image the return from the clustered pointcloud overlaid into the bounding box of the closest object detected. The lidar only reaches about 100m out and the other objects are currently out of range for clustered returns.

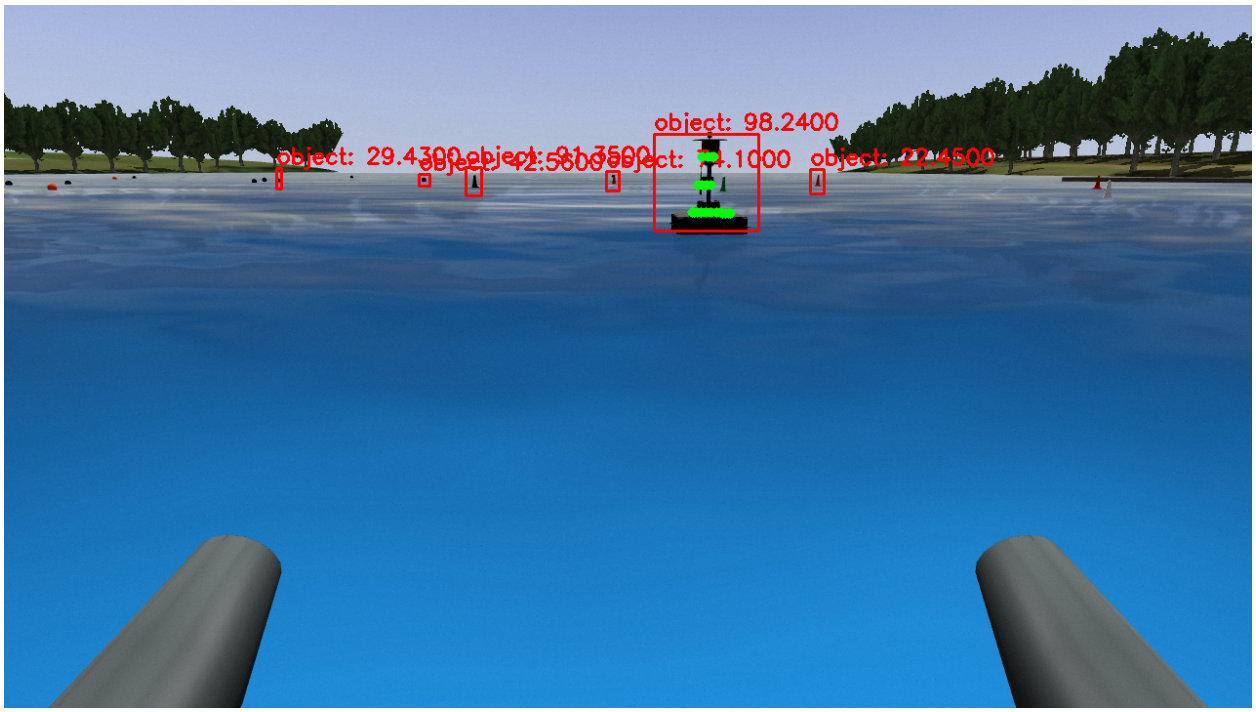



Figure: Pointcloud returns projected into image in green. Objects detected in red bounding boxes.

## 4. Motion Planning

Putting the capstone all together. A typical autonomous robot requires the robot to plan, sense, and act. For the robot I chose, a marine robot based on the WAM-V platform, I will have the robot drive to a buoy (route plan), sense the buoy with its sensor stack (lidar/camera), and identify the object (action). The lidar reports object position based on a reference odometry point set at the beginning of the mission. The camera identifies the object by using the previously defined machine learning model.

In the final video the robot starts at the defined 0,0 odometry point. It then drives towards the buoy, attempting to sense and classify the buoy as an object. Once satisfied the robot attempts to return to its starting point. The robot is following a predefined list of waypoints set in the odometry frame to accomplish the motion planning task.

### Launch Steps:

- Rbot280_bringup
    - Launches the Gazebo simulated world and robot, rviz for sensor visualization, robot localization (sensor fusion for odometry), map server, transform between map and odom frame, pointcloud_to_laserscan
        - $ roslaunch rbot280 rbot280_bringup.launch
- Wamv_detect
    - Launch the refactored YOLO node that will use OpenCV natively on Linux rather than through the installed version in ROS
        - $ roslaunch rbot280_detect wamv_detect.launch
    - Launch the pointcloud_camera_filter app that only returns pointcloud data within the camera's field of view
    - Launch the pointcloud_image_overlay app that transforms the pointcloud 3D points to 2D image points and overlays the points on a new image
- Wamv_control
    - Launches controller to follow goal points from path planner
        - $ roslaunch rbot280 wamv_conrol.launch simple_control:=true
- Wamv_move_base
    - Launches the ROS Navigation Stack
        - $ roslaunch rbot280 wamv_move_base.launch

# References (Module 5)


1. https://github.com/AlexeyAB/darknet
2. http://wiki.ros.org/topic_tools/throttle
3. https://medium.com/analytics-vidhya/train-a-custom-yolov4-object-detector-using-google-colab-61a659d4868
4. https://blog.roboflow.com/training-yolov4-on-a-custom-dataset/
5. https://github.com/tzutalin/labelImg#labelimg

6.

Github: https://github.com/mikedef/rbot280_vrx