RMIT UNIVERSITY

Project Corona CleanTeam
Software Development Report

Version: V1.0
Date: 14/10/2020
Sponsor: James Harland
Number: External
Authors: Susanna Huang, Thomas Pfundt, Bryce George

User Guide

Introduction

This user guide describes how to setup and execute a simulation of a MIR100 Robot running a cycled delivery of hand sanitizer per COVID Safe recommendations. We use the MIR100 Robot from the RMIT Virtual Experiences Laboratory (VXLab) named Blue.

In the simulation, Blue roams the VXLab searching for human models. Blue begins by moving towards static navigation goals (coordinates), while the Human Detection System looks for targets (humans). If a target is identified, Blue's navigation goals are updated. Blue's own system handles navigation to requested goals, including object avoidance and pathfinding.

The Human Detection System is trained to distinguish between humans and other objects as represented by depth sensor data. Blue can "see" the world around him through the depth sensors on the attached Realsense R200 camera. The coordinates of a detected target human are stored in the Delivery Cycle and passed to Blue as a navigation goal. Blue will then visit each human to simulate the delivery of hand sanitizer, visualised by a hand sanitizer model carried by Blue. The Delivery Cycle keeps track of which humans have been visited, and will prevent Blue from revisiting them within a certain time frame.

A demonstration video of the simulation can be found here:
https://drive.google.com/file/d/1pbgYpISdYYJxAhpS5y4V_yZZRbDFFRhJ/view

This simulation was developed as a capstone project at RMIT in Programming Project 1 (COSC2408). The project utilises RMIT Virtual Experiences Laboratory (VXLab) resources for collaborative experimental design, operation and testing of the robotics systems. Due to COVID-19, our project was developed in a simulated environment called Gazebo, which was configured, hosted and supported by the VXLab team. Although the project is focused on Covid Safe activities, it also enables a much larger range of possibilities for the VXLab. The project provides a base to expand upon, which will enable the VXLab to program the robots to detect objects of a given type, and move the robots to dynamic locations based on various sensor feedback.

Simulation Feature List

- Navigation via Robot Operating System (ROS)
- Custom Model creation and placement
    - Human models
    - Hand Sanitizer bottle
    - Realsense R200 sensor
- Generation of 3D Point Cloud through imported sensor

- Human Model Detection - Trained with generated Point Cloud Data files to detect humans
- Delivery Cycle - Visited humans are recorded in memory to not revisit visited humans

Setup Manual

Requirements

- Docker
- Docker Compose
- Capable machine - Simulation requires some beefy hardware, especially containerized
- Linux and terminal usage knowledge is assumed

Step 1 - Running Docker

Inside the root directory of our repository (https://github.com/s3724447/PP1-baxter), the docker-compose.yml file defines configuration for multiple docker containers required to run the robots and simulation environment required for this project.

Before running the containers, they must first be built. To build the containers use the command: `` `docker-compose build` ``

This process will take a long time, approximately an hour the first time it is run, as each container will be downloading and installing the required dependencies.

Before running the containers, the docker network required for the containers to connect to each other must be created by running the script: `` `./create-vxlab-network` ``

Once containers and network are built successfully, the containers can be started by running: `` `docker-compose up` ``

The containers that will be started are:

- 1 container for the noVNC output
    - Used by all other containers to show the required graphics applications.
- The master container - vxlab-rosie
    - Houses the main Gazebo simulation and also the ROS master node for controlling Rosie.
- The navigation container - vxlab-rosie-nav
- The MIR100 (Blue) container - vxlab-blue
    - Contains the ROS node for Blue, and also processes all target detection.

Step 2 - Opening noVNC window

In a web browser, navigate to http://localhost:8080/vnc_auto.html. Once the containers have started, the Baxter control panel for controlling Rosie's arms will be visible in the noVNC window. This window is the graphical interface used for accessing Gazebo, RVIZ and the Robot control panels.

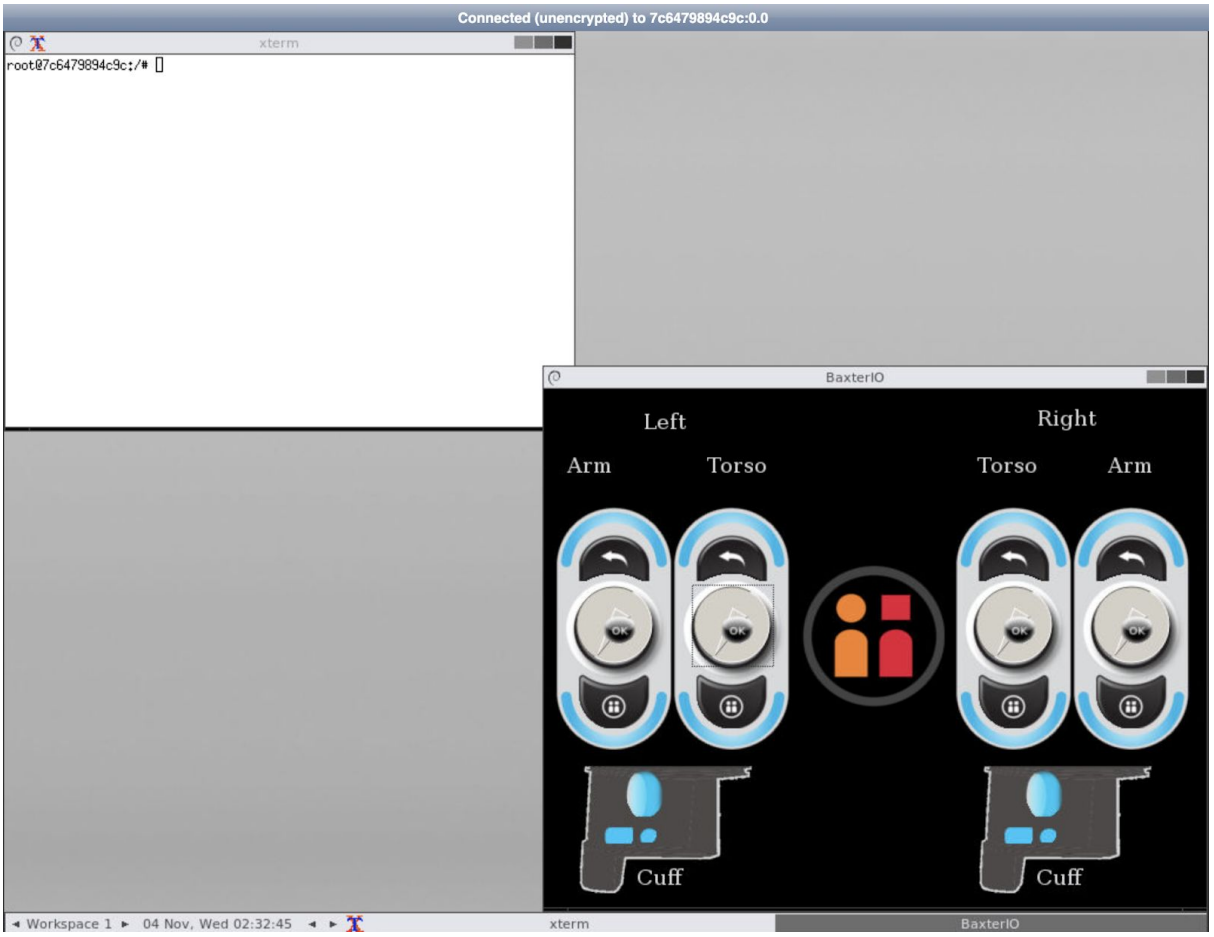

Figure 1: Starting noVNC display

Step 3 - Accessing Docker containers to start services

Containers can be accessed by using the `docker exec` command. This access to the container provides users with an interactive terminal with a root access bash prompt, required to start programs and services. To gain access into the primary container for example, run the command: `docker exec -it vxlab-rosie bash`

```
coronaclean@thor12:~/baxter-mobility-base-simdemo$ docker exec -it vxlab-rosie bash
root@18f9eb360c98:~/rosie#
```

Figure 2: Connecting to a container with docker exec

Step 4 - Starting Gazebo

To start gazebo, first connect to the vxlab-rosie container as per step 3. Once connected, start the Gazebo client: `` `gzclient &` ``

Return to the noVNC window and you will see the simulated VXLab environment, with Rosie and the human models placed around the lab inside the client. You can click & drag and shift+click & drag to move and rotate the view around the simulation.

Gazebo is the simulation for the project, and as such, can be used to interact directly with the robots and environment. Models can be moved around by adjusting the pose parameters on each model in the sidebar. You can click on individual models to select them (including the robots) and change properties on the fly.

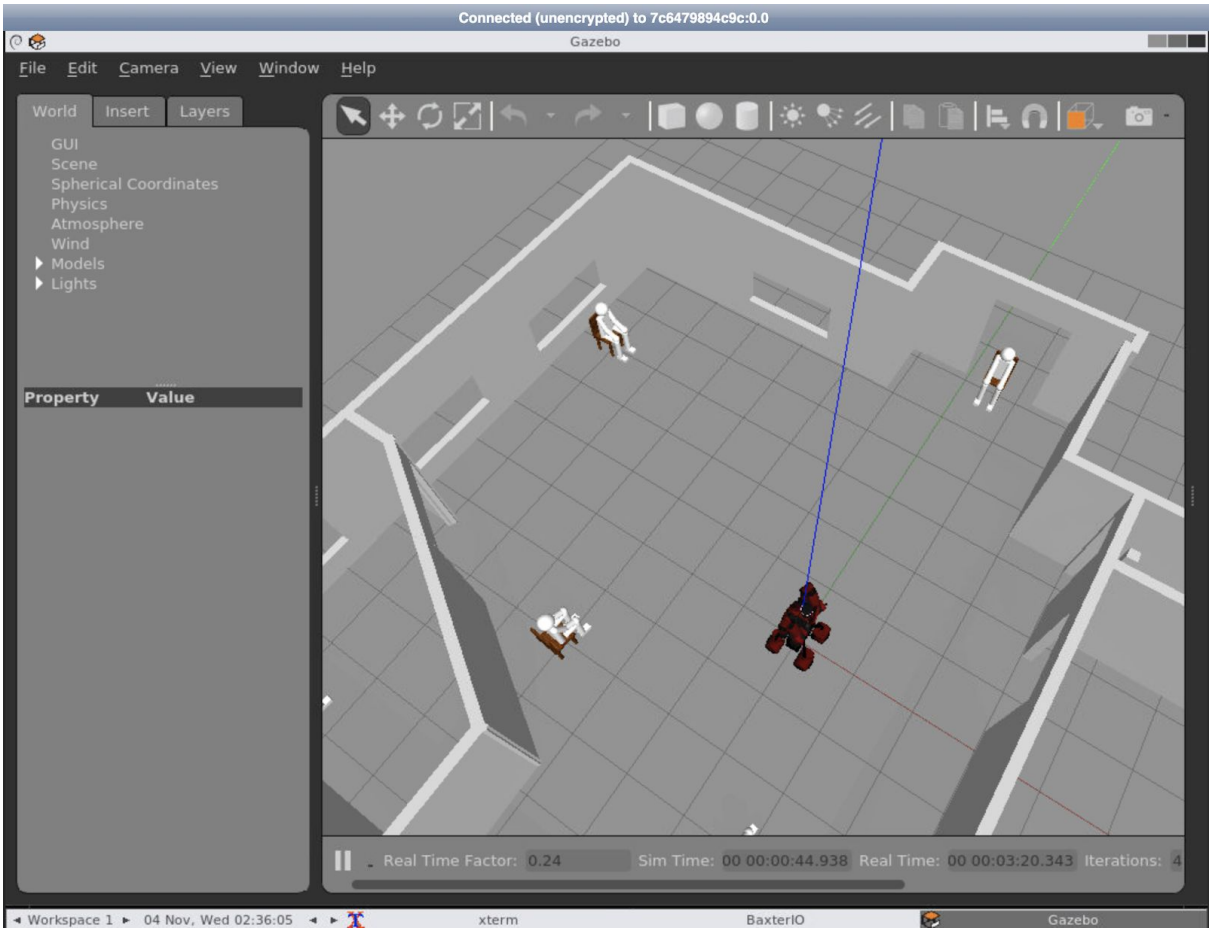

Figure 3: The Gazebo Client

Step 5 - Spawning Blue's Model and sensors

While still connected to vxlab-rosie docker container, run the following command:
`./blue-minimal`
This command will spawn Blue's model with the required sensors and the Hand Sanitizer bottle, which will now be visible in the Gazebo window. Blue needs to be spawned through this method, as the sensors need to be directly connected to the simulation to receive data. Later, we will be rerouting the topics published from the sensors to the vxlab-blue container where required.

Step 6 - Starting Blue's core processes

To start Blue's core processes, such as localisation and navigation, connect to the vxlab-blue container as per step 3, with the following command:
`docker exec -it vxlab-blue bash`
Once connected, change to the main directory housing the scripts with:
`cd ~/mir100/blue`
Finally, run the script: `./blue_start`

This process and terminal will need to be kept open. This script starts up Blue's core processes. First, it relays all topics from the Master ROS Node in the vxlab-rosie container, to access sensor data from the model inside the simulation. Then Blue's main processes are started - localisation, navigation etc. Blue's control panel for manual driving will appear on the Gazebo client.

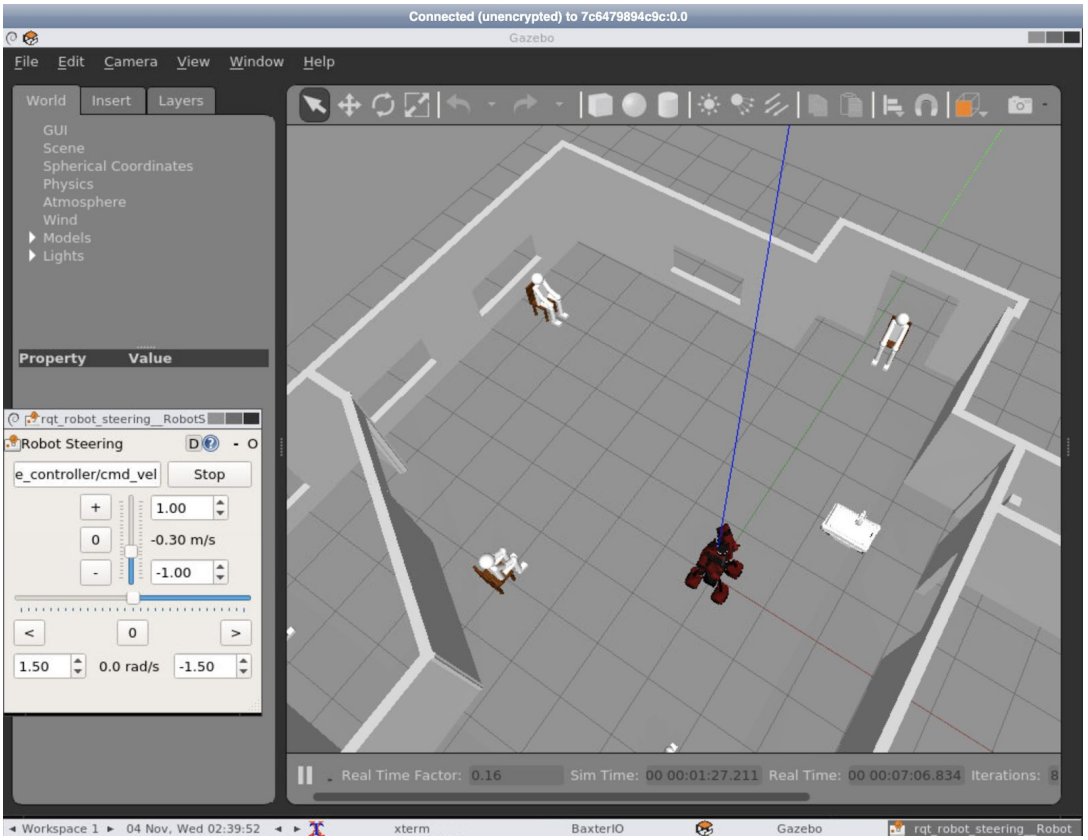

Figure 4: Gazebo and Blue's movement controller

The control panel can be used to drive and turn Blue. Blue is capable of driving forward and backwards quite fast, and can turn on the spot by rotating left or right without forward/reverse movement. Whilst Blue's navigation is very reliable, Blue will sometimes get "stuck" making small movements in tight spaces (eg. doorways), as part of collision avoidance. The system will always give Blue more room than is required as a safety precaution. Manual control can be used to maneuver Blue forcefully when he is in these situations if you do not wish to wait for Blue to navigate by itself.

Step 7 - Start RViz for Blue

To start RViz for blue, once again we must connect to the vxlab-blue container via a separate terminal, as per step 3. Once connected, Run the following commands:

```
`cd ~/mir100/blue`
`./blue_rviz`
```

This process and terminal will need to be kept open. This will open the RViz (Robot Visualization) tool in the noVNC window. This window provides a graphical representation of what the robot's sensors see, and where the robot thinks it is due to its calculated position based on the odomotory sensors. This may differentiate from the actual positions in the Gazebo sim, which acts as the source of truth.

RViz will allow you to see exactly what Blue is seeing through the visualization of the 3D point cloud. All sensor data can be visualized, such as the laser scanners and point cloud sensors. In RViz, the model of Blue with all the wheels and sensors is actually where Blue is identifying its own position in the world, not its actual position. If the position shown in RViz matches the position of Blue in Gazebo, Blue's pose and navigation systems are functioning correctly.

RViz is an excellent tool for debugging the data that Blue is processing, and identifying problems with navigation and other systems. Specifically for this project, it's the perfect solution to see the 3D point cloud showing what Blue is seeing in regards to detecting humans.

Now that everything is initialised, the delivery cycle and target detection can be run.

Step 8 - Start Target Detector

Connect to the vxlab-blue container as per step 3. Run the following commands:

```
`source /ws/devel/setup.bash`
`roslaunch target_object_detector target_object_detector.launch`
```

This process and terminal will need to be kept open. This has started Blue's target detection system, taking the point cloud data and processing it against the trained model to recognize point cloud clusters as the human models in the simulation. You will be able to see in the output how many possible targets are detected. The targets will need to be detected at least twice at the same location before Blue will recognize they are an actual human, to ensure some accuracy in results.

Step 9 - Start Blue's delivery cycle

Connect to the vxlab-rosie container. Run the following commands:

```
`cd scripts`
`./blue_delivery.py`
```

This process and terminal will also need to be kept open. This script runs Blue's standard navigation cycle - moving between the rooms of the VXLab, searching for humans to visit. When a human is detected and confirmed, Blue will move to the human to simulate the delivery of Hand Sanitizer. When visited, Blue will remember the position of a visited human and not revisit the same human until a defined amount of time has passed.

At this stage, you will have two key terminal windows open, and three key windows in noVNC open.

Terminal Windows

Target Detection

This process can be used for viewing how many target candidates have been spotted by Blue during the delivery cycle. There will be intermittent outputs with a count of how many candidates have been detected, as per Figure 5.

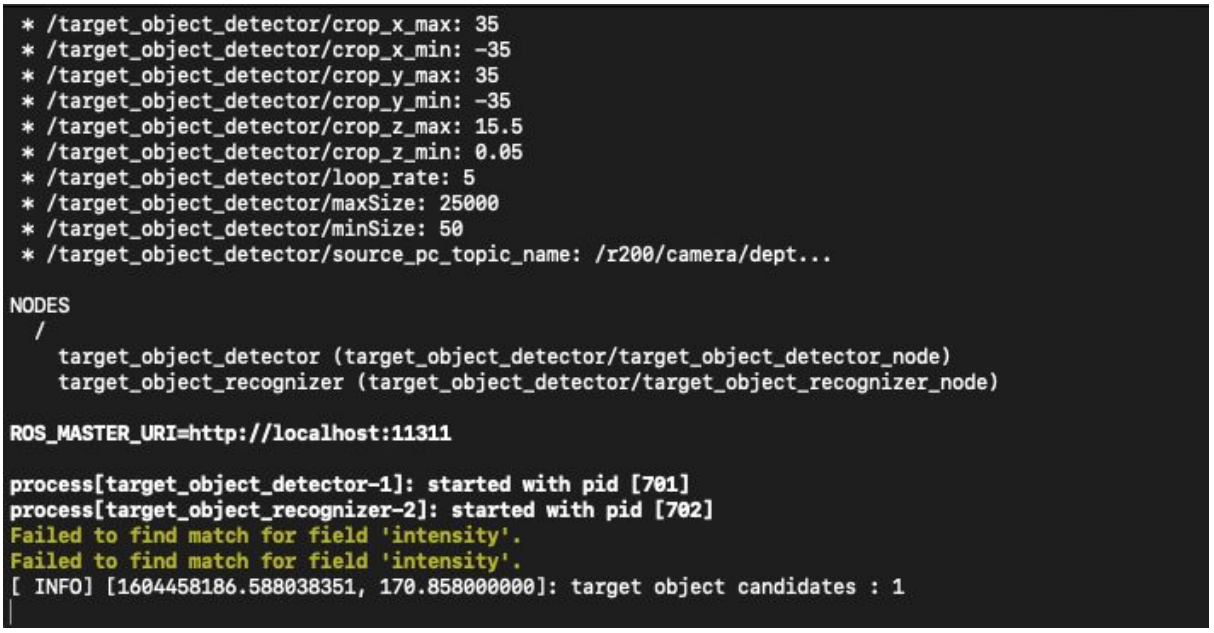

Figure 5: Example output from the Target Detector

Blue Delivery

This process is acting as the main "brain" of Blue in our implementation. While the blue delivery script is running, the terminal outputs where its current target goal is. This will be either one of the four predefined static targets, or if a human is detected, it will output the target goal as "Human".

When a human is detected and added to the dynamic goal collection, it is logged. Successful and unsuccessful goal executions are also logged for debugging purposes.

```
root@18f9eb360c98:~/rosie/scripts# ./blue_delivery.py
[INFO] [1604458423.633751, 0.000000]: Waiting for server...
[INFO] [1604458423.890492, 204.190000]: Target goal: Door
[INFO] [1604458423.892901, 204.190000]: Waiting for response...
[INFO] [1604458513.747688, 217.392000]: Goal execution done
[INFO] [1604458513.754352, 217.394000]: Waiting for server...
[INFO] [1604458513.755929, 217.394000]: Target goal: Window
[INFO] [1604458513.757813, 217.395000]: Waiting for response...
[INFO] [1604458597.981693, 229.499000]: Goal execution unsuccessful!
[INFO] [1604458597.986382, 229.500000]: Waiting for server...
[INFO] [1604458598.069615, 229.511000]: Target goal: Little Room
[INFO] [1604458598.071808, 229.512000]: Waiting for response...
 0
[INFO] [1604458617.243032, 232.258000]: Goal added to dynamic collection.
[INFO] [1604458712.792807, 246.118000]: Goal execution done
[INFO] [1604458712.797874, 246.118000]: Waiting for server...
[INFO] [1604458712.799414, 246.118000]: Target goal: Human
[INFO] [1604458712.801056, 246.118000]: Waiting for response...
```

Figure 6: Example output from Blue Delivery System

noVNC Windows

Blue manual control

Blue can be controlled with the panel as seen in the screenshot below.

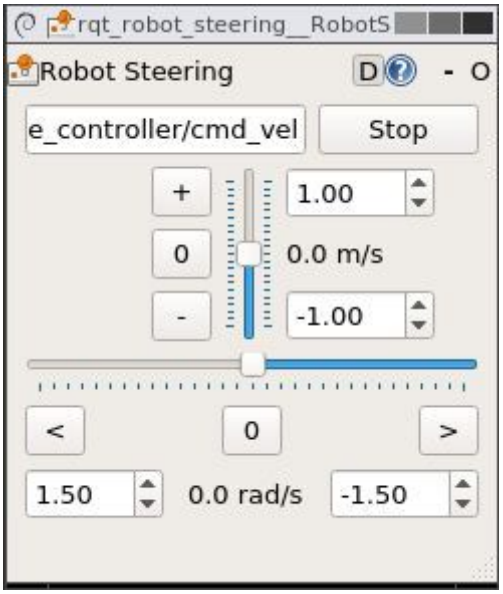

Figure 7: Blue's movement controller

The plus and minus buttons can be used for forward/reverse movement. The left and right arrow keys control rotation/steering. The "0" button in between each set of controls resets the current setting back to zero for each control set.

Gazebo

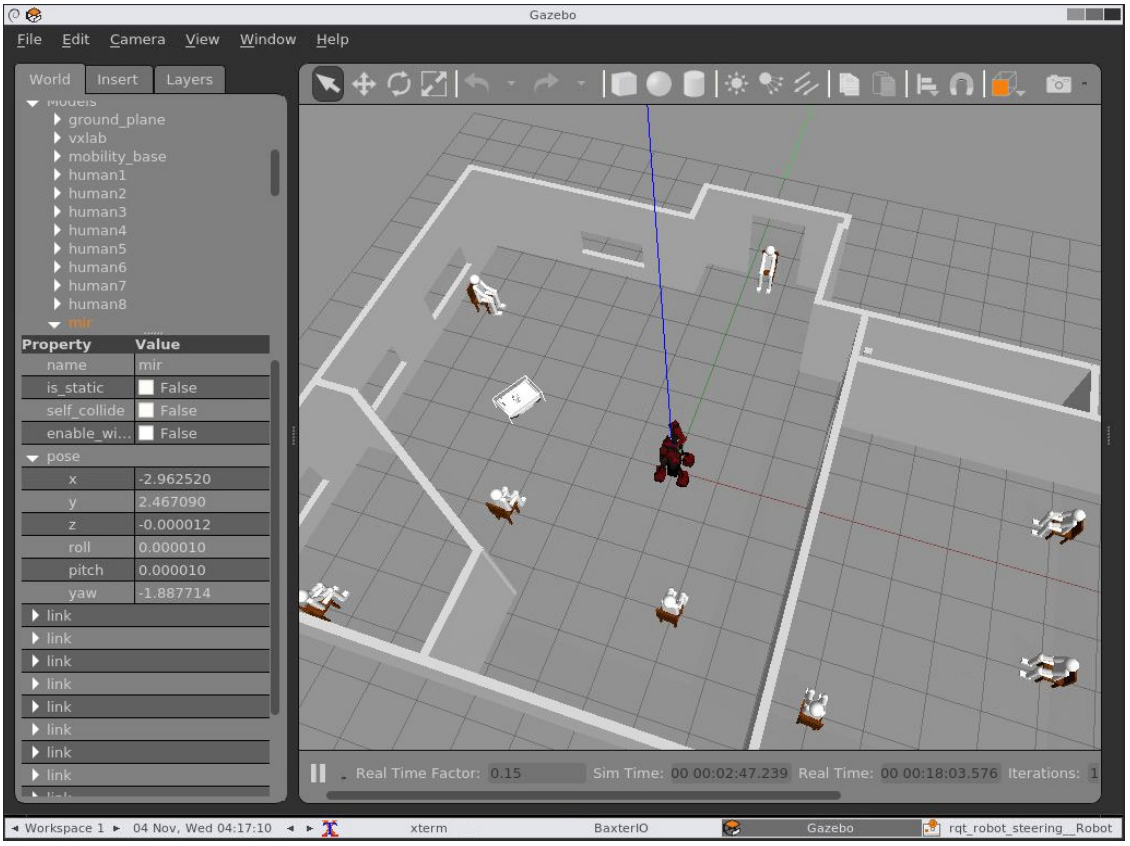

Figure 8: Gazebo with expanded sidebar

The attached figure shows the Gazebo client. The main view window can be controlled with the mouse to move and rotate. Models inside this view can be clicked to be selected. The left sidebar shows a list of all models currently inside the simulation. Once selected, a detailed view is shown below, showing all information about the model. The pose subsection, which is shown open, can be used to reposition the model by changing the x/y/z value.

RViz

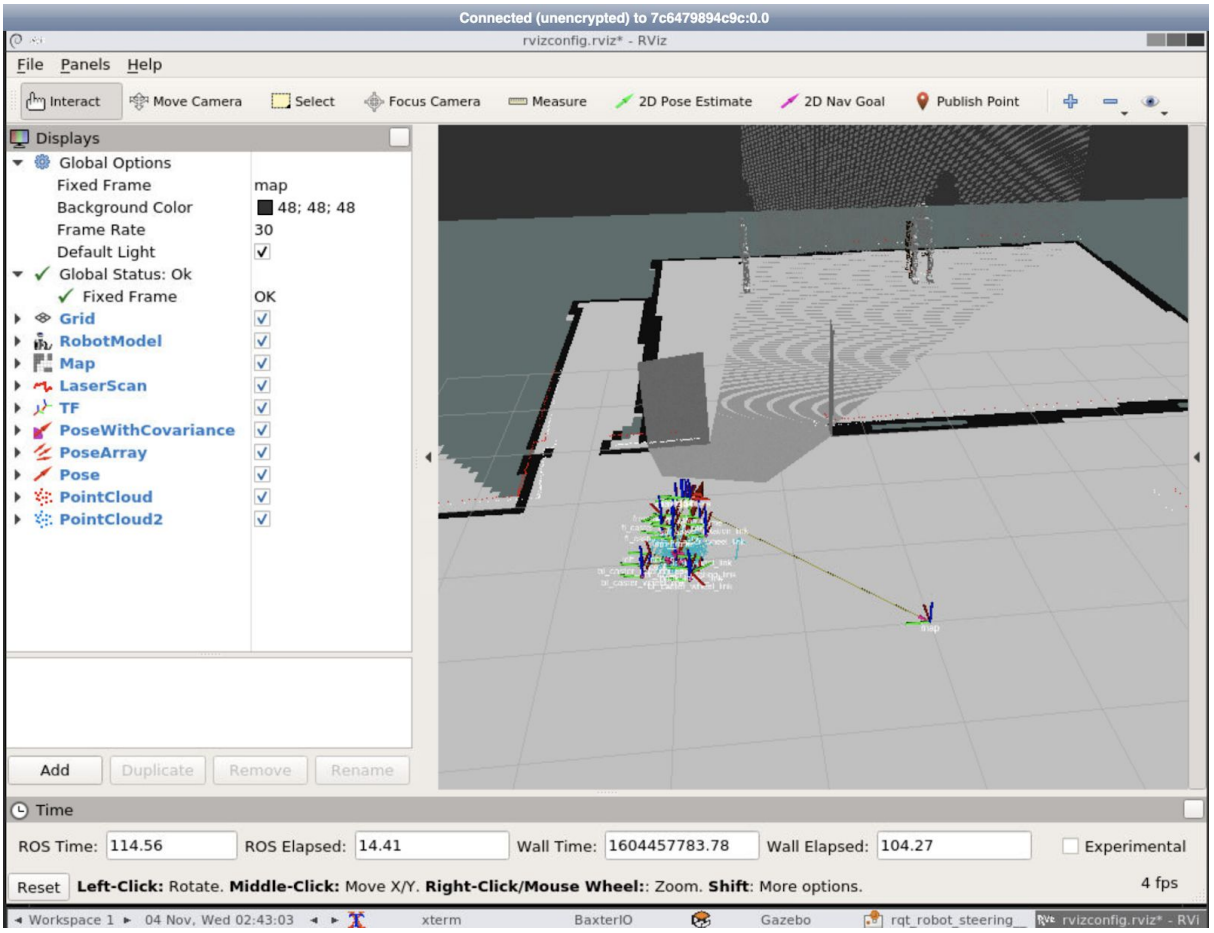

Figure 9: RViz for Blue

This figure shows RViz, the Robot Visualization tool. The sidebar shows a list of all displayed topics, including the map and other sensor data. Each topic can be ticked to be shown or hidden. More topics can be added to the visualization tool with the add button, if other sensor input is required for display. The main topic we're looking at for our project is the "PointCloud2", which is the output of the depth camera showing the 3D point cloud. This is the data source used for the target detection system, and is a good visual to see what Blue is seeing when a human is detected.

Testing

Given the project does not fit a standard software development model, it was emphasised early on in the project that the project testing approach would be no exception. Being in a simulated environment, the project as a whole is a test of sorts, where robot operations can be performed in a physically safe environment before being implemented in the physical world.

Testing approaches that were implemented were static code analysis, manual black-box testing and user acceptance testing was integrated as a part of our definition of done upon completion of a story. Non-functional requirements were also defined, such as usability, extensibility, maintainability and reliability.

Static Code Analysis

Static code analysis tools were used on file formats such as cpp, python, yaml, bash and xml to detect bugs in syntax, indentation etc. During the early stages of the project we were experiencing issues with robot navigation when it was close to walls and corners, indicating that there may be an issue with the costpath planner. The static code analysis of the yaml files that define the parameters of the navigation planner detected issues with the indentation which was causing these navigation anomalies.

Black-box Testing

The main focus of our project when it came to testing was on manual black box testing. As the project was run in a simulated physical environment, black box testing focused on implementing a change and observing the resulting effect via the graphical representation in Gazebo and / or RViz, to ensure that the alteration had the desired effect. Logging of the ROS topics and ROS node states was also implemented to confirm target coordinates and orientations were met, as well as to validate results of features such as human detection that did not affect the visual display in Gazebo.

An example of a black-box test was when importing the new model for the hand sanitizer bottle. The initial model was added to the master docker container and spawned into the simulation environment. The first attempt saw the hand sanitizer bottle instantly fall through the floor, and hence the test failed. Upon rectifying this by altering the collision settings, the test was rerun, and the hand sanitizer bottle successfully spawned above the floor, but instantly tipped over and was rolling around on its top. This clearly indicated to the developer that the bottles centre of gravity needed to be adjusted. Once this was resolved, the hand sanitizer bottle spawned correctly on top of Blue and sat as would be expected in a physical environment.

Once a developer had taken a story through rigorous manual testing, the feature was demonstrated to the rest of the development team, who performed a review of the work and approved the story ready to present.

User Acceptance Testing

When development of a story had been completed and was ready to present, we presented it to the client at our weekly showcase meeting. This demonstration was used as a form of user acceptance testing which, due to the inability to run automated tests in such a project, gave us an avenue to assert that the provided product adequately met the functionality that was defined and agreed upon by the development team and client in the story card description.

Non-functional Requirements

As the project is an open source project that can be built upon or referenced for future VxLab projects, the non-functional requirements were a key component of the project's success. The non-functional requirements highlighted by the team as key to project success were usability, extensibility, maintainability and reliability.

To get the project up and running in the simulation environment involves a sequence of seven or so key steps (excluding the docker build etc). These steps have been structured to minimise the number of steps required, through the use of bash scripts that initialise common functions e.g. ./blue-start, whilst still maintaining a certain degree of modularity. This approach has improved the usability of the base project we initially started with, even though we have built upon it with extra plugins, and extensions.

As the project may be built upon in future VxLab projects, extensibility was a key requirement highlighted by the client. Certain features were implemented throughout development with the vision of potential future expansion. One such feature is human detection, which provides many opportunities for extension to detect any object the system is trained to detect. Not only could this system be extended upon, but by following our development strategy, future teams could use this as a guide to add a similar vision system to the robot, such as facial recognition.

Maintainability and reliability have been a key focus of ours throughout development. We have aimed to keep documentation current and to implement descriptions and comments in code where required. We also developed with the premise that our code should be documentation in itself, and that it was easily understandable for further development. Logging was implemented throughout the blue delivery script so that any defects were easily detectable and traceable.

Feature	Test Case	Expected Result	Actual Result
Model: Hand Sanitizer bottle	Check that hand sanitizer bottle spawns and is visible when the `blue-minimal` script is run.	Hand sanitizer model should spawn on top of Blue.	Hand sanitizer model spawns on top of Blue.
Model: Humans	Check that 4 human models are visible when the Gazebo client is started.	4 human models should be visible around the VXLab simulation.	4 human models are visible around the VXLab simulation.
Actor: Walking Human	Check that humans walk around VXLab simulation when spawned using `actorspawn` script.	The human walks around the lab randomly, avoiding obstacles.	Human walks around the lab randomly, however does not avoid all obstacles and walks through some walls.
Navigation of Blue and Rosie through ROS Navigation Goal system	Check that bash scripts `<robot>_<location>_navgoal` move the correct robot to the specified location.	The given robot will move to the specified location, i.e. Blue to Window	The given robot moves to the specified location, i.e. Blue to Window
Blue Delivery: static move goal	Run `blue_delivery` script with no humans spawned.	Blue moves between 4 static move goals searching for humans	Blue moves between 4 static move goals searching for humans
Blue Delivery: dynamic goal detection	Run `blue_delivery` script with humans spawned.	Blue moves between 4 static move goals searching for humans - when human is detected it is added to dynamic goals collection	Blue moves between 4 static move goals searching for humans - when human is detected it is added to dynamic goals collection
Blue Delivery: dynamic move goal	Run `blue_delivery` script with humans spawned and humans detected in dynamic move goal collection.	Blue moves to the detected human instead of a static goal	Blue moves to the detected human goal instead of a static goal - sometimes unreliable due to occasional detected goal inaccuracy
Blue Delivery: Ignores visited humans	Run `blue_delivery` script with humans spawned and humans detected in dynamic move goal collection	Blue does not move to a detected human after they have already been visited	**Testing data insufficient to reach conclusive result
Human Model Detection: Simple Human Detection	Blue set to be in front of a human with target detection enabled	Blue detects a human, and detection is output with similar x,y coordinates as human position on map	Works unreliably, humans are detected but sometimes after an extended period of time (2-5 minutes)
Human Model Detection: Detection on Delivery	Target Detection enabled while running `blue_delivery` script	Blue detects nearby humans within vision of sensor	Very unreliable detection occurs, data insufficient for accurate statistical results
Human Model Detection: Detector ignores Rosie	Target Detection enabled while running `blue_delivery` script	Blue passes with Rosie in clear vision of sensor, but target detector does not detect Rosie	Blue passes with Rosie in clear vision of sensor, but target detector does not detect Rosie

Technical Solution/System Design

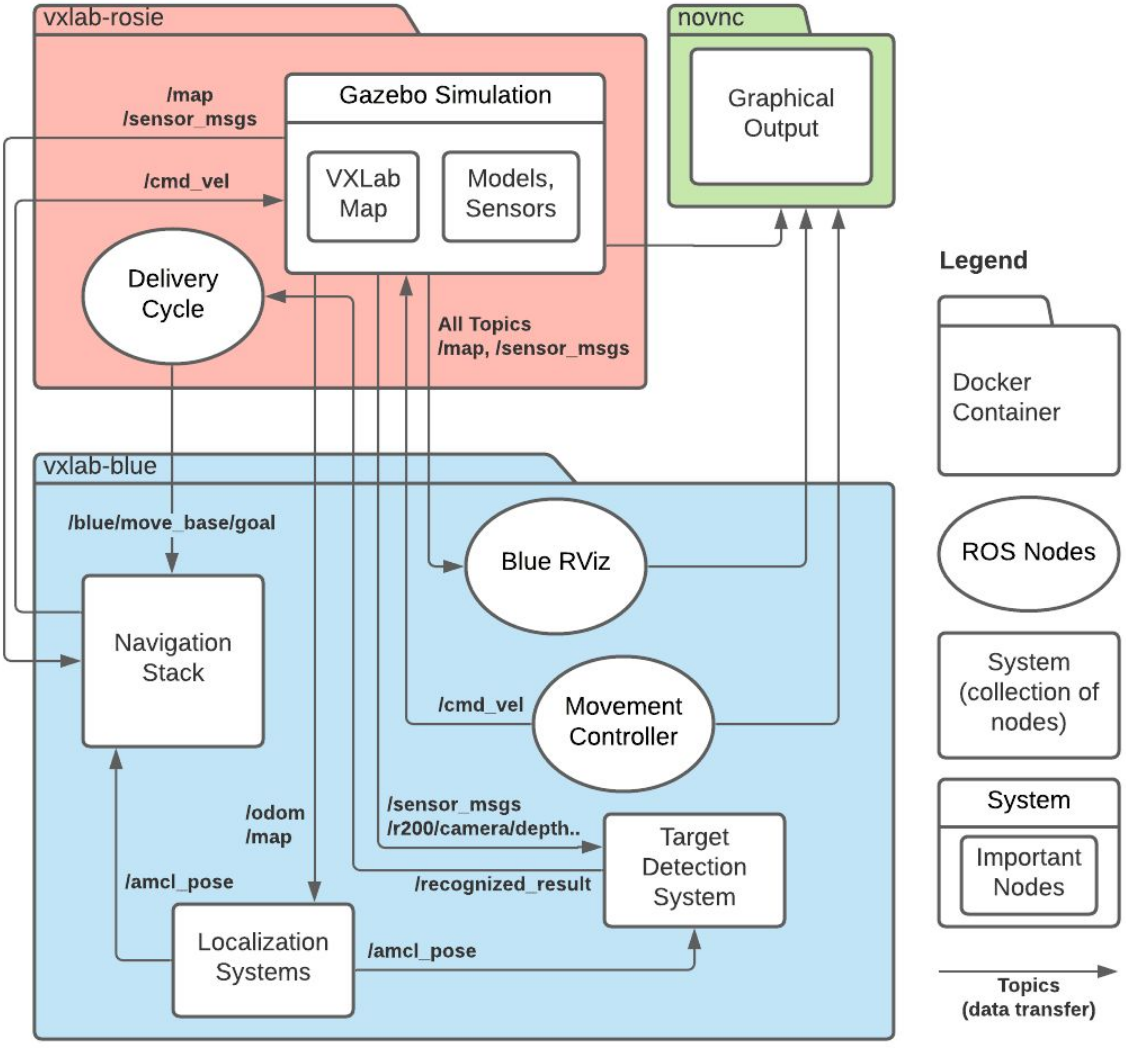

Figure 10: System Runtime Architecture

As per Figure 10, there are three containers which make up the runtime environment for the robot simulation. These containers are segregated to contain collections of similar nodes so that the project isn't a single monolithic container. The nodes in the system operate in an event driven architecture in which they subscribe to, and publish to Ros Topics. These topics transfer data and events between the nodes, even if the nodes reside in separate containers.

The vxlab-rosie container is the master container. This container acts as the Ros master, whilst also encompassing the collection of nodes and subsystems required for the gazebo simulation. The VXLab map, and all simulation models are spawned via this container. The blue delivery cycle node is also initialised in this container when invoked during runtime. The

design decision to initialise the node in this container was due to all other navigation scripts being located in this container.

The vxlab-blue container hosts nodes specifically related to blue's control, navigation and features, such as the target detection system. The target detection system publishes detected objects on the /recognized_result topic, which the delivery cycle subscribes to. The delivery cycle in turn publishes move goals to the /blue/move_base/goal topic, which Blue's navigation stack is a subscriber of. This is a simple example of how the separate containers communicate via topics, and how the event driven architecture loosens coupling between nodes.

The novnc container is dedicated to displaying the graphical user interface. This displays both gazebo and rviz, as well as the robot movement controllers. The segregation of the GUI is vital as GUI's are notorious for slowing processes and causing them to hang. Due to the criticality of near real time robot control and sensor feedback, separation of the GUI processes is a must.

Target Detection - Detecting Human Models with Blue

One of the key features of our project is the Target Detection system we have implemented onto Blue. The system uses a modified version of an existing target detection system implemented for a past ROS robot (https://github.com/CIR-KIT/human_detector). Using the Realsense R200 camera, a 3D point cloud is passed through ROS topics to the Target Object detector. The detector recognises target candidates from point cloud clusters that resemble the trained model. Once the target is detected and confirmed, then the target is passed to the delivery navigation system as detailed previously.

Each section of the Target Detection depicted below is its own C++ program compiled into either a ROS Node or Plugin.

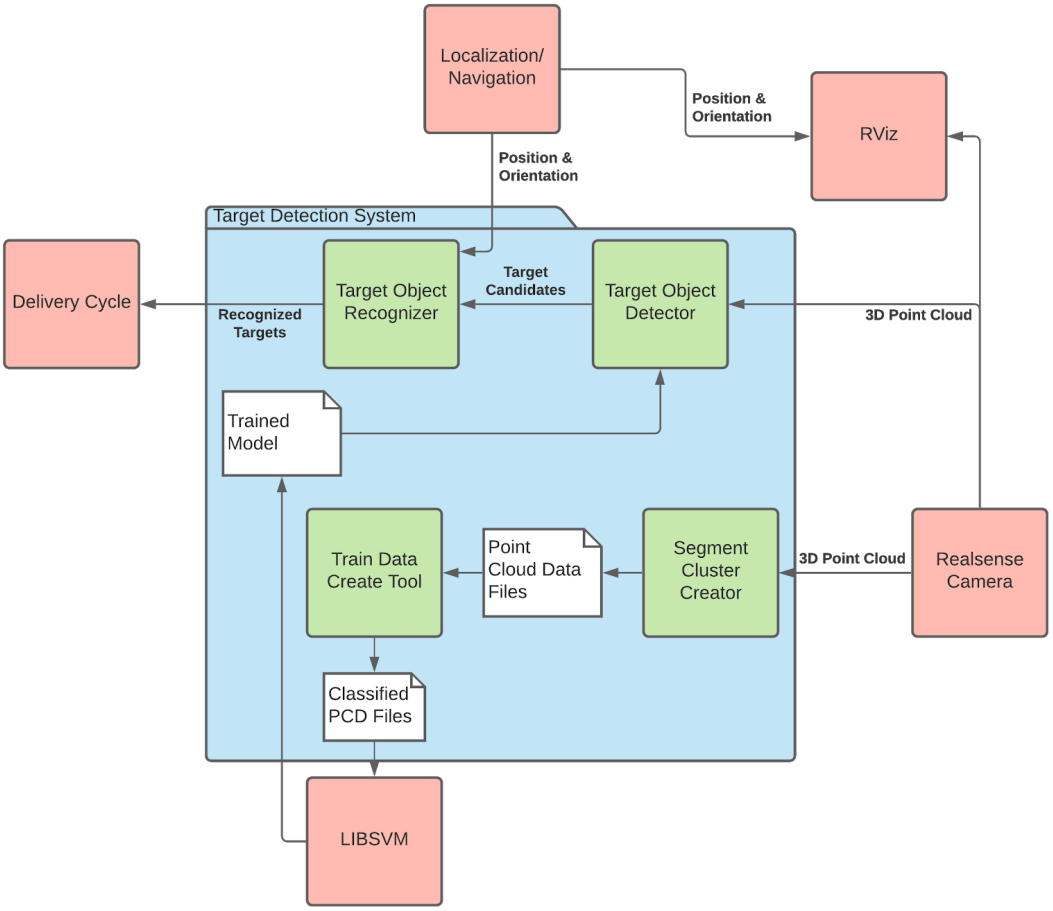

Figure 11: Target Detection Architecture

Target Object Detector

The main process of the Target Detection. The target detection takes clusters from the subscribed 3D Point Cloud topic, which is compared to the trained model. Each cluster that is detected similar to the trained targets is sent as a target candidate to the Target Object Recognizer.

Target Object Recognizer

The Target Object Recognizer is responsible for ensuring the accuracy of detected targets. Where the Detector is comparing the clusters spotted, the Recognizer works as a verification process. Object candidates are processed based on similarity of new detections of the same targets, to validate the accuracy of the detection. Each detection is compared based on the coordinates of the detected cluster relative to previous detections, using the map as a frame of reference (eg. Similar X/Y coordinates relative to map on multiple detections). Recognized Objects are passed to the Delivery Cycle system for use in navigation for the simulated delivery process, as detailed further under Blue Delivery Cycle.

Segment Cluster Creator

This is used for data generation when training. Clusters are generated from the 3D Point Cloud passed from the Realsense Sensor. Each cluster is stored as a .pcd file, which can then be classified for training the detection model.

Train Data Create Tool

This tool is used to convert generated Point Cloud Data files into usable information for the training model. When running this tool, the tool will process all files in the current directory, displaying each .pcd (point cloud data) file in a 3D Point Cloud Viewer. Each cluster is either approved or rejected by users (Y or N key). The model is trained to detect approved clusters and ignore rejected clusters. This approval process is then output to a .CSV for processing into the trained detection model.

Libsvm

A popular machine learning library. Used to process the resulting .CSV files from classification into the trained detection model. Using the supplied scripts, the data is validated and then processed into the trained detection model used by the Target Object Detector.

Realsense Camera

We are using an imported Realsense R200 model and sensor as a 3D Point Cloud generator. The sensor contains multiple cameras: Colour, IR and Depth cameras. Using the depth cameras, the generated depth data is processed into a Point Cloud to be passed to the relevant nodes where required.

Localization & Orientation

This object represents the robot's ability to track where it is using the navigation system and sensor information. This information is used to track the position of detected objects relative to the robot, and the map itself, to help recognize successfully detected objects.

RViz Integration

RViz is an important tool in understanding how to use the Target Detection. The tool allows visualization of sensor data, basically enabling you to see exactly what the robot sees. This tool is useful for debugging sensors and other information by comparing what you see in RViz vs. what you expect to see.

Development Guide

First Steps

The first step for developing a ROS based simulation project is setting up your environment. You will require Gazebo, ROS, access to robot models and all required dependencies. As part of the initial handover process of our project, we were supplied with a base docker environment with functional robots. We already had access to Rosie (a Baxter robot on a Move Base) and Blue (a MIR100 robot) as working robots inside the simulation, with navigation already functional. We were also provided with working examples of scripts and code such as navigation and demos of scripted behaviours so we could traverse the learning curve as quickly as possible.

That being said, there is a steep learning curve to overcome before starting development of a similar project. You will need to have, or acquire a solid understanding off:

- Docker
    - How the containers work to a basic level
    - How to run commands
    - How to edit builds to include required dependencies or install new software
- Scripting
    - How to use the supplied ROS Python libraries to interact with the robots to write your own functions or demos
    - Using bash scripting to automate building/launching/running certain tools
- Gazebo
    - What Gazebo is and how it works
    - How to interact with the Gazebo interface
    - How to launch new worlds or models
    - Poses, Orientations and Quaternions
- ROS (Robot Operating System)
    - How ROS Topics work for communicating information to and from the robot
    - How to interact with the robot through the use of topics
    - How ROS nodes are initialised, publish to topics and subscribe to topics
- RViz (Robot Visualization Tool)
    - The purpose of RViz, how it is different from Gazebo
    - How to use RViz to debug what is happening in the simulation
    - How to use topics for visualization of sensor feedback
- Robot and other Models
    - What the file formats are (XML, SDF, URDF)
    - How to import them, spawn them into the sim, launch them with world files
    - Some level of physics to understand how models will interact with the environment

The majority of your first few weeks working on these projects will be spent learning the prerequisite knowledge, depending on what your development will be focused on.

Features of our development

Importing Models

Hand Sanitizer

As part of the goal to produce a simulation of robots assisting with Covid Safe measures, we imported a custom hand sanitizer bottle model into the simulation to sit on Blue for delivery. Whilst we could have used models from Gazebo's existing database as a placeholder, we wanted a more accurate visual representation for the sake of viewers of live demonstrations and videos.

Human models

Similar to the hand sanitizer, human models are also an intuitive visual representation of our goal for the simulation.

We have placed human models spread in multiple locations around the VXLab as we wanted to demonstrate the capability of the system, by having examples of completed goals in a variety of situations. These placements also double as a representation of possible reality when people are required to socially distance inside the VXLab when returning to campus.

Sensors on Blue

The existing original MIR100 Gazebo model provided in the original handover did not have the sensors required for target detection. The real life MIR100 has two Realsense cameras, in addition to all other sensors. In order to closely imitate the real life robot, we decided to import and attach a Realsense R200 camera to Blue as the depth sensor required.

The camera was retrieved from an existing github repository (https://github.com/SyrianSpock/realsense_gazebo_plugin). Only one instance of the camera is attached, as two separate cameras are not required for the purpose of target detection. The camera is mounted on the front of Blue's model, and produces a 3D Point Cloud generated from the depth camera. This point cloud is the data source used and processed by the target detection system.

Target Detection

While the target detection system is adapted from an existing project, a large amount of development time was spent repurposing the system for our project. As there was limited documentation, most time was spent integrating the system into our project.

A key factor required for human detection is the relay of topics across the docker containers. As the main simulation is running inside the vxlab-rosie container, all data acquisition takes place in that container. The camera/sensor data must be launched inside the existing Gazebo simulation to be able to read the simulation data. To be able to access this data in

other containers, we must relay the topics across. This is handled with a bash script called “relay”. Topics to be relayed (either publisher or subscriber topics) are added to this script to be accessed inside the vxlab-blue container.

The Target Object Detector (target_object_detector.cpp) node requires a 3D point cloud topic as input. The original implementation uses a Hokuyo sensor (/hokuyo3d/hokuyo_cloud2). When implementing the sensor, we had to replace this topic in the Target Object Detector to the new Intel Realsense R200 topic (/r200/camera/depth_registered/points). The system can use any 3D Point Cloud data topic and is compatible with any such sensor.

Modifying the Target Object Recognizer (target_object_recognizer.cpp) was required to increase sensitivity and reduce accuracy of detections. The system was originally set with much stricter criteria for detected objects, such as:

- Objects must be detected 6 or more times before considered as successfully detected
- Objects must be detected within 1m of first detection to be considered for successful detection, otherwise removed
- If the detected object is further than 10m away from the detector (Blue), the object is removed

These criteria can be modified to allow for a wider margin of error but increased detections, or a smaller margin or error with more accurate detection, albeit fewer results.

The most important part of the target detection for development is the training. By running the Segment Cluster Creator, the detection system is run in a sort of blank slate mode, generating point cloud data files intermittently of any detected “object”. These data files are used with the machine learning LIBSVM library to generate the model used for target detection in the Target Object Detector.

As we were aiming to detect humans, we ran “training” sessions with Blue, where he was situated in the lab with human models in varying positions and orientations around Blue. Blue was then set to rotate, generating the point cloud data as the sensor picked up any objects. These are then parsed manually using the Train Data Create Tool to create the machine learning model to be used for target detection.

Blue Delivery Cycle

The blue delivery cycle is controlled via a python script `blue_delivery.py` in which a rospy client node is initialised. The client subscribes to the human detection ROS topic and controls Blues navigation through the use of move goal actions.

When a human is detected, and it’s coordinates published by the human detector, a callback is triggered in the delivery script by the ROS topic subscription. The detected coordinates are passed to the callback as a detailed object, which is filtered and converted into a much simpler move goal. Each detected move goal is checked against a collection of existing

dynamic move goals (if previously detected) and if the move goal already exists or has been visited recently, then it is ignored, else it is added to the collection.

The delivery script will assign priority to detected humans over the static move goals - selection of the next move goal is determined based on the size of the dynamic move goal collection. When all dynamic move goals have been actioned, the delivery cycle reverts to its systematic exploration of the VxLab via the static move goals.

Guide for future development

ROS

To expand upon the current project, a basic understanding over the key aspects of ROS is required. ROS (Robot Operating System) is a robotics middleware platform. A few of the key services essential to this project that ROS provides are services such as hardware abstraction for processes, message-passing between processes through the use of the publish–subscribe pattern, and package management.

An individual process is called a node, uniquely named and registered with the ROS master process. Nodes are at the core of ROS programming, as nodes take actions based on information sent to and received from other nodes. The buses in which these messages are communicated over are called topics. The content of messages can be anything from sensor data, to motor control commands, state information, or anything else a node wishes to publish. A package in ROS refers to a compiled Catkin workspace, which generally consists of a group of nodes and/or plugins with common functionality and dependencies.
Acquiring a good understanding of nodes, topics and packages will enable a developer to interact with the system through a bash command line interface.

Using ROS tools, developers are able to list active nodes, and details about each node such as which topics they subscribe and publish to. Developers can also list available topics, echo topic details, such as data structures, data types and data values. These are invaluable debugging tools when more granular detail is required.

ROS Topics

Understanding ROS topics is integral to working with ROS. As the primary means of communication to and from the ROS node, you will need to learn how a standard Pub/Sub topic works, and further how to interact with them in the context of ROS. By using these topics, you will be able to access much more information about the robot and its behaviour, including navigation and sensor data. These topics are extremely important, especially when used alongside RViz.

RViz

RViz (Robot Visualization) is a standard ROS tool. Used as debugging software, RViz gives you a visual representation of the robot's systems including navigation and orientation. RViz will allow you to select which topics to visualize in the interface, usually in the form of sensor

data. When you are able to “see” what the robot sees, through the visualization of the laser scanners or other sensors, you will be able to quickly identify errors in navigation or positioning. RViz was particularly useful for our project, when working with the 3D point cloud sensor, as we were able to verify when Blue was looking at a human.

Shell / Bash Scripting & CLI

A solid understanding of bash is essential in any software development project, but when it comes to robotics simulation projects, it is critical. Scripts enable the development team to automate multiple steps of the project, such as launching the simulation, spawning models, initialising nodes and their core services. Without such scripts, the start up procedure would be in excess of 15-20 steps, required to be run in multiple shell environments, all in a specific order.

Scripting and CLI knowledge ties back into previously mentioned interactions with ROS knowledge regarding nodes and topics, and how to interact with them through the command line. This can be further built upon when attempting to implement a simple feature. Bash scripts are a simple option to automate testing a feature as a proof of concept, instead of manually running each command individually, ensuring consistency and repeatability.

Importing Models

As Gazebo has its own existing model database, a good starting point would be to use it as a way to get familiar with how models are spawned in Gazebo, including coordinates and how to locate the model in the Gazebo interface. This is also a good way to explore the basics of model files, specifically SDF and URDF files which are in XML format. Most of the properties in these files relate to physics and require knowledge of areas not usually taught in software development.

Potential future developments of the models include adjustments to the actor (moving models) with plugins so that they are able to accurately avoid obstacles in the simulation, as this was not explored in depth due to time constraints.

Target Detection

Target Detection is almost completely reusable for future projects. With its current configuration, if you are wanted to detect a target that is roughly the size of a human (Rosie for example), you would easily be able to retrain the system with the required target. Based on our understanding, it could be possible to reconfigure the cluster generation system to detect targets of drastically different sizes if required.

As the target detection system functions with successful human detection, it could be integrated into other projects such as the past project of Rosie answering the VXLab door. With the correct sensor, Rosie could be upgraded with the ability to spot humans outside the VXLab door and open the door dynamically.

Human detection could also be quickly implemented to enable deliveries, such as accepting parcel deliveries from the door to inside the lab, reducing human contact. Blue could be set to roam around the lab or wait near the door, and if detecting a human near the door, drive out to accept the package, then navigate to any human inside the VXLab to deliver the package.

If time allows, we recommend spending more development time on enhancing the target detection system to enable more accurate detection in a smaller time frame. This can be achieved through configuration of the target detection system itself, and more time spent training the system to detect the objects required.

Navigation

The navigation script, `` `blue_delivery.py` `` was built using the rospy Python library. The rospy client API enables developers to interface with ROS topics via a node initialised in the python script. In our script, this node subscribes to the topic in which the target object recognizer publishes to. A ROS package called actionlib, which provides a standardised interface for interfacing with preemptable tasks, was also implemented to send move goals from our python client node to Blue.

Moving to a detected target was the main objective of our navigation system, though the navigation script has been written in a generic way, so that the code pattern implemented could be used as a base reference for further expansion where long running dnamic move goals are required. With some minor changes, the code could easily be reused on Rosie, to dynamically move her around the VxLab for a different purpose, with goals generated via a subscription to a different topic. This would enable the robots to do all sorts of routine tasks, such as deliveries, cleaning etc.

Project Processes and Progression

Project Scope

The current outbreak of COVID-19 has caused huge changes in the way we live our daily lives, with restrictions on human interaction and increased health measures helping to curb the spread of the virus. Many of these health measures are simple routine tasks, such as sanitizing/cleaning and temperature checking. By replacing humans with robots in these simple tasks, it would greatly reduce the number of human-to-human interactions therefore reducing the risk of exposure to COVID-19.

With this in mind, our project aimed to explore the potential use cases of robots in enforcing COVID-19 health measures. Specifically, we brainstormed ideas for a Covid-Safe VXLab using the robots already in the physical lab, Rosie and Blue. Ideas considered and discussed with our client included:

- Checking the temperature of people as they enter the lab.
- Verifying people are wearing masks.
- Enforcing social distancing protocols.
- Supplying hand sanitizer.
- Receiving deliveries.

After extensive discussion with the client, we narrowed down our ideas to focus on the main goal of using Blue to supply hand sanitizer to people in the lab.

Processes

We implemented an agile, incremental approach when developing the project, but due to the steep learning curve of the project, the main goal and expected time to complete evolved as the project progressed. As we continued to learn more about the platform and required technologies, our ability to gauge time and effort required for tasks also improved. This progress is reflected in our documentation, both in meeting minutes and our Trello board. Where initial tasks were often broad and difficult to declare as ‘done’, later tasks were more specific and had a clearer Definition of Done (DoD).

Regular meetings with both the client and technical lead also assisted greatly in establishing tasks with clear DoD and ensuring the project stayed true to the original vision. Weekly showcase meetings were held with the client and technical lead to present progress from the previous sprint and based on the progress made, decide task priority from the sprint backlog, and set new tasks for the following sprint.

At the start of our project, our sprints began on Monday and ended on Friday. However by Sprint 3 we shifted the sprint start and end dates to better reflect the work styles of the group members and more effectively implement recommendations from our client and technical

lead. As team members often did work over the weekend, ending our sprints on Monday ensured that all work done over the sprint was discussed during retrospective meetings and sprint planning meetings were held after weekly client meetings on Tuesday afternoons.

A lot of the project development consisted of trial and error - e.g. how did a particular feature implementation respond in the simulation. To ensure that we did not get bogged down on tasks and end up down a rabbit hole, we implemented time boxing where we felt tasks were requiring too large of a time investment for limited reward. This was a valuable tool in ensuring development progressed week in week out.

Our team also implemented some extreme programming techniques - most notably pair programming. Often, we would have team dial ins with two or all three team members, where we would work together to complete a particular task, or to implement the integration between two features such as human detection and the navigation system. We also used these pair programming sessions to review one another's work to ensure it was completed and ready for showcase to the client.

Progression

Throughout the course of this project, all team members learnt a lot about technologies and processes that were outside of the scope of the typical software development projects that we had been familiarized with over the duration of our studies. This cross-disciplinary learning experience has allowed us to grow in areas we never knew even existed before starting this project.

The base code for this project provided was the culmination of several capstone projects done on Rosie and Blue in previous semesters along with additions by the technical lead. We have built upon this code significantly, particularly as Gazebo has only been in use since last semester due to the onset of COVID-19. An area which has had visible improvements is in the models for Gazebo as they simply were not necessary previously. Last semester, the models used were from the Gazebo database (a coke can) but this semester our project has introduced several custom models into the simulation such as the hand sanitizer bottle and humans.

Importing Models

From the existing project, we were shown the basics of model usage in Gazebo. The previous progress involved spawning blocks, which were being used for interaction with Rosie. While we did not use the marker blocks for this project, it was a useful jumping off point into learning about models. The existing scripts included examples of spawning models with different coordinates and orientations, and deleting models. We also had access to the simple models (the marker blocks) to learn simple model file structures and formats.

Hand Sanitizer Bottle

Importing custom models into the Gazebo simulation became a bigger task than initially expected during initial stages of the project. Whilst Gazebo has an existing

community-supported model database, there were no hand sanitizer (or comparable) models therefore we decided to import our own custom hand sanitizer bottle into Gazebo. This process involved searching for a 3D model online, then downloading the file and making adjustments with SketchUp and Blender to give the model physical properties such as collision. After importing the model file into Gazebo however, we were met with many obstacles. Initial attempts did not succeed, as the model was listed in the Gazebo interface, with a ‘successfully spawned’ message on the CLI after running the spawn command, yet was not visible in the simulation itself. After bringing the issue up with our technical lead, we were directed to the environment variable `GAZEBO_MODEL_PATH` and found that the path needed to be set to our local ‘models’ folder that held the files for the hand sanitizer bottle, as the default path led to Gazebo’s existing database and therefore could not find the custom model that was being referred to.

Once this was achieved, the model was visible when spawned in the simulation, however would promptly fall through the floor and disappear. Whilst researching this error, we discovered that documentation on Gazebo models and SDF files is very limited. This led us to adjust our approach, shifting to trial and error with editing the model sdf file, instead of attempting to rely on solutions found online as they simply did not exist. Through adoption of this trial and error approach, we were able to gradually improve the model, adjusting until reaching the desired behaviour. That is, where the hand sanitizer bottle acts as you would expect it to in real life.

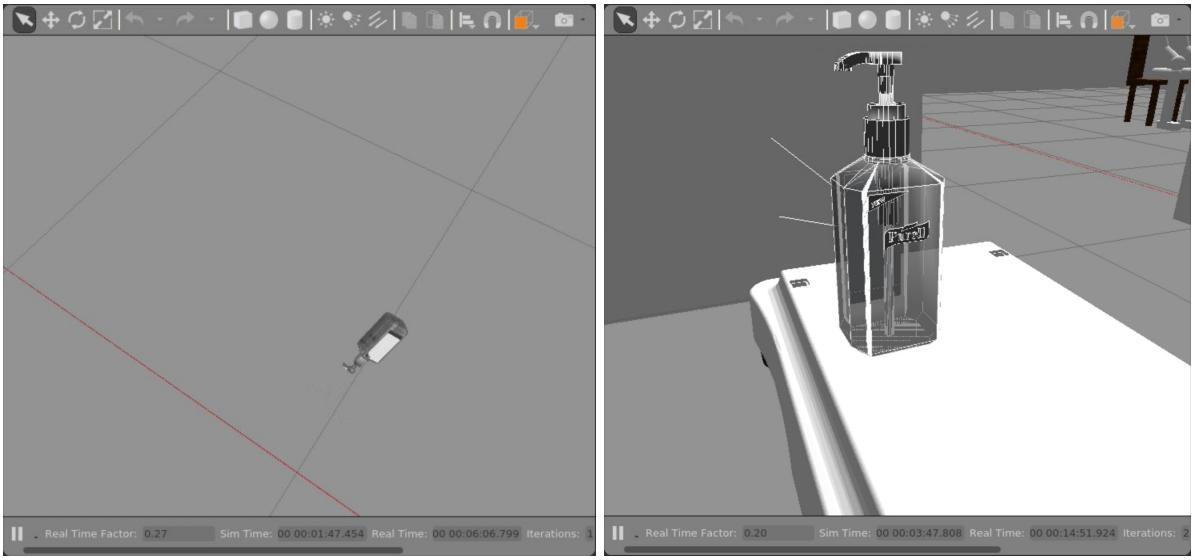

Figure 12: Hand sanitizer bottle progress

The figure above shows the progress of the hand sanitizer bottle model; on the left where it no longer fell through the floor and instead would sit upside down and ‘float’ on the floor. Through using Gazebo’s tool to show centre of mass, we found that the centre of mass was set incorrectly and worked on editing the sdf file, testing how different properties affected the model by adjusting the values or simply deleting the property altogether. Through this approach, we adjusted the centre of mass and added an extra collision point which allowed it to finally stand upright as seen on the right in the above figure.

This trial and error approach worked best for us as whilst there was official documentation explaining model sdf files and their properties, it was simply out of scope for us to completely understand the physics behind the model and how it behaved in the simulation.

Human Models

With the preexisting examples provided to us, initial model spawning was relatively simple for the human models. Initially, we had found some simple human models, and spawned them directly into gazebo. We noticed we were missing some required knowledge when only a single foot was visible, instead of the entire model. After researching and including required properties, the entire human model became visible, but collapsed to the floor in a heap. Finally, after advice from our technical lead, we set the models to be static, and they stood in the simulation without error.

After our demonstration of the initial human models, our client requested an updated model that looked more human. As part of the target detection system, the project had included human models to be used for detection. We were able to load the human models into a separate Gazebo world, and save the model for reuse in our own project. This model also included a sign next to the human, which was sitting on a chair. After closer inspection of the URDF file of the model, the team removed the sign to simplify the model.

At the client's request, we then attempted to introduce human models that could walk around the simulation to be detected by Blue. These walking human models (actors) were introduced to the simulation through Gazebo's existing actor database and tutorials. The tutorials instructed that actors be placed into the 'world' file, so that they would appear when the world itself was created in the simulation. As we did not want the actors to appear everytime the simulation started, we used our knowledge from spawning the custom models to create sdf files for the actors and spawn them on command. We then adjusted a C++ program that was a plugin for the actors to move randomly and avoid objects in the simulation world. This worked to an extent, as the plugin only functioned when the code for the actor was placed in the world file and not its own sdf file. The actors also did not avoid all obstacles in the world and it was not clear what was affecting it. Progress on the actors was halted due to there not being enough time left in the project, as there was also the issue of the actors not being detected by Blue's sensors.

Realsense R200 camera

During the project, we discovered we required a new sensor to generate depth data. As we needed this model to be attached to an existing model, Blue, we encountered a new set of challenges. We did not want to independently spawn the sensor into the simulation as, while possible, would not align with our goal of Blue detecting and navigating to humans. After some research, we had decided to include a Realsense R200 camera that already existed as a ROS plugin.

While we researched how sensors were currently integrated onto Blue, each sensor was inherently built with Blue. As we were attaching an independent ROS plugin, we needed to research how to build and import a ROS plugin before trying to include the plugin in the

existing model files of Blue. We solved the problem by including the plugin into the vxlab-rosie container, to be built with the other required dependencies. The plugin was then integrated into Blue's URDF model file, with a separate macro also imported from the plugin. This macro combines the raw sensor data produced into a more useful data stream. For example, the camera actually has two depth sensors both gathering data. These two data sources are combined into a single ROS topic, a 3D point cloud.

Target Detection

As with most aspects of a ROS project, the first step with any development goal is research. We searched the ROS wiki and internet for previous projects with similar features. Luckily, we quickly came across the Target Detection system that would be used for the remainder of the project.

Initial stages of integration were slow. We had little to no experience with catkin workspaces, a package management tool for ROS. On our first attempt, we first deployed the docker containers, then while running, installed the project manually onto the running container. With a few tests attempting the different commands, we learned the installation had been successful, and integrated the steps taken into the Dockerfile so the system was integrated into the container without needing to be manually installed.

Once installed, we started taking steps to understand how the human detection system was implemented. By analyzing the system, we were able to spoof target detection using the "fake target detection" system that is included as part of the project. We were able to manually write data of coordinates to a text file, and the system would then simulate detecting targets at those coordinates, with output shown on RViz. With this step, we were confident the system would be able to achieve the results desired once functionality was fully implemented.

Upon further examination of the Target Detection system, we learned a 3D point cloud was required as the source of data to be processed. As the Gazebo model of Blue was not equipped with any sensor capable of producing a 3D point cloud, we refocused to find and mount a suitable sensor onto Blue's model before continuing (see Importing Models above).

After we successfully imported the model, we encountered another problem. For a sensor to be functional, it needs to be able to access the main simulation, running on the vxlab-rosie container. However, the target detection is running alongside Blue, on the vxlab-blue container. After speaking with our technical lead, we integrated Topic relaying.

As the main simulation is running on a separate docker container (vxlab-rosie) to where the target detection is processed (vxlab-blue), the topics need to be relayed across the containers. Sensor data can only be generated inside the main simulation (where the base model is spawned), so necessary topics must be relayed across to Blue's container for use in the target detection. This is handled when blue is started with a bash script, bridging the topics across the internal vxlab network implemented as part of the build process.

From our previous research, data is quickly implemented into the target detection system, overwriting the original 3D point cloud topic included with the project, with our new 3D point cloud topic from our integrated Realsense R200. Unfortunately, the target detection system did not work as we had hoped. While the point cloud data was correctly being accessed by the target detection system, we needed to integrate the machine learning system to see results.

With more research into the target detection system, we discovered a training module, which generates data files to then be processed. This involved a three step process:

- Using the Segment Cluster Creator to generate .PCD files based on detected “clusters”, tight grouping of points from the 3D point cloud
- Manually approving or rejecting clusters with the Train Data Create Tool
- Finally, using the processed data with LIBSVM, a common machine learning model, to generate the target detection model

We ran multiple sessions of “training” for Blue. Blue was navigated to the centre of the VXLab, with multiple human models spawned in sight of Blue in various positions and orientations. Blue was then set to turn slowly, and clusters were generated from the Segment Cluster Creator tool. These sessions were run for a total of about 8 hours, each with different positions for humans and Blue, to gather as wide a range of data as possible.

After training, while we did see some success, the detection was not as successful as we had wished. Blue would detect a human, but the humans would never be published as recognized to the expected ROS topic. The team worked together to examine the target detection system, and found our results were being hampered by strict criteria from the Target Recogniser system.

This system is used to ensure accuracy of results, by requiring multiple detections and cleaning old results. As an example, a target candidate needs to be detected in the same location five separate times before a successful detection would be published, and if a target was detected out of range by over 1 metre, the target was removed as a candidate. We reduced the threshold of these checks, and saw a marked increase in detected targets, albeit at the cost of some accuracy.

While we now have working target detection, with successfully recognized targets being published for the delivery system, there are still improvements we could make. We discussed spending extra time on training, but due to the limited remaining time of the project, decided to continue with the results we had and focus on other tasks.

Navigation

The base code we inherited came with a few inherent navigation issues, especially when Rosie was navigating near walls and corners. As mentioned in the testing section, the root cause of the navigation issue was found to be due to indentation in the yaml parameters file. This saw a big improvement in Rosie’s navigation, but unfortunately when we narrowed the scope of the project, our primary focus was to be on human detection and navigation with Blue.

Initially a rudimentary delivery cycle between 4 arbitrarily selected static positions throughout the VxLab, known as move goals, was implemented for Blue using a bash script. The script would compare the robot current position against the move goal to gauge when it achieved its target. Although this approach worked, it was more as a proof of concept as it was very inextensible, given that it would be extremely difficult to integrate with the human detector.

The delivery cycle was upgraded to be controlled via a python script, located in the rosie scripts directory, `./blue_delivery.py`. Originally, we attempted to implement multiple threads in the script so that one thread could manage Blue's move goal, while the other was intended to monitor the human detector once implemented. It was found that running multiple threads would not suffice, as a node is required to run on the main thread and therefore we could neither run the move goal node, nor the topic subscription node.

Subsequently, the script was altered to run multiple processes instead of threads to run the nodes on. This was successful, and the script was used throughout the human detection training and monitoring. Once human detection was implemented, the script was required to receive data from the recognized_result topic, and convert it into move goals. It was then discovered that the goals were being received on the move goal process, but when added to the goals collection it was found the state data was not easily transferable between processes.

Finally it was discovered that subscribing a node to a topic in the python script provides a callback when data was published to the topic. This allowed the final iteration of the script to be single threaded, and simply initialise a client node which publishes move goals to Blue, whilst also subscribing that node to the human detection ROS topic. While there are no humans detected, the client node is passed one of 4 arbitrarily selected static positions throughout the VxLab, known as move goals. Blue systematically moves between these 4 move goals, while the human detector scans objects in an attempt to recognize a human model.

Project Outcomes

Demo Video
[https://drive.google.com/file/d/1pbgYpISdYYJxAhpS5y4V_yZZRbDFFRhJ/view]

GitHub Repository [https://github.com/s3724447/PP1-baxter]