Mobile Robotic Arm with Voice Instruction

CPE 800A Project Final Report

Sivankit Bhanot
Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, USA
sbhanot@stevens.edu

Prof. Shucheng Yu
Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, USA
syu19@stevens.edu

***Abstract*—We work towards the creation of a smart robotic arm by adding on to it components of vision in the form of a camera, sensing in the form of LIDAR sensor, voice instruction, and mobility. This work builds from the work of S.Jain et al [1]. We have been able to go past the stages attained in that work by exploring and working on establishing communication between the various components. While first we were able to set up the four components individually, we have now begun work towards communication between the object detection module and the physical robotic arm over ROS. We have also been able to finalize the motors and build the chassis plates for the custom vehicle that is designed to house the robotic arm, and thereby add mobility to it.**

NOTE: Working in a team with Krishna Singh(KS) and Haoyang Zhang(HZ).

***Index Terms*—Interbotix Trossen RX 150, ROS, computer vision**

I. Introduction

In the following few pages we attempt to create a working model of a mobile robotic arm that can take voice instruction and accordingly pick objects. This is a continuation of and building on top of the work carried out by Sanket Jain [1]. While in that paper, the work was mainly focused on the installation and setup of the robotic arm and the camera module. While some proposals were made regarding interaction between the two, there were only at the proposal stage. We will attempt to explore methods for making the components: The robot arm, the camera module, and the LIDAR sensor interact with each other, and work towards adding the following two functionalities: a) Mobility: We plan on achieving this with the design of a robust and powerful vehicle that can support the mass of the robotic arm. b) Voice instruction: This will be explored once all the other operations are in place. Eventually we will have to look at installing the microphone on the robot and running speech recognition scripts to aid in the process.

Sanket et al [1]was successful in making the robot arm move in a particular trajectory and then install a camera module that could perform real-time object detection. For the initial stage and prelminary work, we work to get the similar setup and functionality with the robot arm, the camera module, and the LIDAR sensor. For these tasks the Robot Operating System (ROS) middleware suite, and Jetson Nano computer were used.

Once we are able to attain the setup of these three main components, we will explore ROS further to make the three components interact with each other and be able to react to any detection, and from there, add on complexity in the form of mobility and voice instruction.

II. Problem Statement

The ultimate aim of this activity is to develop a robotic arm on a vehicle that can identify objects and perform actions on it. However, there is a large learning curve required and hence this is a longer term goal. In the intermediate stages. The first of these would be the setup of the individual components. The second would involve writing and building nodes for interaction over ROS. In the final stage, we will add on voice functionality and mobility.

In this report, we detail our progress up to the middle of the second intermediate stage as we work to develop nodes for interaction between our object detection module and the robotic arm.

For each of these stages, we have divided the task into four building blocks comprising of the arm, the camera module, the LIDAR sensor, and the vehicle on which the system would be mounted. The apparatus for this setup is as shown below:

1) Trossen Robotics ReactorX150 Robotic Arm
2) NVIDIA Jetson Nano Computer
3) Monitor for Head Mode operation and setup of Jetson Nano , (accompanied with HDMI cable, mouse, keyboard)
4) Garmin LIDAR Lite V3 Module (along with jumper cables for extending length of connection)
5) Raspberry Pi Camera V2 Module

For the initial stages, the primary challenge that we face is making the robotic arm move about in simulation and actuality through commands in the ROS middleware suite. ROS will help us achieve the robot control. After this stage, we will focus on adding on other processes in synchronicity with the arm. These include the object detection using the camera and the LIDAR sensor. For the camera and LIDAR module that will be attached, we will be using the Jetson Nano computer to better coordinate and control. The object detection scripts being run on the camera module have to run in real time so that the robotic arm is aware of the object that has to have action performed on. Once the camera and LIDAR sensors have been successfully attached and are performing well synchronously

with the robotic arm, we will be exploring adding voice instruction also through the Jetson Nano Computer.

III. Our Approach

As described in the Problem Statement above, there are four main components to this project. These are the Robotic Arm, the camera module, the LIDAR Sensor, and the development of a vehicle for the arm. Our approach has been to first develop all of these in parallel and have them operating in the desired mode with the correct functionality. In addition to working on these separate building blocks, an understanding of ROS has also been developed as this is the protocol over which the other components would communicate.

A. The Arm and ROS

The approach for the arm involves understanding the directory structure of ROS packages and commands. The manufacturer has made available some of these packages on their Github repo [4]. This consists of four main packages which have various files with source code for making ROS nodes, the meshes and URDFS for the robot model to be able to run a simulation in Rviz and Gazebo, and some example scripts that can run some trajectories of the code. It is important to consult this repository to understand the structure of the packages. One of the packages has the SDK which will be useful in order to create and write code for the arm. Consulting the source code we can determine the various ROS Topics that the node that represents the physical arm, subscribes and publishes too.

Our approach for the arm therefore involved running some of the packages from the repository, and writing some scripts using the sdk package to get familiar with the robotic arm. A more thorough understanding of the various parts and parameters of the arm are described in the Results and Understanding section.

B. Camera Module

The camera used for adding vision capabilities is the Raspberry Pi Camera Module V2. It has a CSI port just like the Jetson Nano board has on the J13 port. A longer CSI cable was attained and used to connect the camera to the board. For the setup of the camera module, the instructions on Dustin Franklin's Jetson-Inference repository were followed [7]. Following along we were able to run image recognition and object detection scripts on various images, videos, and real time video input from the camera module.

In order to better equip the object detection script to communicate with the robot and eventually its joint motors, we also explored the making of ROS nodes for the detection script. Another of Dustin Franklin's repositories helped us with this [7]. This provided us with the knowledge of various topics that this `detecNet` node subscribes to and the types of messages that are published to these topics.

C. LIDAR Sensor

The LIDAR Sensor used is the Garmin Lidar Lite V3 Laser Ranging Module. It has a 6 position connector housing receptacle at the bottom center of its body, into which the the connector socket terminal (provided with it) attaches [10]. The six pins correspond to Vdc, power enable, mode control, I2C SCL, I2C SDA, and Ground. The connection and mapping of these for working with the Jetson board is described in the Results section.

The LIDAR sensor will mainly be used to keep track of the distance between the arm's gripper and the object. Once, using the camera and its object detection, the arm is able to align its wrist and gripper with the object, it will approach it in a unidirectional fashion. And it is here where the unidirectional LIDAR sensor will help as the gripper approaches the object.

D. Vehicle

To extend the reachable area, a suitable vehicle should be assembled under the robot arm. This will enable the arm to transmit an object further than a restricted space. The robotic arm with camera, and LIDAR sensor will be placed on a four wheel drive chassis. It has four regular wheels and can only move forward and backward at this time.

In the future, Arduino board will be used and it is in charge of converting the message to motor speed value. The chassis board currently is made of aluminum and four low power DC motors. The electrical parts and whole driving system have an independent power system. Two motors on the same side are powered by one LIPO 50C battery. At this time, two bread boards are used as a processing board and the battery can power motors. Figure below shows a CAD diagram of current vehicle.

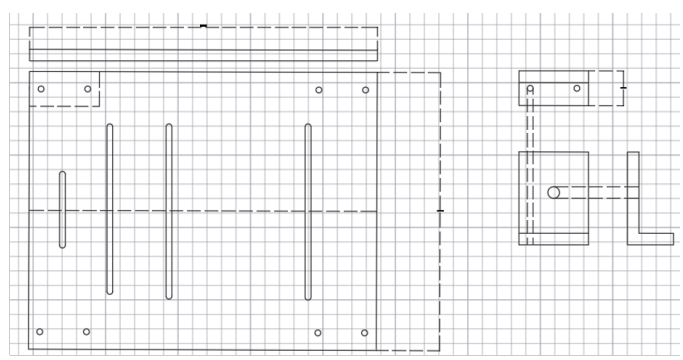

Fig. 1: CAD diagram of vehicle structure

IV. Results and Understanding

Robot Operating System is the name for a collection of tools and libraries that help in executing complex robot behavior. For the purpose of this activity, we are working with ROS Melodic Morenia on Ubuntu 18.04.

For the preliminary stages, we worked to understand the structure of ROS directories, workspaces and packages, and use simulators and node diagrams to display the functioning of some simple programs.

Catkin is one of the methods of building and organizing the ROS code. it is preferable to the older rosbuild method as it helps in easing cross-platform development. In addition

there are several filesystem tools of ROS in addition to the Unix commands on terminal. These file system tools include commands like `rospack`, `roscd`, and `rosls`.

`rospack` allows us to get information about packages, whereas `roscd` allows us to change directory directly to a package or stack, and `rosls` allows us to list files and folders in a package by name rather than using the entire absolute path. Creating a ROS package involves usage of catkin that was introduced earlier. Prior to creating the package, we must create and then change to the source space directory of the catkin workspace. After creating the package, we can build it using the `catkin_make` command. Upon building and executing this command, we observe that several folders are created in the workspace. These include `build`, `devel`, and `src`. The `build` folder is the default location of the build space and is where `cmake` and `make` are called to configure and build packages. The `devel` folder is where executable files and libraries are stored.

The key to understanding ROS and how to use it for robotic operations is first getting a grasp of nodes, messages and topics. A ROS node is an executable file within a package. It communicates with other nodes using client libraries. Examples of client libraries are `rospy`, written in Python, and `roscpp` in C++. These allow nodes written in different languages to communicate. Prior to running any ROS operation, we must run `roscore`. A message is a ROS data type that is used when subscribing or publishing to a topic. A node may publish one or more messages to a specific topic and if another node is subscribed to that topic, then the second node will receive that message.

In Figure 2 we observe a simple example whereby a simple simulator known as `turtlesim` receives messages from another node which is `turtle_teleop_key`. Figure 3 shows the node diagram for this activity. The `turtle_sim_node` and the `turtle_teleop_key` node are communicating with each other via a ROS Topic. The `turtle_teleop_key` node is publishing the key strokes over the topic and since the `turtlesim_node` is subscribed to that topic, it is receiving those messages and thereby performing the respective moves. In particular the topic in this case is the `cmd_vel` topic over which the communication is happening.

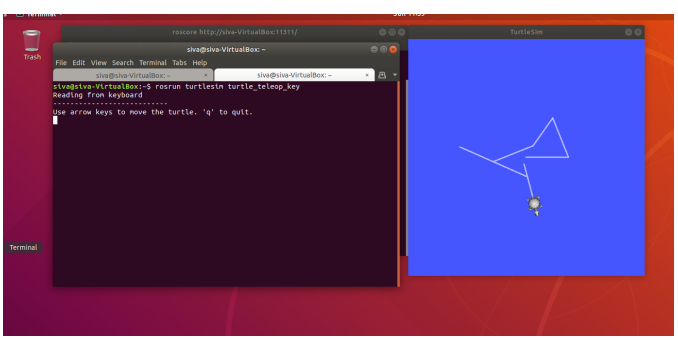

Fig. 2: Screenshot showing control of turtle in turtlesim using keyboard keys

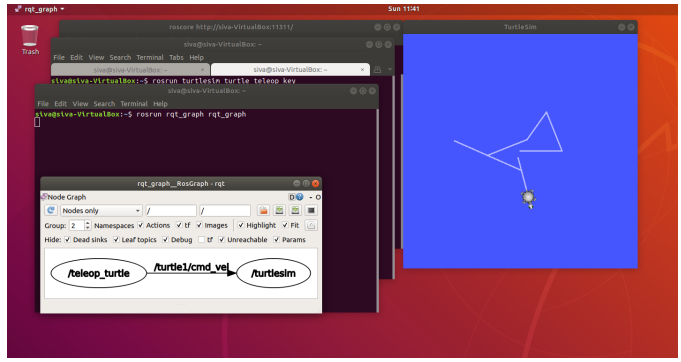

Fig. 3: Screenshot showing node diagram of above activity

The `roslaunch` command starts nodes as defined in a launch file. Following the tutorials, we made a simple launch file, where two turtle simulator nodes would start up and one would mimic the other. Figure 5 below shows the activity diagram for this. First we publish a topic, as shown in the terminal at the bottom in Figure 4. This topic is published and sent to `turtlesim1`. Then through the mimic launch file, the `turtlesim2` node follows the `turtlesim1` node.

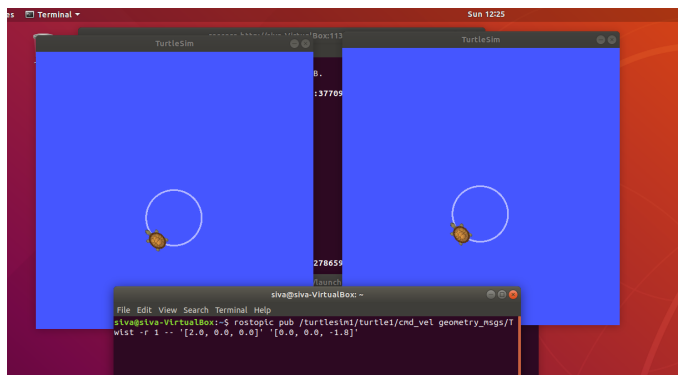

Fig. 4: Two turtle simulators with one mimicking the other as per launch file

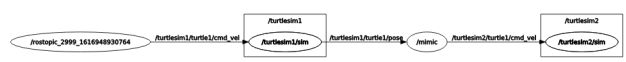

Fig. 5: Node diagram in rqt_console for above activity

A. *Motion Trajectories on Robotic Arm*

The manufacturer repository on Github has various packages to aid understanding of how the robotic arm is able to carry out movement and tasks. Much of the actions and communication over the ROS protocol can be abstracted through source code written in C++ and Python. The `interbotix_sdk` package contains the source code where the driver node and examples to run it are stored.

The Python file `robot_manipulation.py`, for example has the class `InterbotixRobot` declared. The class

has various public methods that can be called once an instance of its object has been created. The constructor of the class creates an instance of the ROS node with the name `_robot_manipulation`. This node represents the physical robotic arm. The public methods in the Python class then contain various ways to control the joints of the robotic arm, make it go to positions such as sleep and home.

This file has been used as reference to create scripts where we have made the robot follow particular trajectories.

While the `interbotix_sdk` package contains the reference class with the relevant methods required to create and run an instance of the robot, it is itself a ROS package and can also be run using the `roslaunch` command. It is composed of a launch file `arm_run.launch` which runs the driver node called `arm_node`. This node controls the physical robot. Some of the various services, publishers, subscribers, and other actions associated with the node are shown in Figs 6, 7, and 8. The complete list is provided in the manufacturer's Github page. Two of the services we are concerned with are `torque_joints_on` and `torque_joints_off`. Turning off the torque motors means the robots joints motor are off and it can be moved to any position manually. Then to freeze it in that position, we call the service to turn on torque motors. This command was used to make the robot stay in various positions. These positions made up the overall trajectory that we wanted the robotic arm to follow. Once the arm was put in a desired position that made up a part of the trajectory, and the torque motors were turned on so that the arm stayed in that position, we wrote a script to get the positions of the various joints. This involved calling a method from the `InterbotixRobot` class after creating an instance of it in the script. This method returns a list containing the positions (in radians) of the 8 joints (in the case of the RX150 robotic arm which is what we are working with). Manually positioning the robotic arm in various positions, we were able to attain a list of these values. This list was stored and then used in another script to make the arm follow the trajectory.

Services

- `/<robot_name>/torque_joints_on` - service to torque on all motors (including gripper)
- `/<robot_name>/torque_joints_off` - service to torque off all motors (including gripper)
- `/<robot_name>/get_robot_info` - service to get robot information like joint limits, joint names, and a couple preset robot poses; refer to RobotInfo.srv for implementation details.
- `/<robot_name>/set_operating_modes` - service to set firmware operating modes (like position, velocity, current, etc...) for a robot's joints; refer to OperatingModes.srv for implementation details. Note that the motors will torque off for a moment when changing operating modes so make sure that the robot is in its 'sleep' pose (defined in the motor config file) before calling this service.
- `/<robot_name>/set_firmware_pid_gains` - service to set the PID gains for position/velocity control in the motor firmware; refer to FirmwareGains.srv for implementation details.
- `/<robot_name>/set_motor_register_values` - service to set multiple motor register values at once for a user-provided register name; refer to RegisterValues.srv for implementation details.
- `/<robot_name>/get_motor_register_values` - service to get multiple motor register values at once for a user-provided register name; refer to RegisterValues.srv for implementation details.

Fig. 6: List of services that the `arm_node` node has. Listed in the `interbotix_sdk` package

While we were able to make the robotic arm conduct certain tasks and follow trajectories using this method, it was still clearly manual-intensive. We need the arm to be able to conduct its movement by itself and react to inputs based on speech or object detection.

In a step towards making the robot react to speech, we worked with the `pynput` library by using it in a script so

Subscribers

- `/<robot_name>/joint/commands` - subscribes to JointCommands messages; this topic is used to control a specified group of joints synchronously (which is more efficient than sending commands to individual joints consecutively). Refer to the message description for implementation details. This 'group of joints' can be defined in the 'order' sequence in a motor config file ('gripper' joint excluded if present).
- `/<robot_name>/gripper/command` - subscribes to ROS Float64 messages to control the gripper joint (if present). Refer to the 'arm_write_gripper_command' function in the header file for implementation details.
- `/<robot_name>/single_joint/command` - subscribes to SingleCommand messages; this topic is used to control a single joint. Refer to the message description for implementation details.
- `/<robot_name>/arm_controller/joint_trajectory` - subscribes to ROS JointTrajectory messages; this topic is used to send desired trajectories to a specified group of joints. Refer to the 'arm_joint_trajectory_msg_callback' function in the header file for implementation details.
- `/<robot_name>/gripper_controller/gripper_trajectory` - subscribes to ROS JointTrajectory messages; this topic is used to send desired trajectories to the gripper. Refer to the 'arm_gripper_trajectory_msg_callback' function in the header file for implementation details.

Fig. 7: List of topics that the `arm_node` node subscribes to. Listed in the `interbotix_sdk` package

Publishers

- `/<robot_name>/joint_states` - publishes ROS JointState messages at 100 Hz; note that in general, positions are given in radians, velocities are given in radians per second, and effort is given in milliamps. If the gripper is being used, the positions for the 'right_finger' and 'left_finger' joints are given in meters relative to a virtual 'fingers_link' placed dead center (launch a robot model in Rviz to get familiar with the link and joint names/positions).

Fig. 8: List of topics that the `arm_node` node publishes to. Listed in the `interbotix_sdk` package

that we could map certain keys to perform increments or decrements operation on the particular joint motors.

B. Image Recognition and Object Detection

After successfully connecting the Raspberry Camera Module V2 on the Jetson Board port J13 using the CSI cable, we were able to download some neural networks pre-trained and made ready by Nvidia. Building these downloaded files and repository, we were able to write our own scripts for image recognition and object detection. The source code of these is attached in the Appendix.

First, we ran the image recognition neural network on some images and then ran our script on real time camera input that was being received by the camera module. The default neural network in this case is GoogleNet. Other networks can be downloaded and a different network can be selected by setting the `--network` parameter in the command in the terminal.

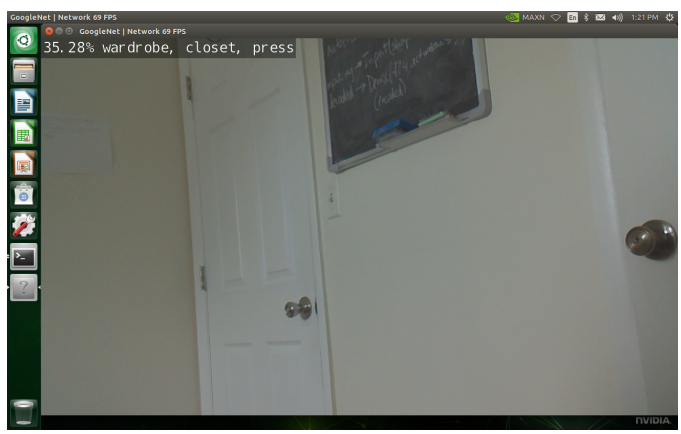

Fig. 9: Running `imagenet` object on camera feed

Running the object detection script on camera feed, we realized that it is better suited for our purposes where the robot would need to identify an object, orient one of its motors

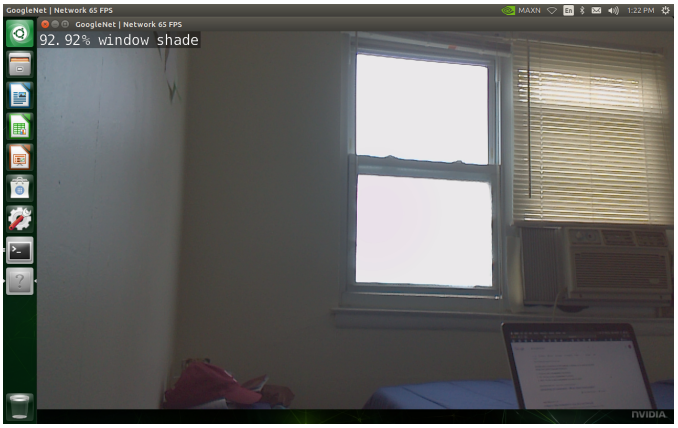

Fig. 10: Running `imagenet` object on camera feed

(wrist rotate) towards it and then move towards that particular object. For this reason the object detection's bounding box overlayed on top would help with latter processes where we are concerned with the orientation and location of a particular object within the screen or view of the camera. Going through the source code of the `detection.py` file we see that the default neural network used for detection is `ssd-mobilenet-v2`. Figs **??**ss show some examples of the script running in real time on camera feed and detecting multiple objects.

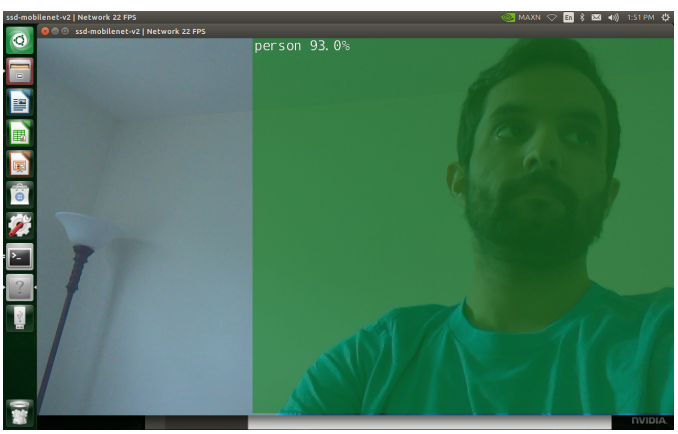

Fig. 11: Running `detecnet` object on camera feed

While we were able to perform real time object detection using the camera module, our aim is to make it communicate with the robotic arm. In order to do this we need to create a node of the object detection module. As mentioned earlier, the source code for the creation of this node is available on Dustin Franklin's Github repository. The source code for this repository is in C++ and stored in the file `node_detecnet.cpp`.

In order to make this node communicate with the node representing the physical robot, it is essential for us to understand the various topics that the `detectnet` node subscribes and publishes to. The list is shown in Figure 14.

Since we want the robot to react to detection of an object, the topic that we are concerned with is `detections`. To

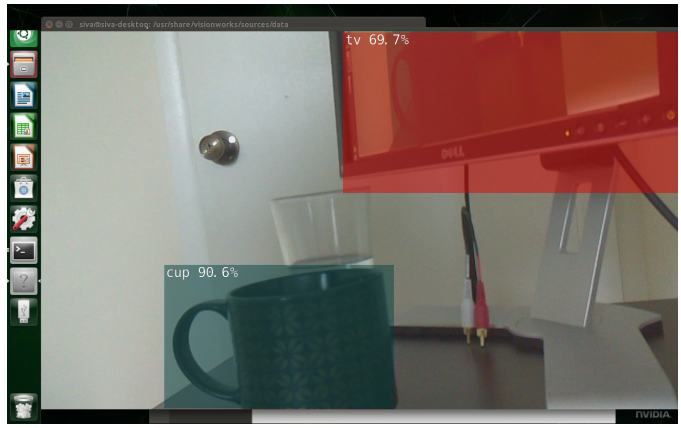

Fig. 12: Running `detecnet` object on camera feed

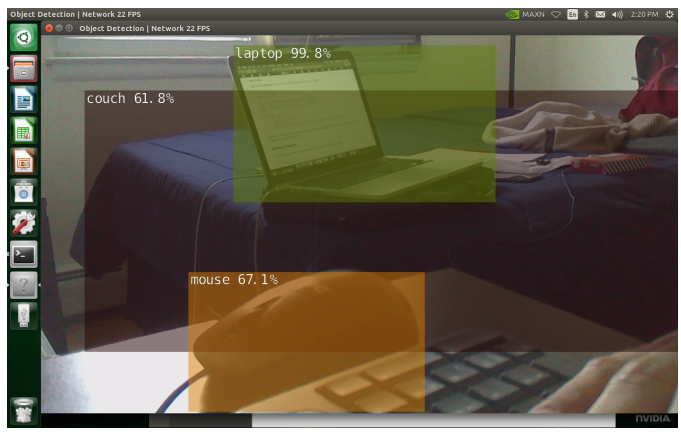

Fig. 13: Running `detecnet` object on camera feed

better understand the structure of the message that is published on this topic, we explored the type of the message. Following along the various message types, we were able to comprehend that the part of the message that we are concerned with is that which expresses a position and orientation on a 2D manifold. This message has three components, all of `float64` type. These are `x`, `y`, and `theta`. The `x` and `y` are the coordinates at the center of the bounding box of the object detected.

detectnet Node

Topic Name	I/O	Message Type	Description
image_in	Input	`sensor_msgs/Image`	Raw input image
detections	Output	`vision_msgs/Detection2DArray`	Detection results (bounding boxes, class IDs, confidences)
vision_info	Output	`vision_msgs/VisionInfo`	Vision metadata (class labels parameter list name)
overlay	Output	`sensor_msgs/Image`	Input image overlayed with the detection results

Fig. 14: List of topics that the `detectnet` node publishes and subscribes to. We are concerned with the `detections` topic which outputs a message of type `vision_msgs`

vision_msgs/Detection2DArray Message

File: vision_msgs/Detection2DArray.msg

Raw Message Definition

```
# A list of 2D detections, for a multi-object 2D detector.

Header header

# A list of the detected proposals. A multi-proposal detector might generate
#   this list with many candidate detections generated from a single input.
Detection2D[] detections
```

Compact Message Definition

```
std_msgs/Header header
vision_msgs/Detection2D[] detections
```

autogenerated on Thu, 06 Jun 2019 19:10:35

(a) Detection2DArray message and its contents

vision_msgs/Detection2D Message

File: vision_msgs/Detection2D.msg

Raw Message Definition

```
# Defines a 2D detection result.
#
# This is similar to a 2D classification, but includes position information,
#   allowing a classification result for a specific crop or image point to
#   to be located in the larger image.

Header header

# Class probabilities
ObjectHypothesisWithPose[] results

# 2D bounding box surrounding the object.
BoundingBox2D bbox

# The 2D data that generated these results (i.e. region proposal cropped out of
#   the image). Not required for all use cases, so it may be empty.
sensor_msgs/Image source_img
```

(b) Detection2D message and its contents

vision_msgs/BoundingBox2D Message

File: vision_msgs/BoundingBox2D.msg

Raw Message Definition

```
# A 2D bounding box that can be rotated about its center.
# All dimensions are in pixels, but represented using floating-point
#   values to allow sub-pixel precision. If an exact pixel crop is required
#   for a rotated bounding box, it can be calculated using Bresenham's line
#   algorithm.

# The 2D position (in pixels) and orientation of the bounding box center.
geometry_msgs/Pose2D center

# The size (in pixels) of the bounding box surrounding the object relative
#   to the pose of its center.
float64 size_x
float64 size_y
```

(c) BoundingBox2D message and its contents

geometry_msgs/Pose2D Message

File: geometry_msgs/Pose2D.msg

Raw Message Definition

```
# Deprecated
# Please use the full 3D pose.

# In general our recommendation is to use a full 3D representation of everything and for 2D specific applications make the app

# If we have parallel copies of 2D datatypes every UI and other pipeline will end up needing to have dual interfaces to plot e


# This expresses a position and orientation on a 2D manifold.

float64 x
float64 y
float64 theta
```

(d) Pose2D message and its contents

Fig. 15: The various message types contained within the detections message. Going through these we understand that the position and orientation on a 2D manifold of the object detected is provided in the Pose2D message which is wrapped in the BoundingBox2D message, which is in turn wrapped in the Detections2D message and this is further wrapped in Detections2DArray message. Essentially the `detections` message returns an array of the objects detected and in each

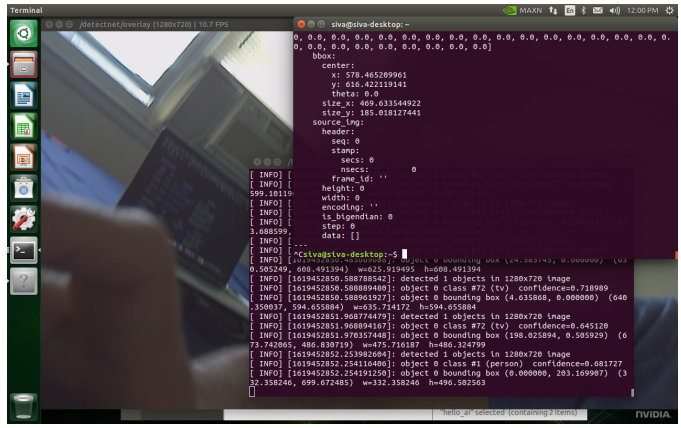

Fig. 16: Running the rostopic echo on the /detections topic.

Using this information about the type of message that is published to the `detections` topic, we ran the command to list the message being published to this topic once the `detectnet` node was running live. This displayed a list (array) of all the objects detected within the window of the camera. Fig 16 shows the message for one of these detections. In this case it is detecting a person (the user), who cannot be seen as the bounding box is behind the terminal windows. Focusing on the Detection2D message we see that it displays the `x` and `y` coordinates of the center of the bounding box as well as the `theta` value all of which constitute the `Pose2D` message that is wrapped in the `BoundingBox2D` message that is wrapped in the `Detection2D` message, lots of which make up the `Detection2DArray` message, all of which are shown in Fig 15.

C. Running LIDAR

The LIDAR module can operate in two main configurations: I2C and PWM. In the I2C configuration, it acts as a slave device controlled by the master device which is a microcontroller (in our case the Jetson Nano board). In the PWM configuration, the mode control pin provides a means to trigger acquisitions and return the measured distance via Pulse Width Modulation without needing to use an integrated circuit interface [10]. For now, we have used the former configuration by connecting the LIDAR to the bus (J38 on the Jetson Nano). The specific pin connections are shown in Table I. This pin configuration was identified and used after consulting the manufacturer's repository on Github [11]. On the repository the connection configuration is described for Raspberry Pi but for the Jetson Nano, it is identical.

Wire Color	Function	Pin
Red	VDC(+)	4
Black	GND(-)	6
Blue	SDA	3
Green	SCL	5

TABLE I: Pin configuration of LIDAR module

Once the sensor was successfully connected to the pins on the Jetson board and detected, we were able to run the C++

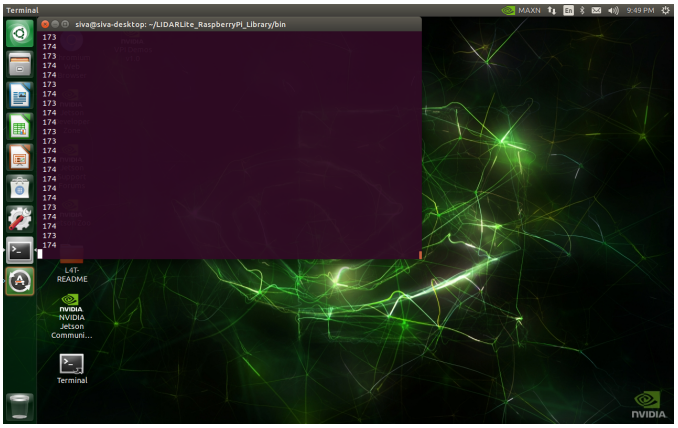

Fig. 17: LIDAR readings printed on console

file that prints the readings of the sensor on the console. This is shown in Fig 17

D. Chassis for the Vehicle

Based on the CAD diagrams designed by HZ and the motors finalized by HZ and KS, HZ was able to also attain the metallic plates fabricated in the Stevens MakerSpace. It consists of two layers. The plates are shown in Figs

Fig. 18: Layer 1 of Metal Chassis for Vehicle

V. Conclusions and Future Work

The work for this project thus far is on-going and continuing into the next semester. We have been successful in understanding the ROS structure of the packages of the robotic arm. Based on the `sdk`, we have also been able to write scripts whereby the arm follows certain pre-determined trajectories. With the camera module, we have run object detection in real time. Along with this we are able to run the ROS node for this object detection. Following investigation into the ROS node for object detection (known as the `detecnet` node), we have understood what are the relevant messages and parts that we

Fig. 19: Bracket for motor will be fixed on Layer 1

Fig. 20: Layer 2 of Metal Chassis for Vehicle

need. The `detectnet` node publishes a Detections2DArray message to the `detections` topic. This message contains various bits of information among which we are concerned with the center `x`, `y`, and `theta` for each detection and bounding box.

Going further into this project, our next tasks will be concerned with writing and implementing an intermediary node that can subscribe to the topics that the `detectnet` node publishes to. From the message published to the `detections` topic, the intermediary node will extract the relevant information about the orientation of the object detected. This intermediary node will then publish to another topic the commands required to move a particular joint of the robot arm. In order for this to happen, the `_robot_manipulation` node, that represents the physical robotic arm will subscribe to the topic to which the intermediary node publishes the commands to. For the next steps, we will be looking at sending a command to control just one joint, namely the wrist rotate. Once meaningful communication between the `detectnet` node and the `_robot_manipulation` node, through the intermediary node, that we will design, is achieved, robotic path planning and machine learning techniques can be explored for control of other joints in response to object detection. A deeper investigation into the physical robot node's services, publishers and subscribers would also be required similar to the one that has been done above for the `detectnet` node. There are some challenges that we face currently even in the understanding of this object detection node. One is that while the `detections` topic consists of message that has orientation of the bounding box in the form of its center's `x`, `y`, and `theta`, it has been observed that the value of `theta` remains zero no matter how much the orientation of the object is moved. When the values were printed live on the console with the camera running, and the camera was moved about, while the `x` and `y` values changed, the `theta` value remained zero. This will also require further investigation. Looking into the source code of the node in C++, we were able to see this value has been assigned zero for default settings. We will need to look into how to get a measure of this as it would be a major step in making our robot's gripper or wrist joint come 'face to face' with the object.

We are also currently working on a script that will make the arm scan a room for the objects and then move about to perform tasks on one of the objects.

The next few steps will also involve designing a node that wraps the LIDAR sensor output. Currently to get the LIDAR sensor values, we run a C++ file. These distance values can be published to a particular topic to which again the `_robot_manipulation` node can subscribe and thereby act accordingly by moving a particular joint motor forward towards the object.

It has been mentioned earlier that we also explored the usage of `pynput` library to gauge the response of the various joint motors to assigned keys on the keyboard. The tests for these are ongoing currently. The reason we pursue these are because in the longer run they could prove useful when attempting to add voice responsiveness and activation. This would involve adding a microphone to the ensemble, then perform NLP to understand and output text corresponding to voice input. This text could be used to provide similar stimulus to the robotic motors like the particular keys would do.

We have been successful thus far in making the robotic arm perform various pre-defined activities and trajectories, run image recognition and object detection using the camera module on the Jetson Nano, run the LIDAR sensor module and output the distance measurements onto the console, and build the chassis for the vehicle that will hold the arm. The four main components of the projects have been developed in parallel successfully.

We are now on track for the next stages of this project which largely involve investigation into the ROS topics, messages, subscriptions and nodes, to enable communication between these four components. The work described in the next stages has been described in the paragraphs above and we are currently working towards these.

References

[1] S. Jain and S. Yu, "Robotic Arm Control with Computer Vision,".

[2] Y. Wang, L. Liu, X. Zhang, W. Shi, "HydraOne: An Indoor Experimental Research and Education Platform for CAVs,"

[3] "ROS Tutorials." ROS.org, Creative Commons Attribution 3.0, 4 Feb. 2021, wiki.ros.org/ROS/Tutorials.

[4] Trossen, Matt, et al. "InterbotiX X-Series Arm & Turret ROS Packages." Github, Github, Inc, 29 Oct. 2020, github.com/Interbotix/interbotix_ros_arms.

[5] "Getting Started with Jetson Nano Developer Kit." NVIDIA Developer, Nvidia Corporation, developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit#intro.

[6] Weiwei, Mu. "7th International Conference on Machinery, Materials and Computing Technology (ICMMCT 2019)." A Study of Robot Arm Control System Based on ROS, vol. 7, 2019.

[7] Franklin, Dustin, director. Real-Time Object Detection in 10 Lines of Python Code on Jetson Nano, NVIDIA Corporaton, 31 Jan. 2020, news.developer.nvidia.com/realtime-object-detection-in-10-lines-of-python-on-jetson-nano/.

[8] M. Quigley et al, "ROS: an open source Robot Operating System, ".

[9] Megalingam, Rajesh Kannan, et al. "4th International Conference on Computing Communication and Automation (ICCCA)." Keyboard-Based Control and Simulation of 6-DOF Robotic Arm Using ROS, vol. 4, 2018.

[10] "Garmin Lidar Lite v3 Operation Manual and Technical Specifications" Garmin Ltd and its subsidiaries, 2016.

[11] "LIDAR-Lite Raspberry Pi Library" Github, Github, Inc, 2019, https://github.com/garmin/LIDARLite_RaspberryPi_Library

Appendix

I - `get_joint_data.py`

```
from interbotix_sdk.robot_manipulation import
    InterbotixRobot
from interbotix_descriptions import
    interbotix_mr_descriptions as mrd
import sys
from time import sleep

def main():
  arm = InterbotixRobot(robot_name="rx150", mrd=mrd)
  print arm.get_joint_positions()

if __name__=="__main__":
```

```
  main()
```

Listing 1: Prints values in radians of the 8 joints of the robotic arm to console

II - joint_position_control.py

```
from interbotix_sdk.robot_manipulation import InterbotixRobot
from interbotix_descriptions import interbotix_mr_descriptions as mrd

#jp = [[0.030679617077112198, -0.03221359848976135, 1.607611894607544, -0.1288543939590454, -0.012271846644580364],
 #     [0.9035146832466125, 0.1257864236831665, -1.0170292854309082, 1.0737866163253784, 0.07363107800483704],
  #    [-2.5095925331115723, 0.0076699042692780495, -0.45712628960609436, 0.4939418137073517, -0.11811652034521103]]

jp = [[0.14, -0.21, -0.03, -0.35, -0.44],
      [0.14, -1.38, -0.03, -0.35, -0.44],
      [0.14, -0.01, -0.01, 1.75, -0.44]]


def main():
  arm = InterbotixRobot(robot_name="rx150", mrd=mrd)
  arm.go_to_home_pose()

  arm.set_joint_positions(jp[0])
  arm.set_joint_positions(jp[1])
  arm.set_gripper_pressure(1.0)
  arm.open_gripper()
  arm.set_joint_positions(jp[2])
  arm.go_to_home_pose()
  arm.close_gripper()
  arm.go_to_sleep_pose()


if __name__=="__main__":
  main()
```

Listing 2: Makes robotic arm perform trajectory involving 3 stops

III - my-recognition.py

```
#!/usr/bin/python3

import jetson.inference
import jetson.utils

import argparse

#parse the command line
parser = argparse.ArgumentParser()
parser.add_argument("filename", type=str, help="filename of the image to process")
parser.add_argument("--network", type=str, default="googlenet", help="model to use, can be: googlenet, resnet-18, etc. (see --help for others)")
opt = parser.parse_args()

#load the image specified
img = jetson.utils.loadImage(opt.filename)

#load the recognition network
net = jetson.inference.imageNet(opt.network)

#classify the image
class_idx, confidence = net.Classify(img)

#find the object description
class_desc = net.GetClassDesc(class_idx)

#print out the result
print("Image is recognized as '{:s}' (class #{:d}) with {:f}% confidence".format(class_desc, class_idx, confidence*100))
```

Listing 3: Performs image recognition on camera feed with googlenet neural net

IV - my-detection.py

```
import jetson.inference
import jetson.utils

#load the object detection model
net = jetson.inference.detectNet("ssd-mobilenet-v2", threshold=0.5)
camera = jetson.utils.videoSource("csi://0")
display = jetson.utils.videoOutput("display://0") # ''my_video.mp4' for file

while display.IsStreaming():
  img = camera.Capture()
  detections = net.Detect(img)
  display.Render(img)
  display.SetStatus("Object Detection | Network {:.0f} FPS".format(net.GetNetworkFPS()))
```

Listing 4: Performs object detection on camera feed with ssd-mobilenet-v2 neural net