Final Report

Autonomous Water Taxi

Team Members (Team B):
Tushar Chugh
Shiyu Dong
Bikramjot Hanzra
Tae-Hyung Kim
William Seto

Mentors:
John Dolan
Dimitrios Apostolopoulos

Sponsor:
Jeremy Searock

THE ROBOTICS INSTITUTE

May 6, 2016

Abstract

In this project, we designed the autonomous navigation system for a 27 ft boat. The aim of the project was to demonstrate the feasibility of a possible transportation method on the river using an autonomous driving boat, i.e., the water taxi.

The navigation system uses Radar to get the information of obstacles, shores and bridges. Since the radar is a commercial product, we used an open source library, OpenCPN, to interface with and obtain raw data from the radar. Since the raw data is quite noisy, we leveraged OpenCV functions and ROS packages to filter the radar data. To successfully navigate under the bridge, we segmented shores, bridges and obstacles using morphological operations. Then we overlaid the final obstacles on the occupancy grid map. To make the boat steer away from shores and obstacles, we inflated the cost near the shores and obstacles and sent the costmap to the path planner. The path planner takes user defined goals from the GUI, obstacle information from the perception subsystem, and then plans an optimal path using the Search-based Planning library (SBPL). The planner generates waypoints for the entire path and sends the waypoints to the low level controller to control the boat.

As our final result, we achieved an autonomous driving boat that can detect at least 85% of the obstacles on the river, traverse 2 miles in less than 15 minutes, and arrive within 15 meters of the destination.

THE ROBOTICS INSTITUTE

Contents

1 Project Description **1**

2 Use Case **2**

3 System-level Requirements **4**
3.1 Mandatory Functional Requirements . . . 4
3.2 Desirable functional requirements . . . 5
3.3 Mandatory non-functional requirements . . . 5
3.4 Desirable non-functional requirements . . . 6

4 Functional Architecture **7**
4.1 Functioning with perception and path planning . . . 7
4.2 Functioning with simulator and logged data . . . 8

5 System-level trade studies **9**
5.1 Sensor Trade Study . . . 9
5.2 Map APIs for the OCU . . . 9

6 Cyber-physical Architecture **10**
6.1 Data Acquisition . . . 10
6.2 Obstacle Detection . . . 11
6.3 Path Planning . . . 11
6.4 OCU . . . 11
6.5 Low-level Controller . . . 11

7 System Description and Evaluation **12**
7.1 Subsystem Depictions . . . 12
7.1.1 Perception . . . 12
7.1.2 Path Planning . . . 14
7.1.3 Simulation . . . 18
7.1.4 GUI and Low-level Controller . . . 21
7.2 Modeling, Analysis, and Testing . . . 22
7.2.1 Radar Analysis . . . 22
7.2.2 Path Planning . . . 23
7.2.3 Integration and Testing . . . 24
7.3 SVE Performance Evaluation . . . 24
7.4 Strong/Weak Points . . . 25
7.4.1 Strong Points . . . 25
7.4.2 Weak Points . . . 26

8 Project Management **27**
8.1 Schedule . . . 27

8.2 Budget 28
8.3 Risk Management 29

9 Conclusion **31**
9.1 Lessons learned 31
9.1.1 Technical lessons 31
9.1.2 Project management lessons 31
9.2 Future Work 31

References **33**

List of Figures

1.1 27 ft SeaHawk Aluminum Welded Boat 1

2.1 AutoPirates App 2
2.2 Operator Control Unit 3
2.3 The boat arrives at the destination 3

4.1 Functional Architecture 7
4.2 Functional architecture of logged data and simulator 8

6.1 Cyber-physical Architecture 10

7.1 Obstacle Detection 12
7.2 Radar Filtering 13
7.3 Bounding boxes around the obstacles 14
7.4 Occupancy Grid Map 14
7.5 GUI to add obstacles 15
7.6 Added cost in Occupancy Grid Map 15
7.7 Cost Maps for driving different directions 16
7.8 Following rules-of-the-road 16
7.9 Path on Small Size obstacles 17
7.10 Path on Medium Size obstacles 17
7.11 Path on Large Size obstacles 18
7.12 Path on Large Size obstacles 18
7.13 RViz Visualization 19
7.14 Static Obstacles 20
7.15 GUI 22
7.16 low-level Controller 22
7.17 Radar Control GUI through OpenCPN plugin 23
7.18 SVE Performance Evaluation 25

8.1 Risk Likelihood-Consequence Table 30

List of Tables

5.1 Sensor Trade Study 9
5.2 OCU Map API Trade Study 9

8.1 Schedule for Fall Semester 27
8.2 Schedule for Spring Semester 27
8.3 Hardware/Equipment/Budget provided by the sponsor 28
8.4 MRSD Project Budget 28
8.5 Risk Management 29

1. Project Description

Figure 1.1: 27 ft SeaHawk Aluminum Welded Boat

The aim of this project is to create a proof of concept that demonstrates a viable water taxi system for the City of Pittsburgh. Many of Pittsburgh's attractions are located along the three rivers and would be easily accessible by boat. Currently, there are only a few luxury cruise lines that are mainly used for entertainment and tourism, rather than as a serious form of transportation. Although the federal government has allocated money to the city for building docks to support transportation, neither the city nor any companies have found a good business model for such an operation. We believe developing an autonomous vessel would help bring the water taxi operation to fruition. A fleet of autonomous water taxis would reduce commute time for citizens, boost tourism for the city, and grow businesses located along the rivers.

To support this effort, the National Robotics Engineering Center, the applied research lab of the Robotics Institute, has recently acquired a 27 ft SeaHawk aluminum welded boat (Figure 1.1). The boat has been converted to a test bed for maritime autonomy research, with drive-by-wire controls and a defined interface. The boat is outfitted with autonomy sensors, a positioning system, and also has a cabin and heating/air-conditioning to facilitate year-round use for developers onboard. In this project, we will develop algorithms and tools to enable autonomous navigation of the boat to defined destinations along the river.

2. Use Case

James would like to spend a nice day with his family, beginning with some shopping at Walmart, followed by an exciting game at PNC Park, and finally a nice dinner at the Cheesecake Factory in Southside Works. Unfortunately, as these locations are located in completely opposite directions, he has been unable to make this perfect day materialize.

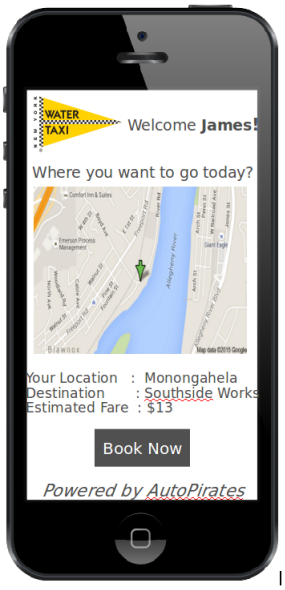

Figure 2.1: AutoPirates App

Luckily, the new Auto-Pirates service and smart phone app has just launched, which allows anyone access to their own private water taxi ride. After installing the app, James sees some predefined pickup points/destinations on the map. James heads over to the closest pickup location, the Monongahela Wharf, and proceeds to request a taxi on his app (Figure 2.1). His location is sent to a server, which manages all the boats in the fleet. The system dispatches the closest boat which is currently unused, or is carrying passengers to James' current location. As the boat is en route, James also receives a special code on his app, which will allow him exclusive access to the boat once it arrives.

Figure 2.2: Operator Control Unit

The boat arrives at the specified Auto-Pirates dock in the Wharf. After James gets on the boat, he proceeds to the Operator Control Unit, which is a touchscreen map similar to his smart phone app (Figure 2.2). James enters the code he received earlier and can now select a destination. He chooses Southside Works, and then the boat begins planning its trip. As the boat autonomously backs out of the dock, James can see a planned path on the map. As James enjoys the beautiful view along the river, he notices a large cargo ship approaching. He looks at the OCU and sees that the ship shows up as a blip on the map and the taxi has recalculated a new path that goes around the obstacle. He can also see buoys and bridges pop up as obstacles on the map.

Figure 2.3: The boat arrives at the destination

Eventually, the boat arrives at Southside Works and autonomously parks at its reserved dock, where the next set of passengers are waiting. (Figure 2.3)

3. System-level Requirements

3.1 Mandatory Functional Requirements

The system shall -

MF.1 Interface with low-level controller of the boat, IMU/GPS and Radar.

NREC has provided us an interface to the low-level controller in the form of ROS [1] messages. The path planning subsystem shall send the appropriate commands so that the boat can follow our desired trajectory.

MF.2 Detect static obstacles in the river with minimum size of $2m \times 2m \times 2m$.

The minimum size specified will be adequate for the obstacles that the boat will absolutely need to avoid since it covers anything larger than buoys. The perception subsystem shall extract the obstacles from the radar image.

MF.3 Segment shores and bridges.

Due to the characteristics of the radar, bridges are detected as one large contiguous obstacle in the river. Bridges must be identified and removed from the radar image, leaving all other obstacles in the costmap. In order to achieve this, the perception subsystem shall segment the shores and bridges, so that the bridges can be removed, and shore information can be kept for obstacle detection.

MF.4 Autonomously navigate from source to destination at a maximum distance of 2 miles in not more than 15 minutes.

This requirement provides a guideline for the final validation experiment. The path planning subsystem shall operate the boat autonomously for 2 miles and ensure the boat travels at a reasonable speed.

MF.5 Avoid detected static obstacles with an accuracy of more than 85%.

The system shall navigate autonomously as to avoid obstacles in the river. The perception subsystem must correctly detect the obstacles in the river and then the path planning subsystem must correctly generate a costmap with the detected obstacles and then plan a path to avoid the obstacle. Finally, the boat must be controlled to adhere to the planned path.

MF.6 Update the path within 5 sec after detecting new static obstacles.

The code efficiency will limit the speed of the boat. Thus, the system shall update the path within 5 seconds after detecting new obstacles.

MF.7 Record sensor data up to duration of 15 minutes.

This requirement demonstrates that we are able to record data. We save data from the sensors on the field test and then test our algorithms with the saved data.

MF.8 Replay saved sensor data. This requirement demonstrates that we are able to play back the sensor data. This is also a required function after we record the sensor data.

MF.9 Have a 2D simulator to test the path planning algorithm.

For our path planning subsystem, we first need to run the planner on a static map and then update the path for detected obstacles. Because of the limited number of test trials, we need a simulator to test the path planning algorithm before we can go out for a field test.

MF.10 Generate Occupancy grid map of the environment.

Generating an occupancy grid map of the environment is a requirement of our system, as the SBPL [2] library needs an occupancy grid map of the environment and we need to update the obstacles in the occupancy grid map.

3.2 Desirable functional requirements

The system shall -

DF.7 Track dynamic obstacles with minimum size of $1m \times 1m \times 1m$.

As the mandatory requirement of our system is to detect static obstacles with minimum size of $1m \times 1m \times 1m$, we would like to detect smaller obstacles and be able to track the dynamic obstacles in the desirable functional requirements.

DF.8 Update the path within 2 sec after detecting new dynamic obstacles.

Our mandatory functional requirement for path planning is to update the path within 5 seconds after detecting new obstacles, but we'll try update within 2 seconds in the desirable functional requirements. This will allow the boat to run faster without hitting the obstacles.

3.3 Mandatory non-functional requirements

The system shall -

MNF.10 Be tested in 25-30 field trials.

This is a requirement due to our limited budget and time. We need to test all the functionalities of the boat within 25-30 field trials. As a result, building a simulator to test our algorithms and having a good test plan are required.

MNF.11 Arrive within a radius of 15 meters of the desired destination.

This requirement specifies the accuracy of the path planner. The boat shall arrive at the destination within a radius of 15 meters.

MNF.12 Be demonstrated in a video.

Our sponsor requires a video to demonstrate the functioning of the boat.

MNF.13 Sail at a velocity not more than 15 mph.

As limited by the refresh rate of the radar and the code efficiency, the boat shall travel no faster than 15 miles per hour for the safety of the boat, and other obstacles in the river.

3.4 Desirable non-functional requirements

The system shall -

DNF.1 Obey the "rules of the road".

If possible, the system shall follow the International Regulations for Preventing Collisions at Sea (Colregs).

DNF.2 Estimate the shape of the static obstacles in the river with a minimum of 50% overlap.

In order to better enhance the performance of the path planner, the perception subsystem shall be able to estimate the contour and size of the obstacles in the river.

---

4. Functional Architecture

4.1 Functioning with perception and path planning

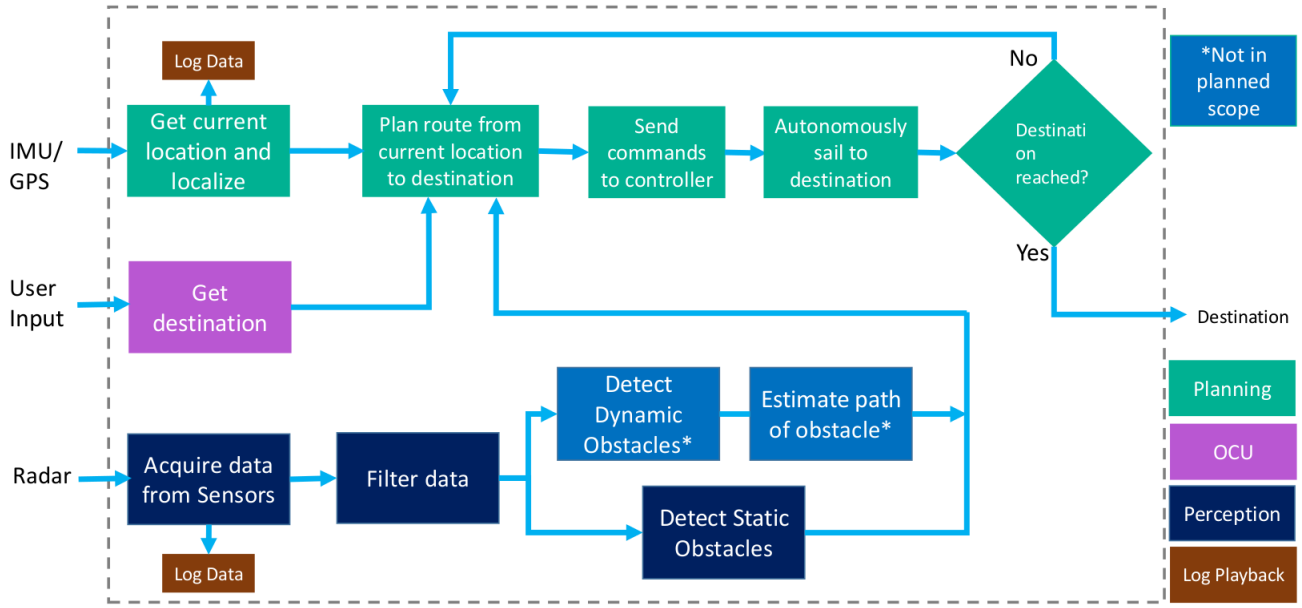

Figure 4.1: Functional Architecture

Figure 4.1 describes the functional architecture of the Autonomous Water Taxi. The functional architecture can be viewed as the building blocks of elements spanning perception, planning, OCU and Log Playback. The desired output of the system is that the autonomous boat reaches the final destination by avoiding obstacles. The input of the system is user input, which specifies the way points and destination, and the data acquired by sensors, which includes INS input that gives data of the current position(GPS coordinates, heading), and Radar to detect the objects.

The first part of the functional architecture is obstacle detection based on Radar. The system acquires data from the Radar sensor, which we can log and play back for further analysis. Since the radar data will potentially have a lot of noise, the next step will be filtering the data to get the useful information that indicates the size and location of the obstacles. For the obstacles, we need to divide the obstacles into two categories, static obstacles and dynamic obstacles and then design different strategies for them, because for dynamic obstacles we need to estimate the path and then decide to follow its path or just avoid it. And we will have an obstacle avoidance algorithm for this part finally.

The second part of the system is Log Playback. Log Playback allows us to log the data from the sensors (RADAR, INS) and then analyze and implement algorithms based on the data. It also provides required information for the OCU module, and finally shows on the user interface.

The third part is the Operator Control Unit (OCU), which allows the user to specify the geometric location, usually the longitude and latitude, of the waypoints and destinations. The idea of the water taxi is that we implement an autonomous driving boat and users are able to use the mobile app to order a taxi service, and the water taxi will pick up passengers at each point and plan a path.

The final part is path planning which gathers data from the INS to get the current position and location of the autonomous boat. The INS has an embedded high-accuracy GPS and digital compass to get an accurate location, which is passed to the path planning module as the input. The path planning function gets the input of the current location, destination and the information of the dynamic and static obstacles. Then it plans a path for the autonomous boat to reach each way point without hitting any obstacles and sends commands accordingly to the low-level controller. Then the following function will estimate the current location and check if the autonomous boat reaches each way point and finally arrives at the final destination successfully. If the autonomous boat has not reached the destination, then it will go back to the path planning function and design a new strategy for path planning, with updated input from the sensors.

4.2 Functioning with simulator and logged data

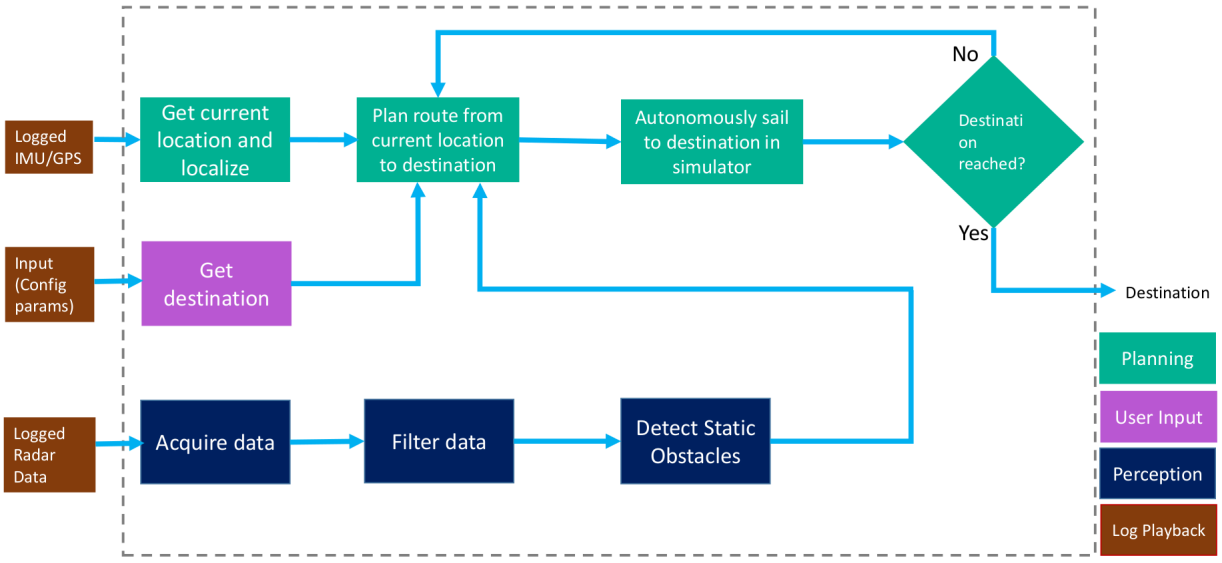

Figure 4.2: Functional architecture of logged data and simulator

Figure 4.2 shows the functional architecture of the simulator. This is very similar to our main functional architecture with a primary difference that it takes recorded data of the radar and INS as the input. In addition to it, the output is the visualization of the path planned rather than the commands sent to the low-level controller to navigate through the waypoints.

5. System-level trade studies

5.1 Sensor Trade Study

Even though our requirement is to focus on radar only, the boat has a mount ready for LIDAR installation, so we decided to perform a trade study on possible configurations we could have with the sensor. The results indicate that we should stick to radar only for now.

Table 5.1: Sensor Trade Study

	Weight	**Radar alone**	**Lidar alone**	**Radar + Lidar**
Ease to install	0.1	9	8	7
Cost	0.2	9	6	4
Code Integration	0.2	7	7	4
Sponsor's Preference	0.3	10	1	5
Output	0.2	7	6	9
Total	1	8.5	4.9	5.6

5.2 Map APIs for the OCU

The Operator Controller Unit (OCU) is a user interface device allowing users to control the autonomous water taxi. The OCU provides visual information and takes commands. The OCU shows the map, location and trajectory of the water taxi and other obstacles around it, A user can input waypoint information which corresponds to the location of a destination port.

Table 5.2: OCU Map API Trade Study

	Weight	**OS Map**	**Google Map**	**QGIS+NE**	**G-Earth**	**Bing Map**
Offline usage	0.2	9	0	6	9	0
User Interface	0.2	6	9	6	9	8
Support/Tutorial	0.2	9	9	8	6	8
Extensibility	0.2	8	8	4	6	6
Coding Language	0.2	9	6	6	6	4
Total	1	8.2	6.4	6.0	7.2	5.2

6. Cyber-physical Architecture

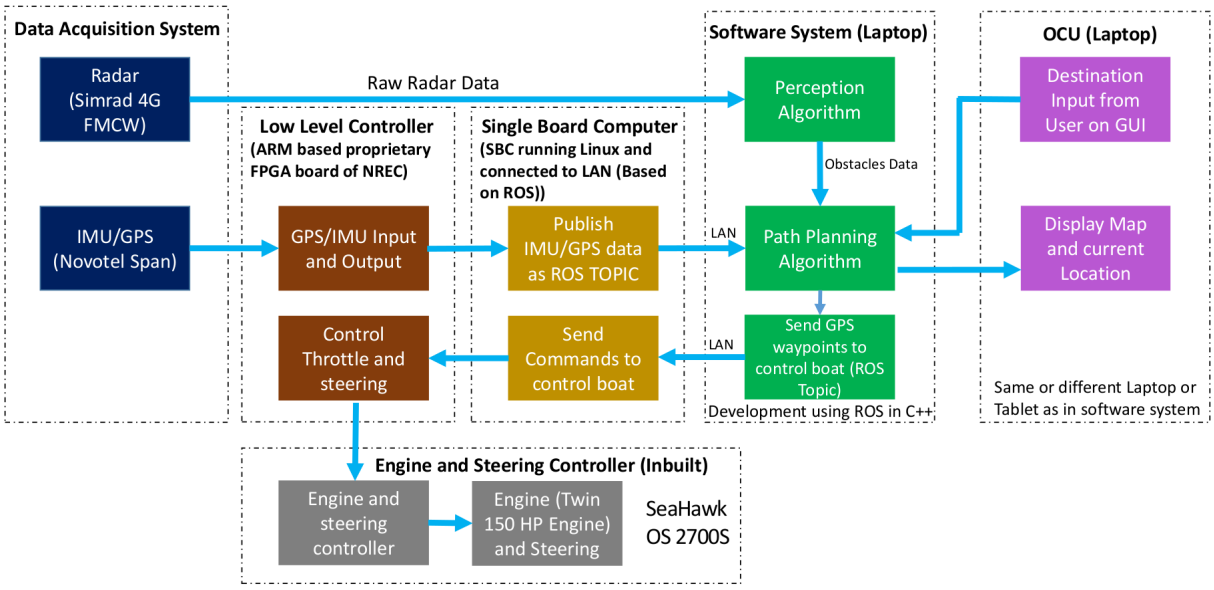

Figure 6.1: Cyber-physical Architecture

6.1 Data Acquisition

As the name suggests, the primary function of the data acquisition system is to acquire data from the various sensors that are outfitted on the boat. Currently, there are 3 sensors that are installed on the boat.

The Simrad 4G RADAR will be used to detect obstacles like ships, bridges, buoys and shores. The Novatel SPAN is an inertial navigation system. It provides us GPS coordinates, as well as velocity and orientation information. There is also a mount on the boat for a LIDAR. For now, as per the requirements of the sponsor, we are not using the LIDAR sensor but we might use it if we do not get the desired results by using only the RADAR.

Once, we have the data from the sensors (RADAR and INS), the next step is to use these data to find the GPS location of the boat using INS data and detect obstacles using the RADAR data. Once we have the location of the boat and the positions of the obstacles are in the environment, we need to plan how the boat navigates to the destination. All these functions are done by the software system. Since this is a software-intensive project, this system is the heart of the project.

6.2 Obstacle Detection

The obstacle detection algorithm uses the RADAR data to detect obstacles and then feeds these data to the path planning algorithm. This subsystem is responsible for interfacing with the radar, and then filtering and processing the data. There are 3 stages in the filtering process. The raw data comes as a polar image, in which data is given by range and azimuth. In order to utilize OpenCV [3] and perform higher-level image processing tasks like morphology operations and contour extraction, we convert our polar image to a cartesian image in the first step. In the 2nd stage, we segment our image so we can identify and remove bridges, and then perform the image processing operations mentioned above. Finally, the remaining obstacles are input into the Octomap ROS package which performs additional filtering. After this, the locations of any obstacles of interest are sent as a message to the planning subsystem.

6.3 Path Planning

The path planning subsystem uses the obstacle data from the obstacle detection subsystem along with the INS data to plan the path. The first step is to incorporate the obstacle information into the current costmap. Depending on the obstacle size and location of the boat, the calculated costs may vary. Next, the costmap will be used by the path planner to generate a safe path to the destination. We have decided to utilize SBPL (Search-Based Planning Library) for the core of our planning subsystem. The library provides several types of search algorithms which allow us to experiment with trade-offs between optimality and speed. Additionally, it allows us to specify motion primitives, in which we can constrain the path planner so that it mimics the dynamics of the robot. We also edited the occupancy grid map so that the boat will stay on the right hand side of the river. Finally, after generating the path, the planning subsystem will send waypoints to the boat in the form of ROS messages so that the low-level controller can navigate to those waypoints.

6.4 OCU

The Operator Control Unit (OCU) provides users visualization of the map and current boat location, as well as final obstacles detected by the radar. Users can input the destination by clicking on the map or choosing predefined waypoints. The OCU will also show the current speed of the boat and how much percentage of the path is completed.

6.5 Low-level Controller

The last system is the low-level controller. Most of the low-level control has already been done by engineers at NREC. We only need to send high-level commands in the form of ROS messages to the engine controller to change the speed and direction of the boat. The low-level controller takes the waypoints from the planner and uses the pure pursuit algorithm to follow the path.

7. System Description and Evaluation

7.1 Subsystem Depictions

7.1.1 Perception

Gathering radar data and segmenting the types of obstacles

Our first challenge in the perception subsystem was to get data from the radar. No one at NREC had used this radar before so we were unsure how difficult the interfacing of radar would be. We found an open source library OpenCPN [4] through which we were able to collect data from radar. We integrated this with ROS as our entire software architecture of perception and path planning is based on ROS. Next, during the field tests we used an RGB camera in addition to the radar sensor to capture the ground truth. Figure 7.1 shows the collected RGB camera and raw radar data. Our next step was to segment bridges, shores and obstacles on the river. We did that by comparing it with the original occupancy grid map (with bridges). The output is shown in the rightmost visualization of figure 7.1. Here. Shores are represented by green, bridges by red, and obstacles by blue.

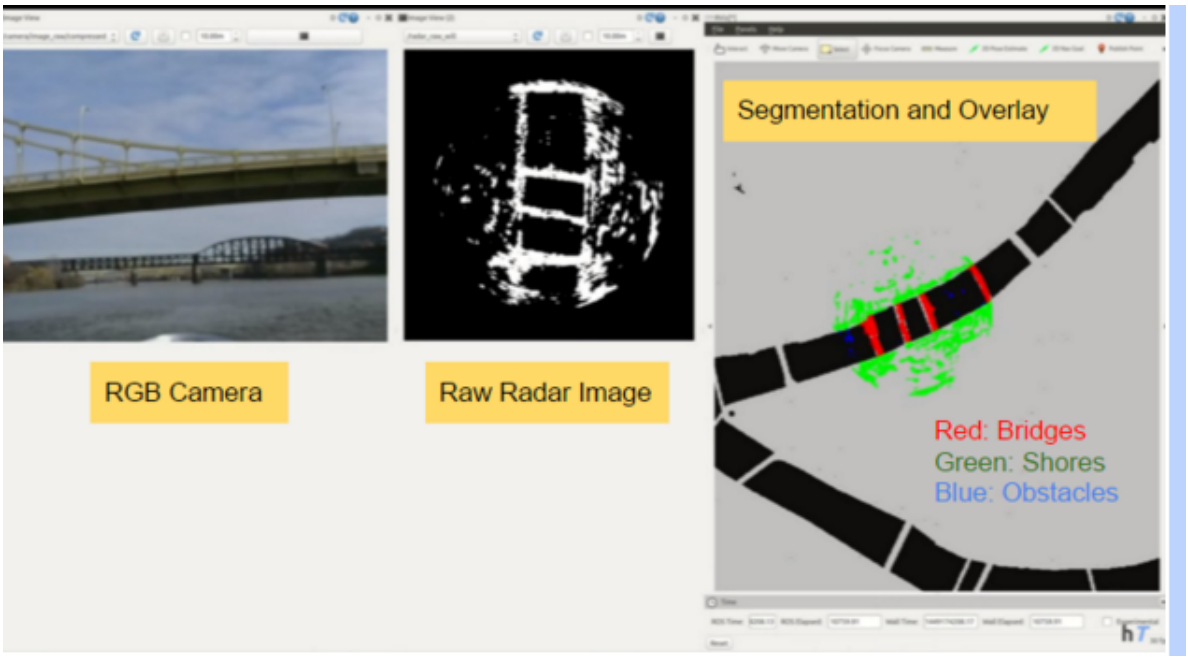

Figure 7.1: Obstacle Detection

Filtering raw radar data

The raw radar data have some false positives which are required to be filtered in order to get the correct location of the obstacles. We used a binomial probabilistic filter [5] to get rid of the false positives. Figure 7.2 shows the result after we apply filters to the data. The center visualization in Figure 7.2 shows raw radar data and right image shows filtered radar data. We can see that a lot of the false positives are not present in the filtered radar data.

In addition to frame-by-frame filtering, we wanted to perform better inference on the data by using all available information over time. To accomplish this, we leveraged the Octomap package from ROS, which generates maximum likelihood occupancy grid maps based on a recursive Bayes filter. The package was not directly suited for our task since it is meant for creating static maps while we wanted the map to update dynamically based on detected obstacles. In order to utilize the package, we first had to prepare our radar data as a point cloud (laser scan), and then tune the model parameters so that the map would react quickly to changes in the perception data. In the end, we were able to achieve a fairly filtered result while not losing information about small obstacles, like buoys, which sometimes generate smaller signatures than noise.

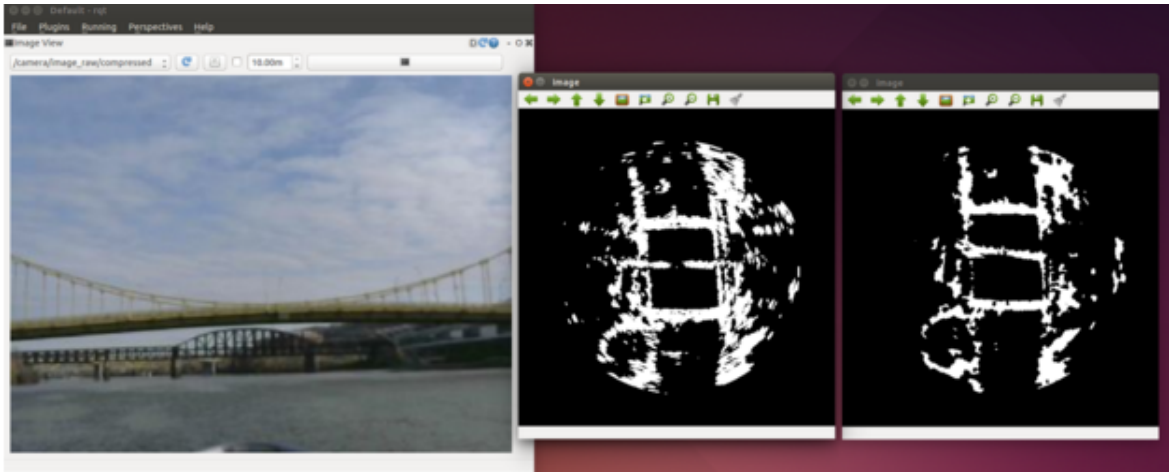

Figure 7.2: Radar Filtering

Putting bounding boxes around the obstacles

In order to measure the accuracy of detecting obstacles, we need to quantify how many obstacles we are able to detect from the given obstacles in the path. For that, we create bounding boxes around the obstacles. This can be seen from figure 7.3 which shows bounding boxes around the blue blobs, which are the obstacles. Two of the big obstacles are marked in figure 7.3. One of them is a big barge. The other small bounding boxes represent detections close to the shores. These are not necessary false positives since there is a lot of vegetation near the rivers that can be plausible explanations, but it does not make sense to include them in our analysis since we need to draw the line somewhere for obstacle detections we are interested in. Thus, although there is no hard and fast rule, we will be ignoring small detections near the shore for the purposes of quantifying our performance, and instead focus on obstacles of interest such as buoys and other boats.

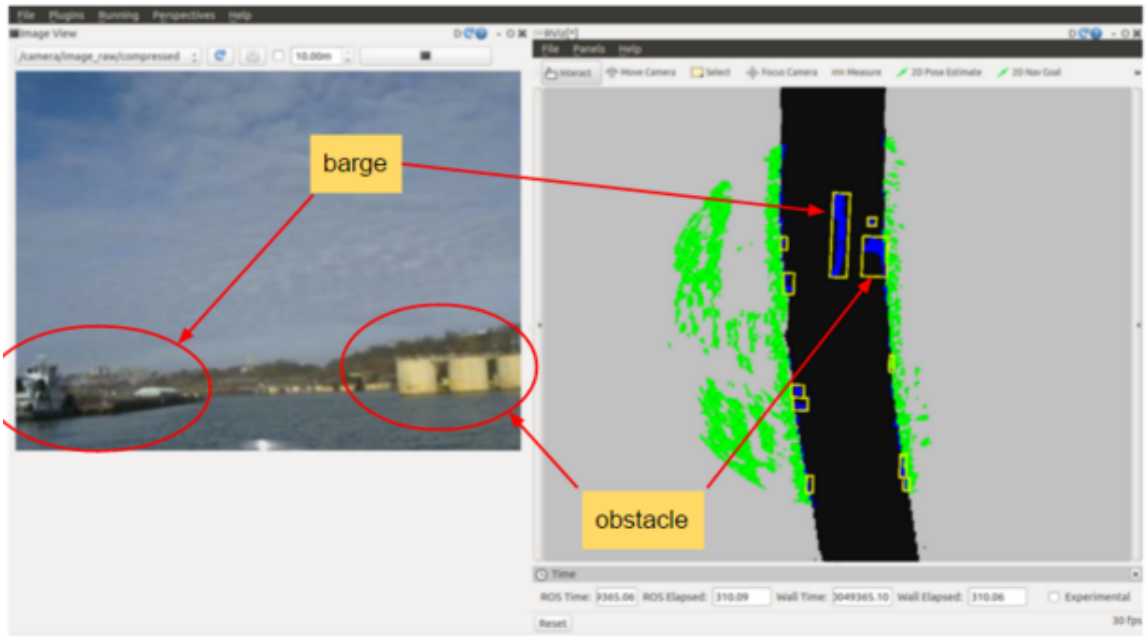

Figure 7.3: Bounding boxes around the obstacles

7.1.2 Path Planning

Occupancy Grid Map

We have created an occupancy grid map (OGM) which the path planning algorithm would be using to find the shortest path between our start position and the desired location. We used QGIS and OpenStreetMap to extract our area of interest i.e. the three rivers flowing through Pittsburgh. Figure 7.4 below shows the OGM where black (zero value) represents free space and white (one value) represents obstacles. The resolution of the map is approximately 5 m.

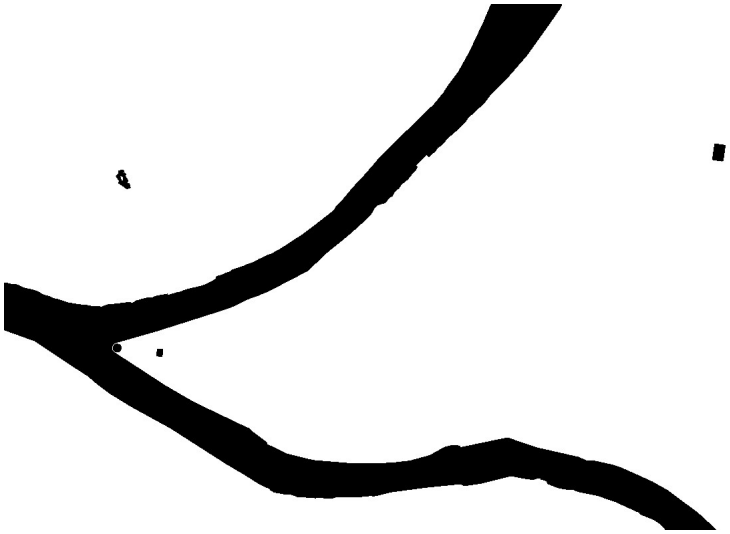

Figure 7.4: Occupancy Grid Map

Inflating the cost near shores and obstacles in the Occupancy Grid Map

The path planner algorithm tends to find the shortest possible path between the start and the goal locations. The causes the generated path to be very close to the shores or obstacles. To overcome this issue, we need to inflate the cost (value of pixels) near the shores and the obstacles. We have created a tool which allows us to experiment with modifying these costs through a simple GUI (shown in figure 7.5). Also, this GUI enables us to add obstacles which we can use to test our algorithms. Figure 7.6 shows addition of obstacles to the image and the inflated costs (in grey) near shores and obstacles.

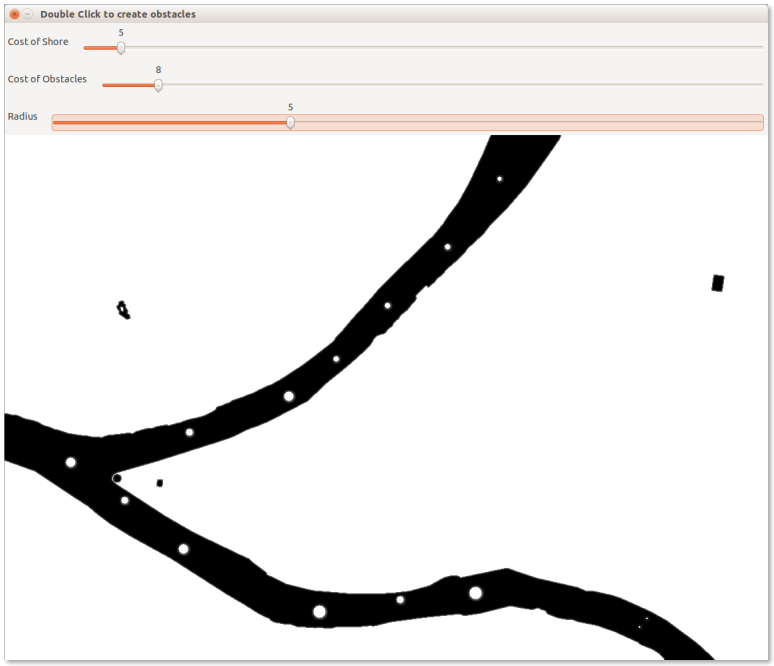

Figure 7.5: GUI to add obstacles

Figure 7.6: Added cost in Occupancy Grid Map

Driving on the right side of the river

Although we were not able to fully observe COLREGS, we can successfully influence the boat to drive on the right side of the river, which is a significant part of following the rules of the road. In order to implement this behavior, we generated two versions of our costmap, one which has high cost if the boat is driving west and another one for driving east. During plan time, we decide which costmap to plan on by first planning on a map with no cost (this is for speed purposes). Based on the beginning of the planned path, we determine which direction the boat will be traveling. With this, we replan the path on the appropriate map, which generates a path on the right side of the river. This strategy can also deal with situations where the boat will travel around a bend and change directions. Since we are constantly replanning, the chosen map will change when the boat changes directions and we detect that our planned path has changed direction in the short term. Figure 7.7 illustrates the two costmaps. Figure 7.8 illustrates how the map and path changes as the boat comes around the river bend.

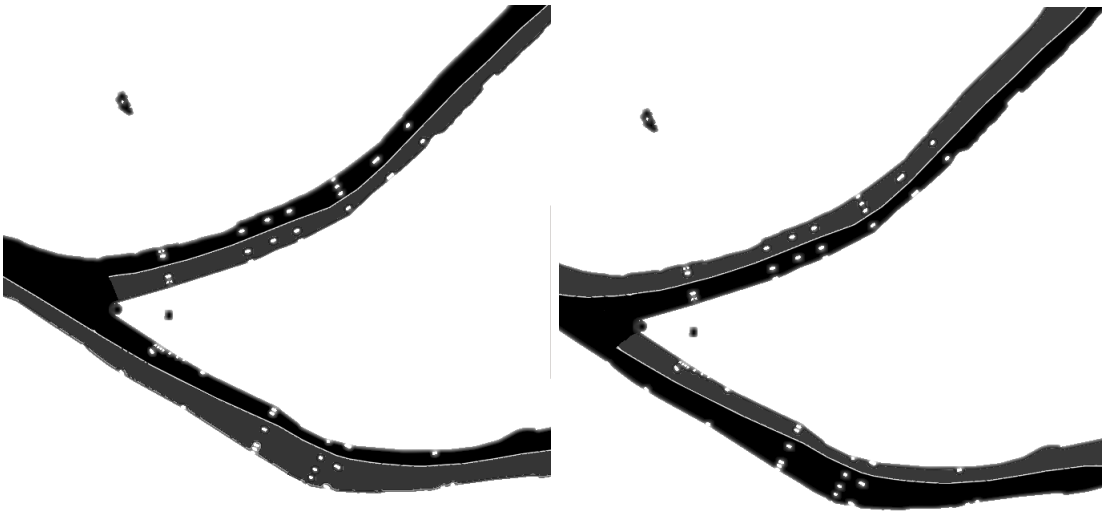

Figure 7.7: Cost Maps for driving different directions

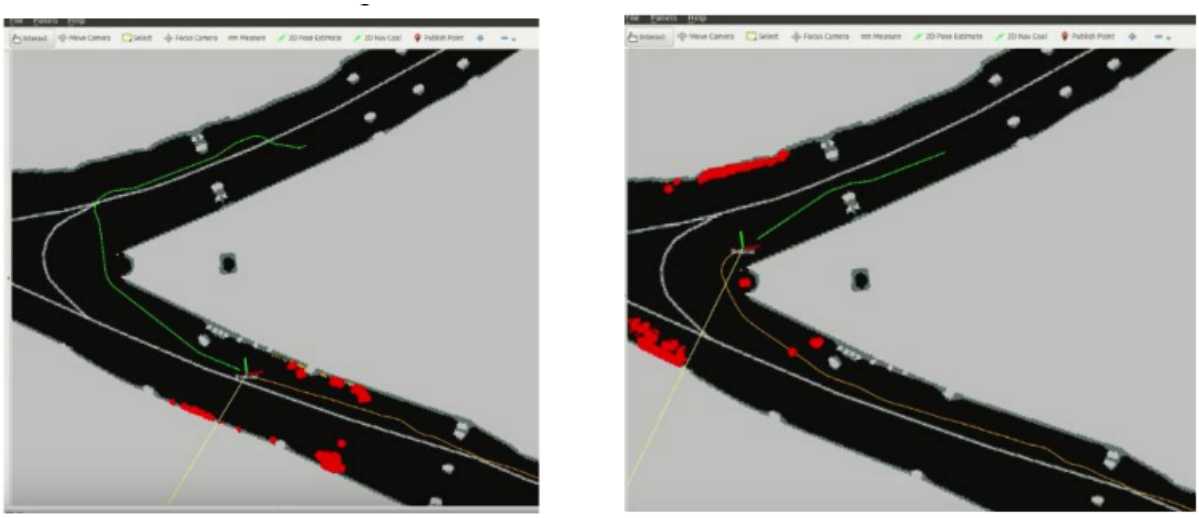

Figure 7.8: Following rules-of-the-road

Path Planning Algorithms

We are using the SBPL library for planning the path. The library has built in planners like ARA* and AD* algorithms. We converted the OGM to the SBPL-compatible format and ran the ARA* planner by providing the start and desired location. The algorithms were tested on small, medium and large size obstacles and the visualization of the results are represented in Figure 7.9, 7.10 and 7.11 respectively.

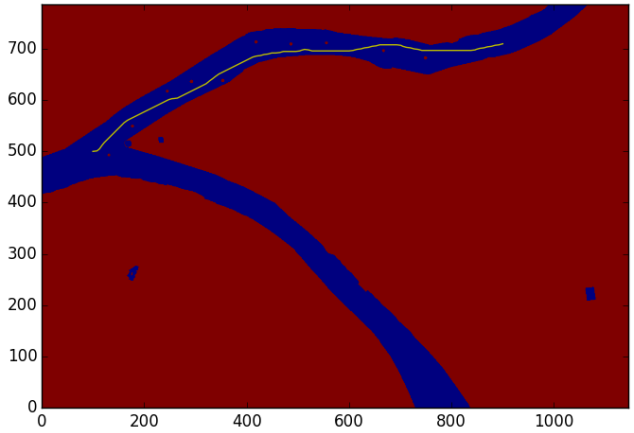

Figure 7.9: Path on Small Size obstacles

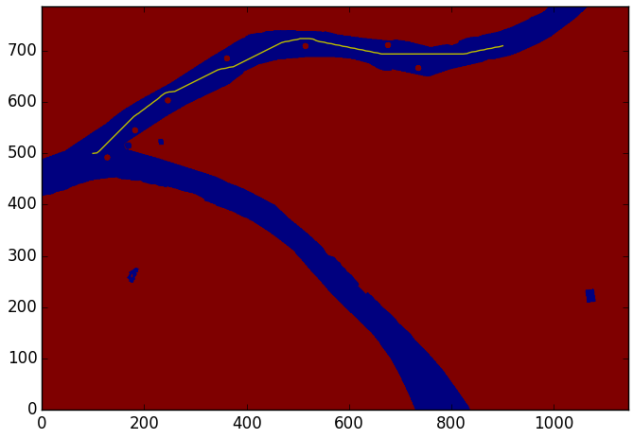

Figure 7.10: Path on Medium Size obstacles

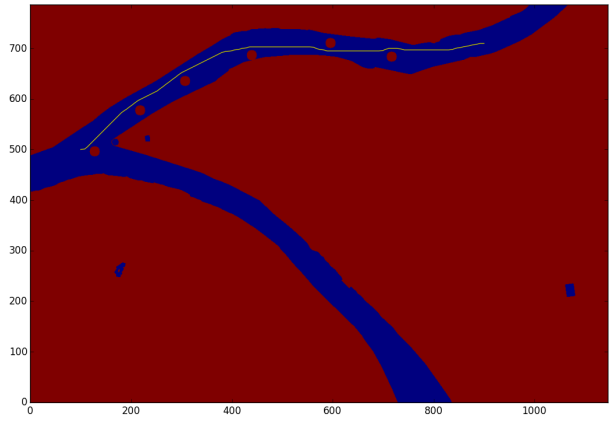

Figure 7.11: Path on Large Size obstacles

Also, we had to tune the motion primitives for the boat by penalizing motions the boat cannot take (for example, 90 degree turns right or left). Before tuning the parameters, the path of the boat is shown in figure 7.12.

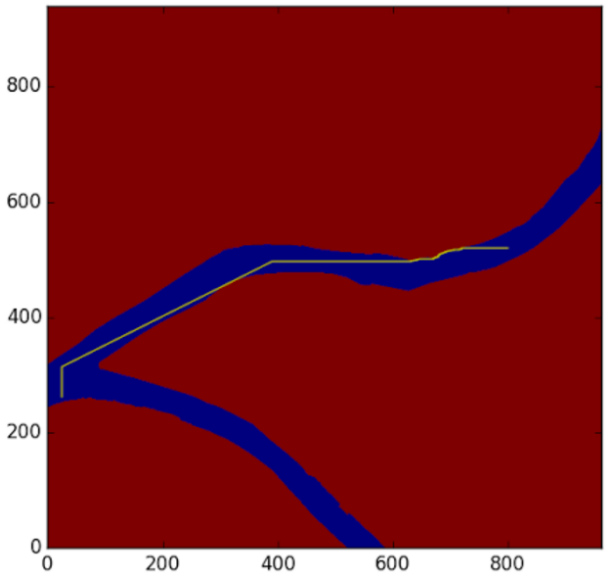

Figure 7.12: Path on Large Size obstacles

7.1.3 Simulation

As we have limited trials and it takes time for us to test the software on the boat, we needed a simulator to visualize the result of our algorithms. So we created a simple simulator on RViz where we subscribe to the data given from the path planner, the current location (SPAN pose), and the map server (figure 7.13).

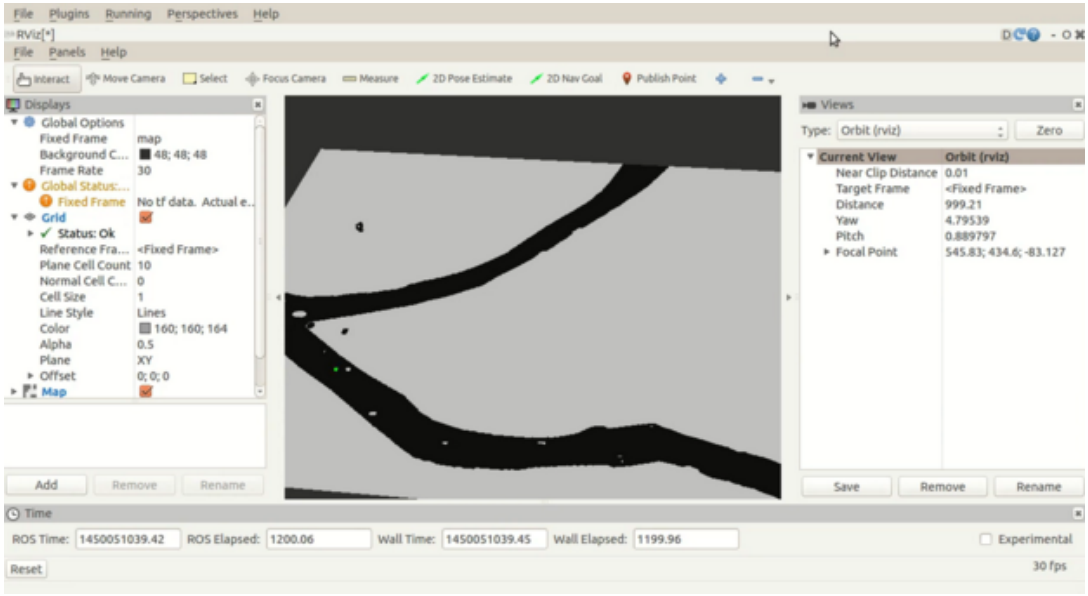

Figure 7.13: RViz Visualization

We used the simulator to –

- Test the path planner
- Tune the motion primitives of the boat
- Test the algorithms while following the rules of the road.

Additionally, we implemented the capability to add simulated obstacles. The following obstacles were added to the simulator –

- *Static Obstacles* – The user can add any number of static obstacles to the environment.
- *Random Dynamic Obstacles* – The user can add any number of random dynamic obstacles to the environment.
- *Teleoperated Dynamic Obstacles* – The user can add one dynamic obstacle controlled using a XBox controller.

Static Obstacles

We used the interactive markers provided in ROS to control the static obstacles. Interactive markers are very similar to the basic markers used in ROS with the additional capability to translate and rotate in the 3D environment interactively.

Figure 7.14 shows the path generated by the path planning algorithm for 2 different positions of the obstacles. There are a total of 10 obstacles. The green line shows the path generated by the path planner.

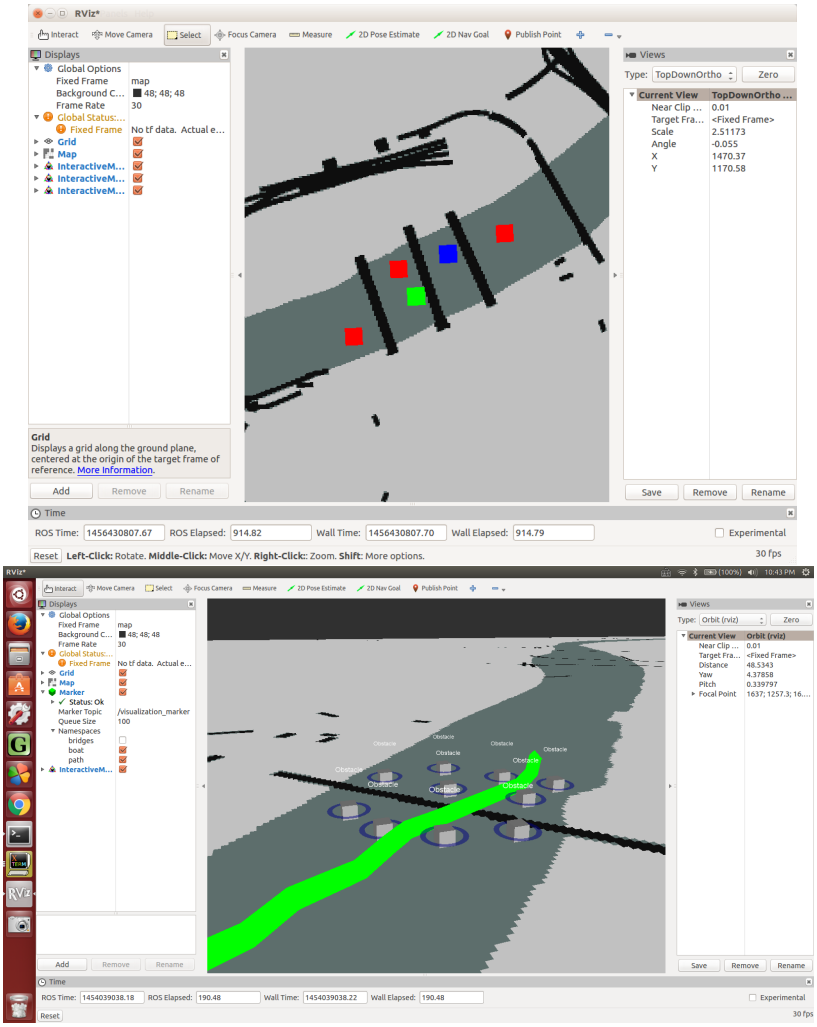

Figure 7.14: Static Obstacles

Random Dynamic Obstacles

The motion of the random dynamic obstacles are simulated in real time using Newtonian equations of motion for a body in a plane. An obstacle is instantiated by specifying an initial position (x_x, x_y) in the environment. The initial acceleration is sampled from an uniform distribution and the object starts to accelerate.

The updates in position (x_x, x_y), velocity $(\dot{x}_x, \dot{x}_y)$ and acceleration $(\ddot{x}_x, \ddot{x}_y)$ are given by the equations 7.1, 7.2, 7.3 and 7.4.

$$\dot{x}_x := \dot{x}_x * t + \frac{\ddot{x}_x t^2}{2} \tag{7.1}$$

$$\dot{x}_y := \dot{x}_y * t + \frac{\ddot{x}_y t^2}{2} \tag{7.2}$$

$$x_x := \dot{x}_x t \tag{7.3}$$

$$x_y := \dot{x}_y t \tag{7.4}$$

During the motion of the obstacles, it is constantly accelerated and deaccelerated based on samples from an uniform distribution. The maximum acceleration that an obstacle can reach is controlled using a threshold.

Teleoperated dynamic obstacles

In order to control the teleoperated dynamic obstacles using a XBox controller, we used the `joy` package available on the ROS website. The `joy` package provides access to nodes that interfaces a generic Linux joystick to ROS.

Collision Detection

One important task while simulating the obstacles was that it should not go out of the river and it should not collide with the banks of the river. In order to implement this, we followed the ROS architecture to avoid collisions. We generated a binary grid map specifying where an obstacle is allowed to move. The value 1 was set for areas where the obstacles could go and 0 where it could not go. Whenever an obstacle is about to enter an area where it is not allowed to, it's direction is changed towards the unoccupied cells.

7.1.4 GUI and Low-level Controller

GUI

The user interface of our system is shown in Figure 7.15. We used ROS Qt and RViz as our visualization tools and the GUI consists of 3 main parts:

- Left: This part gives the user a simple choice of predefined destinations. The user can choose one destination from the drop down list and press GO. Then the system will initialize the planner to that predefined destination. Users are also allowed to see the current speed of the boat and the progress.
- Middle: The middle part provides the user a clear camera view of what's going on in front of the boat. In the particular case of figure 7.15, we can see a boat on the left side of our boat and a bridge further away.
- Right: The right side of the GUI gives user the perception, mapping and planning information based on RViz. First, we load the precomputed map of the 3 river area in RViz. Furthermore, users can see the current position of the boat, which shows as the boat marker. The perception subsystem gives the obstacle information. In the particular case of Figure 7.15, we can see the boat and shores are detected and inflated. The planning subsystem will compute the optimized path, which shows as the green line on the right side of the GUI.

There are two ways for the user to specify destinations:

- Select one of the predefined destinations on the left side of the GUI.
- On the right side of the GUI, click 2D Nav Goal in the menu, and then click one point in the map.

We also integrated voice messages in the GUI. The GUI will inform the user when the boat is en route to a destination as well as when the boat has arrived at the destination.

Low-level Controller

Once we choose a goal from the GUI, it will initialize the path planner and generate an optimal path to the goal. The path consists of several waypoints and by sending the waypoints to the low-level controller, we can control the boat to follow the trajectory.

As shown in Figure 7.16, the low-level controller uses the pure pursuit path tracking algorithm to follow the waypoints. The red point indicates the beginning of the pure pursuit segment and the green point represents

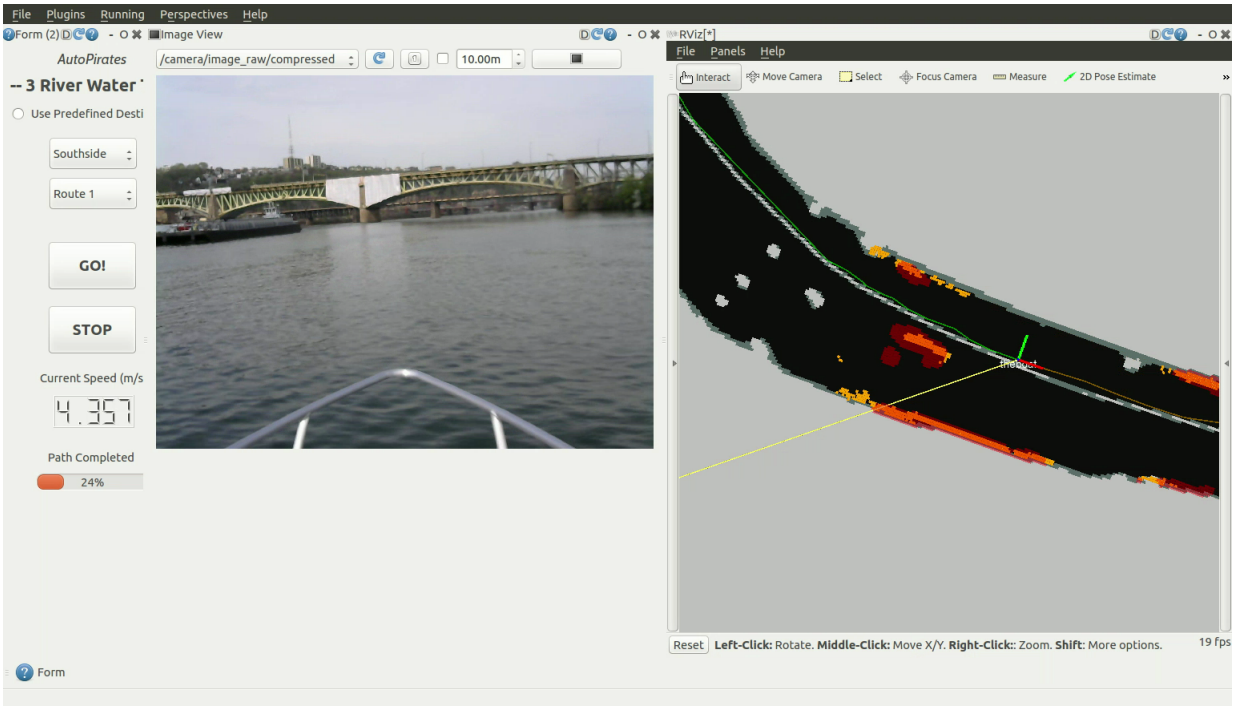

Figure 7.15: GUI

the end of the waypoint. The blue point represents the tracking point in order for the boat to get back onto the trajectory.

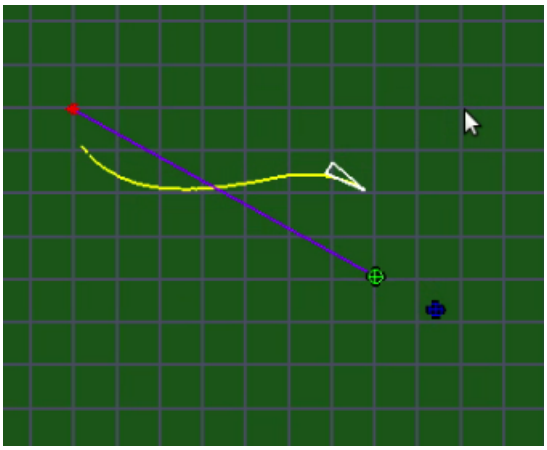

Figure 7.16: low-level Controller

7.2 Modeling, Analysis, and Testing

7.2.1 Radar Analysis

The radar we are working with comes with a display interface installed on the boat. There are many settings we can tune such as gain, range, and scan speed. There are also features such as guard zone monitoring.

Additionally, we can use OpenCPN to control a few of these settings on our laptops (Figure 7.17).

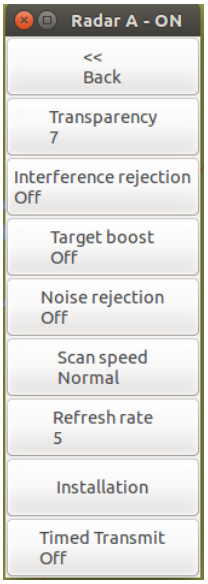

Figure 7.17: Radar Control GUI through OpenCPN plugin

Although we do not fully understand all of the settings and features of the radar, we have experimented a lot with the gain and range settings. For the range of the radar, we have currently set it at 125 meters for all of our logged data so far. We decided on this since we want a higher resolution radar image. However, we must consider the speed at which boats would be traveling the river. If we limit our speed to about 15 miles per hour (7 meters per seconds), and we assume a vessel coming for us at a maximum speed of 45 miles per hour, then an obstacle in the radar image moving at 30 meters per second will still be observed by our system. Currently, we receive radar updates at about 1-2 Hz, which ensures that there won't be an issue of not detecting obstacles. However, additional work will need to be done to develop a tactical path planner which should be able to react almost immediately if necessary. But since most of our development and the spring validation experiment occurred outside of boating season, it did not cause much difficulty.

Considering the gain of the radar, the main trade-off we see is choosing a gain such that we don't introduce too much noise and false positives, while being able to consistently detect small obstacles in the river such as buoys. It is more important that we can detect as many obstacles as possible so we used this idea to guide our gain tuning. Specifically, we made many rounds back and forth at a dock where there were several thin poles in the river serving as the boundaries of the dock. We drove very close to the poles and then slightly further away, while adjusting the gains to make sure we saw sufficient detection and separation in the observed targets.

The 'target separation' feature of the radar is also important for the filtering of our radar data as well as the perception subsystem as a whole. This feature specifies the beam width of each radar scan. Currently, we've set it to the 'High' setting since we would like to be able to distinguish between all the poles we observed in the dock. Additionally, the beam width is crucial for our binomial filter, since the main parameter is specifying how many adjacent cells that we should also check to see if there was a positive return.

7.2.2 Path Planning

A lot of research was done early on to analyze and choose the right path planning library for our project. We chose SBPL (Search-Based Planning Library) since it is relatively easy to use, provides several types of search algorithms, and support is readily available since it is developed here at CMU. Additionally, it

has support in ROS so we can explore and integrate with ROS's navigation stack if necessary. As briefly mentioned earlier, the motion primitives SBPL provides are nice because it gives us a simplistic, but powerful way to define the kinematic constraints of the boat. Although we have not tested it extensively, we've seen in simulation that the paths generated are smooth, and in a couple of live, short distance path planning scenarios during our field test, the boat seemed to adhere well to the generated trajectory.

Another key part of the path planning subsystem is the mapping functionality. We need a fairly high resolution map, and geo-tagged as well so that we can properly position the boat and obstacles for the path planner. QGIS provided us the capability to generate raster maps, and currently we have decided on a resolution of about 5 meters per pixel for our map. This works for us since it is a good balance between having a map that is a manageable size for the path planner, and having enough resolution to be on the same scale as the minimum obstacle size we expect, as well as the boat and our defined motion primitives.

7.2.3 Integration and Testing

Having ROS as our framework made integration and unit testing a lot easier. Generally, our workflow is to develop a small function or feature in isolation as a single ROS node, and then integrate or keep it modular as needed. It greatly speeds up our development since with some recorded data in a bag file, we can develop and test several different perception subsystem functions in parallel. We were able to quickly prototype the new filter as a separate node. We quickly realized that writing the filter in Python would be way too slow, and after switching to C++, the performance greatly improved. Similarly, having logged data have also allowed the path planning development to occur mostly in parallel as well.

7.3 SVE Performance Evaluation

Figure 7.18 shows the script for the Spring Validation Experiment for the whole system.

We were successfully able to demonstrate all of the experiments that we promised for the SVE. Here's the summary of our performance:

- Path: We had the test run from Southside to PNC Park and from PNC Park to Washington's Landing. The total path length was around 10 Kilometers.
- Velocity and time check: We were moving with an constant velocity of around 4.5 m/s and we were able to cover the distance of 2 miles in less than 15 minutes (which was stated in our requirements)
- Re-planning: The system was constantly re-planning during the test run and the re-planning time was less than 5 sec (as stated in the requirements)
- Rules of the road: The boat followed the rules of the road by making sure that it was always on the right side of the river.
- Obstacle avoidance: We encountered various obstacles during the path including shores, 1 barge, 8 buoys, and bridge pylons. We are able to avoid all of them successfully.
- Crossing bridges: We were also able to navigate under the bridges successfully.
- GUI: The GUI gave the functionality to select from pre-determined destinations and also the ability to select any destination in the river by clicking on the map. All these features worked perfectly during SVE.

Step ID	Description	Success Criteria	SVE	SVE Encore
1	Turn on the navigation computer and calibrate INS.	We can ping the master computer and the INS is correctly calibrated.	Successful	Successful
2	Run OpenCPN and get the radar data.	Check the raw radar image	Successful	Successful
3	Run the launch file to start the navigation system	Check RViz to see where the boat is and check radar data is properly overlaid.	Successful	Successful
4	Add fake obstacles on the map	We can move the interactive marker in RViz and the path goes around the fake obstacle.	Successful	Successful
5	Put the boat into auto mode. Then do one of the following: - Select one of the 3 destinations (PNC, Southside, NREC) in the GUI. Press 'START' on the GUI. - Click a goal on the map in RViz	We can see a planned path in RViz and the boat starts moving.	Successful	Successful
6	Start recording rosbag files and stop recording when the boat arrives destination	We can replay the bag file after and see the trajectory.	Successful	Successful
7	The boat detects the static obstacles on the river	- 80% of Obstacles that are 2*2*2 meters will be detected and shown in RViz - obstacles include: buoys and objects near shores (such as docks, poles, or containers)	Successfully detected buoys, shores and barges	Successful
8	The path planner generates a safe path	- The boat maintains a safe distance from obstacles & stays on the right side of the river. - If the boat needs to cross sides, it will do so immediately, unless the goal is on the left side. In this case, it will cross at the end.	Successful path, following rules-of-the-road, osillation in some path	Discussed small path oscillation problem
9	The boat navigates successfully to the destination	The boat arrives within 15m of the destination	Stopping issue because of missing waypoints	Successful

Figure 7.18: SVE Performance Evaluation

7.4 Strong/Weak Points

7.4.1 Strong Points

1. We were able to get continuous technical help from NREC. It enabled us to cruise through any road blocks that we faced.
2. The INS on the boat is military grade and the performance of the radar is better than we were expecting initially. High accuracy of the INS is saving us from doing extra work in doing fusion of the data from the INS and radar.
3. The hardware in the boat, including the low-level controller and Single Board Controller are running ROS nodes that are robust and highly reliable. This saves a lot of time for us.
4. We are using the SBPL library for path planning which has been developed by a research group here at CMU. Any challenges related to it are easily solved using the help from the SBPL team.
5. We are using ROS as the platform to communicate between different modules. Also, we need to log and play back the data collected from sensors on the boat. Here, built-in tools of ROS like ROSBAG

and the navigation stack are helping us a lot in the development process.

7.4.2 Weak Points

1. Radar only gives obstacles data in 2D format. So it is not possible to segment bridges, pillars of the bridges and nearby obstacles. Also, this means that there is no way for us to detect obstacles under the bridge. Moreover, since we remove the entire contour associated with the bridge (not just where it directly overlaps with the ground truth), we cannot deal with barges under bridges as well. The barge will appear to be connected to the bridge and our algorithm will remove it as well. For our project, we have assumed that we will not deal with any obstacles near the bridges.
2. ROS doesn't have good support for radar. We had to interface the radar data with ROS on our own. To use some packages which expected laser point clouds, we needed to create custom interfaces to give ROS what it expected.
3. Planners from the path planning library take around 3-4 seconds to find the shortest possible path. This is too slow and it is not practical considering that we cannot travel faster than about 10 MPH.

8. Project Management

8.1 Schedule

Table 8.1: Schedule for Fall Semester

S No.	Task	Start Date	End Date	Category
1	Acquire data using OpenCPN	9/13/2015	10/12/2015	Perception
2	Interface with IMU	9/13/2015	10/12/2015	Path Planning
3	Interface Radar with ROS	10/12/2015	10/29/2015	Perception
4	Build and explore SBPL	10/12/2015	10/26/2015	Path Planning
5	Basic Data Filtering	10/29/2015	11/07/2015	Perception
6	Generate Occupancy Grid Map	10/26/2015	11/10/2015	Path Planning
7	Morphological Operation	11/07/2015	11/14/2015	Perception
8	Test Occupancy Grid Map on simulator	10/10/2015	11/13/2015	Path Planning
9	Contour Extraction	11/14/2015	11/18/2015	Perception
10	Run planner on OGM to plan the path	11/13/2015	11/24/2015	Path planning
11	Object Detection (Boundary box)	11/18/2015	11/24/2015	Perception
12	Create waypoint following interface	11/24/2015	11/28/2015	Path Planning
13	Start with overlaying radar data on maps	11/24/2015	12/01/2015	Perception
14	Tune Motion Primitives and OGM	11/28/2015	12/02/2015	Path Planning

For the fall semester, we were successfully able to meet the goals which we had set for the team.

Table 8.2: Schedule for Spring Semester

S No.	Task	Start Date	End Date	Category
1	Test path planner algorithm on boat	01/13/2016	01/20/2016	Path Planning
2	Improve radar filtering	01/13/2016	01/28/2016	Perception
3	Improve costmap functionality and path planning simulation	01/20/2016	01/27/2016	Simulation
4	Create GUI to enter destination	01/27/2016	02/03/2016	Path Planning
5	Quantify and measure radar efficiency	01/28/2016	02/03/2016	Perception
6	Dynamically generate OGM	02/03/2016	02/17/2016	Perception
7	Integrate radar obstacles with path planner	02/03/2016	02/23/2016	Integration
8	Dynamically plan the path (Global Planner)	02/17/2016	02/23/2016	Path Planning
9	Dynamically Plan the path (Local Planner)	02/23/2016	03/09/2016	Path planning
10	Improve perception efficiency	03/09/2016	03/20/2016	Perception
11	Improve path planning efficiency	03/09/2016	03/20/2016	Path Planning
12	Identify and practice exact scenario for SVE	04/01/2016	04/20/2016	Integration
13	Buffer to complete miscellaneous items	04/20/2016	End of Semester	Integration

In the spring semester, we more or less achieved most of the tasks we had outlined at the end of the fall semester (shown in Table 8.2). However, there were a few things we shifted around. Since we decided that we should focus on the integration and testing to make sure we were well prepared for the SVE, we shifted the tasks of GUI development and implementation of a local path planner to the end of the semester. While these tasks were important for meeting our final goals, they were not on the critical path to getting the core functionality of our system completed. Thus, we reorganized to make sure we could complete our most important requirements.

8.2 Budget

Table 8.3 shows the hardware and any equipment that is provided by our sponsor (NREC). We are provided with the 2015 SeaHawk OS 2700S boat, the SimRad 4G FMCW Radar and Novatel SPAN with GPS positioning. We also had up to 25 trials for field tests.

Table 8.3: Hardware/Equipment/Budget provided by the sponsor

S.No	Part	Part Name	Quantity
1	Boat	2015 SeaHawk OS 2700S	1
2	Radar	SimRad 4G FMCW Radar	1000
3	IMU/GPS	Novatel SPAN with IMU positioning	1
4	Laptop	Dell i7	5
5	Proprietary Boards	SBC and Microcontroller boards	2
6	Trials for testing	$500 per trial	25 (trials)

In the fall semester, we went on two field tests. In the spring semester, we went on five more field tests. This also includes the SVE.

Table 8.4 shows our MRSD project budget.

Table 8.4: MRSD Project Budget

S.No	Category	Description	Qty.	Cost	Total	Spent	Left
1	Transportation	Travel to NREC (5 people)	64	25	1600	400	1200
2	Videography	Professional video for NREC	1	500	500	0	500
3	Backup for Trials	In case we run out of allotted 25 trials	2	800	0	0	0
4	Hardware	GPU to test algorithms	1	1600	1600	1500	100
5	Miscellaneous	To buy camera, contraption, books, GPS, USB mouse/keyboard, etc	1	300	300	270	30

We have a $ 4000 budget as part of the MRSD project course. We had initially budgeted our funds for transportation, videography, and backup funds for field tests. However, we ended up only using 7 out of our 25 allotted field tests and well underspent our transportation budget. Also, we didn't make arrangements to shoot a professional video, but NREC should be OK since the film crew from '60 Minutes' captured some nice video. Due to our surplus, we decided to purchase a GPU. This will help future teams push the performance of the system since the whole process is very computationally expensive.

Aside from these major expenses, we have spent $ 270 on miscellaneous items. These include a book about programming robots with ROS, a GPS for testing, and various computer peripherals for easier development.

8.3 Risk Management

We identified seven major risks in the Preliminary Design Review (PDR) and discovered two more risks as we went on the field tests. One issue is that the radar is unable to detect bridge supports and the other one is the code efficiency.

Table 8.5: Risk Management

ID	Risk Title	Req	Type	Description	Likelihood	Consequences	Mitigating Actions
1	Test environment availability	All	Testing	Weather conditions delay the field test	4	3	Test radar perception on shore for static obstacles. Use a simulator for path planning.
2	Limited testing trials	MNFR2	Project management	Only 25-30 field tests	5	3	Appoint a field manager to plan logistics of each test.
3	Radar Performance	MF2 MF3 DF1 DF2	Technical	Radar signal is too noisy	3	4	Incorporate additional sensors like Lidar and camera
4	Localization accuracy	MF4 MF5	Technical	Path planning algorithm relies on precise localization given by GPS	3	5	Filter GPS data, Limit the maximum speed of the boat.
5	Insufficient boat protection	All	Testing	No automatic stop in case of a possible collision	2	5	Develop fail-safe software modules.
6	Technical support from NREC	All	Technical	Limited availability from NREC engineers	4	3	Ask for software documentation.
7	Team Communication	All	Project management	Language barrier in the team	4	2	More preparations before the meetings.
8	The Radar only has 2D information	All	Technical	Can't plan path without information of bridge supports	5	5	Research nautical maps that has information for the bridge supports
9	Code Efficiency	All	Technical	Code may run too slow after integration	5	4	Integration early and often to identify bottlenecks.

This is how we responded to the risks:

Risk ID 1: Thanks to the warm winter, we did not face many issues when scheduling field tests. We also mitigated any potential issue by checking the weather frequently and planning tests weeks ahead of time.

Risk ID 2: We have logged radar sensor data and INS sensor data of the boat while doing our field tests. As a result, we were able implement and test our algorithms using the logged data. It helped us minimize the field test trials.

Risk ID 3: Although the radar data had lots of noise on the shore, the radar noise was significantly decreased on the river. For improving perception performance more, we have applied several filters to our radar signal.

Risk ID 4: Until now, the INS accuracy for boat localization was not as bad as we expected. We've been tracking this issue while implementing the path planning algorithm.

Risk ID 5: Our safety driver, Mike, did an excellent job in allowing us to push the boat to its limits while being ready to stop at any second.

Risk ID 6: For path planning, we found that there is an SBPL software researcher on campus, so we've been taking support from him in path planning. In addition, we have support on how to use the existing functions of the boat for doing field tests from NREC engineers.

Risk ID 7: Our team has regular meetings for communication to overcome the language barrier.

Risk ID 8: We were able to find topological information for the bridge supports and add it to our occupancy map.

Risk ID 9: As there is a lot of software running in the full system, the code effiency somewhat degrades as the boat is running. We recognized some of the issues and came up with quick fixes, but future work will need to be done to address the problem accordingly.

The risk likelihood-consequence table is shown as figure 8.1.

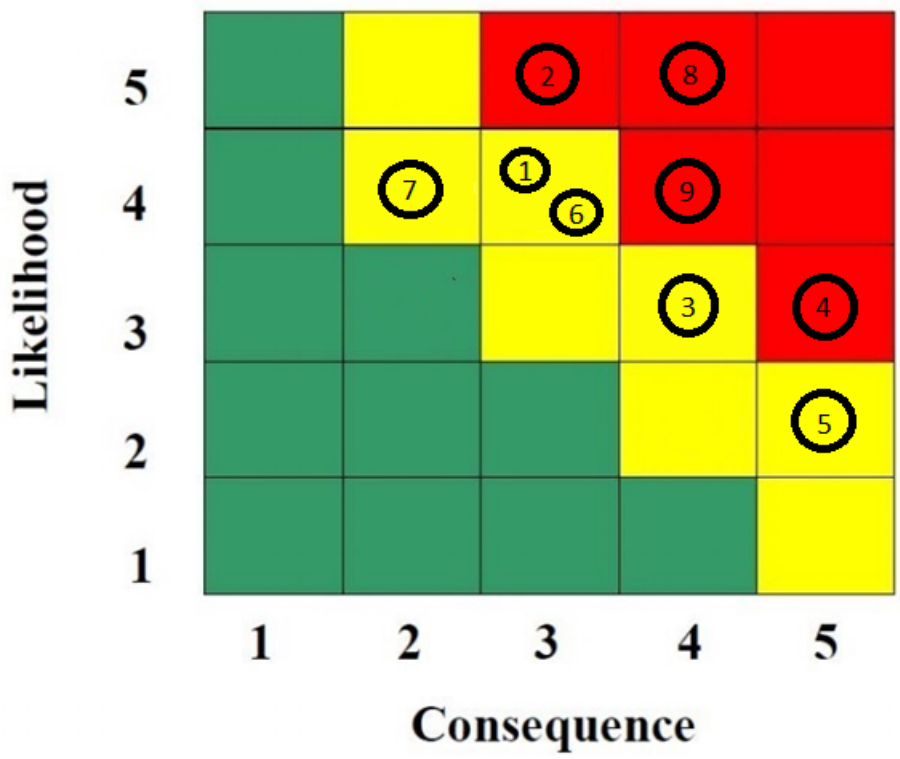

Figure 8.1: Risk Likelihood-Consequence Table

9. Conclusion

9.1 Lessons learned

Our team has learned several lessons after experiencing the design and embodiment of the autonomous boat system like the following.

9.1.1 Technical lessons

1. Field test: Since field tests are one of the important parts of our project, well-planned field tests helped us adhere to our schedule and provided us more time to develop the system. We made a field test plan every time before a field test and made sure we recorded all the data to analyze afterwards.
2. Analyzing the data: Analyzing the data helped us figure out where the problem could be when the system performed bad. For example, there was a case when the boat kept driving towards the shore during our field test. To find where the problem was, we used a ROS package, rqt_bag to visualize the cycle of ROS publishers and noticed that one topic stopped publishing for some time. We then concluded that this was because we were overloading the on-board laptop. We also looked at the GUI and realized we were sending incorrect waypoints. By digging into the code we found that the mismatched buffer size caused the loss of waypoints.
3. Simulation: As we had limited opportunities to go on field tests, it was very important to simulate the system. We tested the path planning using fake obstacles and also used a lot of recorded data to do further testing.

9.1.2 Project management lessons

1. Prototyping: Rapid prototyping and adhering to the agile development process helped us stay on schedule while figuring out the scope of the project. In the beginning, it was tough for us to commit to design decisions as none of us had done a project like this before. By quickly testing prototypes, we were able to feel out the schedule better later in the development process.
2. Code review: Code review is important. Code written by the original author may have some bugs. The other teammates could find these bugs in a code review, which would result in early discovery of bugs. Also, with constraints due to scheduling field tests, a code review process helps teammates spend less time overall with finding errors and problems in the code.

9.2 Future Work

We believe that the low-hanging fruits are all but picked and there will be a steep increase in the level of difficulty if some team wants to work on the project next semester. We have basic autonomy capability

ready and an incremental approach is needed to make the system robust. Some of the possible work that can be done in future is discussed below –

- Currently only static obstacles can be detected and avoided. Although the current implementation can cope well with slowly moving barges, the algorithm does not detect fast moving obstacles like motorboats. This is partly because the radar update rate is slow. The latency in re-planning time further aggravates the problem. In order to solve this problem efficiently, installing a new sensor on the boat might be the right thing to do and testing other state-of-the-art planners like D* Lite, which according to literature can run faster than our current planner which is ARA*.
- A challenging, yet interesting task is autonomously docking the boat. Docking the boat manually is itself a daunting task and it requires multiple tries even from the safety driver. We are not sure what's the right approach to tackle this problem.
- Another limitation of our approach is that we cannot detect any obstacle under the bridges. It is assumed that there are no obstacles under the bridges. This is because the radar data is 2 dimensional, so using a laser range scanner might be the right thing to do.
- Currently we only have a global planner running, which is partly the reason for the slow re-planning. Although we did implement a simple local planner before the SVE Encore, we only tested it in simulation. So, implementing a local planner on a high resolution map and a global planner on a low resolution map is expected to make planning faster.
- Following the rules of the road was one of our extended goals and we were successfully able to demonstrate driving on the right side of the channel. However, there are still many rules relating to multiple boats in head-on situations. This can be an interesting problem to solve, but will be extremely difficult and dangerous to test on the real boat, so a lot of testing will be required in the simulator. The current simulator is fully capable to test this problem.
- The code can be improved in terms of speed as a lot of execution can be done in parallel on GPUs. An example is a lot of OpenCV code can be executed on GPUs. Convolution operations can be sped up using GPUs. Indeed, we bought a GPU from our team budget so that it could be used on the project next semester.

References

[1] Robot Operating System
http://ros.org/

[2] SBPL Reference
http://sbpl.net/

[3] OpenCV
http://opencv.org/

[4] OpenCPN
http://sourceforge.net/projects/opencpn/

[5] Schuster, Michael, Michael Blaich, and Johannes Reuter. "Collision Avoidance for Vessels using a Low-Cost Radar Sensor." World Congress. Vol. 19. No. 1. 2014.