UNIVERSITY OF FLORIDA

Senior Design Final Report

Fetch Robot

Dylan Falzone
University of Florida
Senior Design EEL 4924C
Gainesville, United States
dylan.falzone@ufl.edu

Jonathan Feitel
University of Florida
Senior Design EEL 4924C
Gainesville, United States
jonathanfeitel@ufl.edu

A Faculty Project for Dr. Bobda

Table of Contents

***Abstract*— The Zebra Industries Fetch Robot, equipped with wheels, cameras, and a highly precise poseable arm, has immense potential as we enter an era of AI powered machines. Robots like these, with the ability to interact autonomously with their environment, have unthinkably many uses in industries ranging from fulfillment to assisted living. This report describes one simple, but far-reaching use-case, as a proof of concept for what this robot, and others like it, are capable of.**

Keywords*— *ROS, Ubuntu, Linux, Nodes, Topics, YOLO, AI, Machine Learning, Python, Darknet, Neural Network.

I. INTRODUCTION

For years, robotics applications have been limited to those operated by humans, or those whose tasks are incredibly simple and repetitive. With robots relegated to assembly lines and remote controlled devices, the idea of autonomous, independent robots able to assist humans in their daily lives seemed set to remain science fiction. However, recent advances in a seemingly unrelated field promise to break this status quo. Artificial intelligence, especially in the form of machine learning models like neural networks, has redefined the scope of what machines are capable of. While large language models like GPT may have taken more of a spotlight, it is actually the lesser known image recognition models that pose the most significance to robotics. With the ability to identify the objects in an image with incredible accuracy, these models allow robots to engage with their environments in ways that were previously impossible.

With this project, we aim to explore a simple scenario as a proof of concept for the burgeoning world of possibilities that AI has unlocked for robotics.

II. PROJECT FEATURES AND OBJECTVIES

Figure 1: Front, back, and side view of Fetch Robot

The Fetch Robot, pictured in figure 1, is equipped with a variety of mobility and sensing functionalities. It is capable of omni-directional movement, precise object manipulation, wireless communication, and stereo video capture. These attributes make it an ideal platform for this project.

In order to demonstrate the capabilities of an AI powered robot, particularly in practical applications, performing tasks that only humans have been able to perform, we have devised the following scenario as our goal for this project. The Fetch robot should autonomously patrol a space, in search of an object, in this case, a bottle. It should approach and collect the bottle, before searching for and delivering it to a nearby person. During this process, it shall receive no control signals from a human operator, and it should complete the task without causing any damage to itself or its surroundings.

If a team of two undergraduate students, largely unfamiliar with robotics programming, are able to complete such a goal within a single semester, it will effectively demonstrate just how capable robots have become and how promising a future they have in scenarios like fulfillment and assisted living.

III. TECHNOLOGY SELECTION

In traditional coding, one would write a large file running through logic and operations. This would have an accompanying file with all the necessary functions and variables. For our project, we used a ROS style of coding. This means different functionalities are put into different sections of code, called nodes. These nodes all run simultaneously and rely on a messaging service, topics, to relay information between the nodes. This allowed for easy debug and testing since we could implement one node at a time, and once this node was working, it never had to be modified again.

Linux was an easy choice as ROS for Linux has a vast online presence. This made trouble shooting quite easy. Using Ubuntu to run Linux was also an easy choice as there are a multitude on online tutorials on this topic. YOLO, you only look once, is the AI software we used to identify either a water bottle or a person. The image data from the robot was fed into the YOLO code and the same image would return with a box drawn around any identified object.

IV. Project Architecture

The architecture for this project went through several iterations, explained in depth in the 'obstacles overcome' section. The version we finished with was by far the most straight-forward and efficient. For this section, we will primarily focus on the ROS architecture, as that is where the majority of our design work was done.

Figure 2: ROS nodes and topics visualized by flowchart.

Our ROS program followed a relatively simple structure. In figure 2, all of our primary nodes (the ones that we wrote ourselves) are depicted to have different colors and sizes. The larger nodes contain more logic, and the smaller ones are far simpler. The blue nodes are to be launched first from the main1 launch file, as they are the basis for the rest of the nodes. Darknet and arm_movement are essential to have running before everything else as they launch the YOLO model, and create necessary move groups that the other nodes will utilize. The green nodes can be launched after the fact, though, distance service should be launched before the rest of them due to its unique communication method with a server rather than another node.

To put it simply, the main file begins a chain reaction that touches each node. It runs toggle, prompting distance service to examine the darknet data and decide whether the robot is a near, far, or unknown distance from its goal. This conditional will prompt the robot to either spin and check elsewhere, drive forward to get closer, or reach for the bottle. Once the selected action is completed, a done signal will be sent back to main, which will prompt it to trigger again. As the robot conducts its actions, and signals propagate through the nodes, the robot approaches its goal.

Upon completing arm_to_target specifically, the desired object in distance_service will alternate between bottle and person, so that the robot will take the bottle to a person and once it's done, it will seek out another bottle. This program architecture proved very robust and effective for our purposes.

V. Hardware/Software Selection

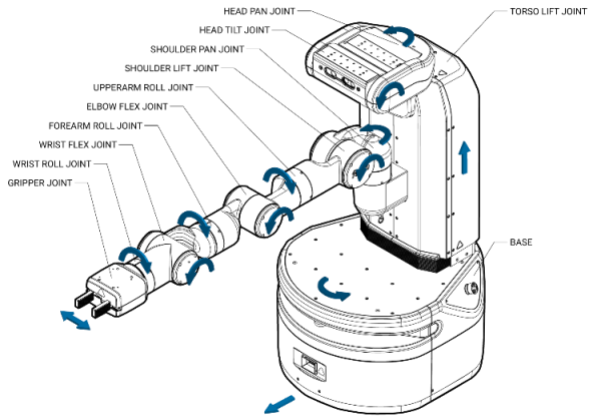

Figure 3: Movement functionality on Fetch Robot.

A. *Hardware*

All hardware used in this project was an already existing component on the Fetch robot. Stereo imaging camera, 7 motor arm, gripper, and two wheels was all that was required to implement the design goal.

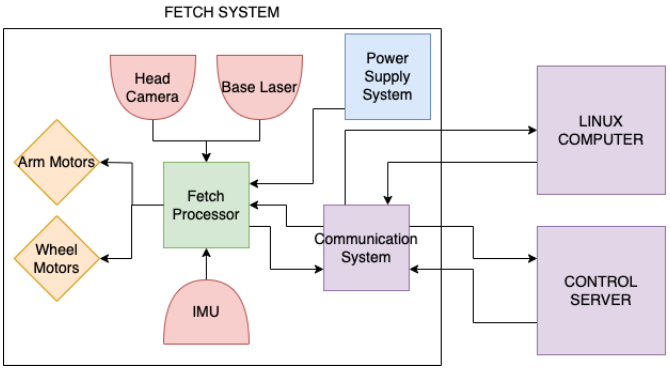

Figure 4: Hardware block diagram

The stereo imaging camera was using to obtain a depth map of the area, allowing the robot to know exactly where in 3-D space the bottle or person was. The camera also allowed us to identify objects using

AI. The arm/gripper configuration was ideal for picking up the bottle and moving it around. Having so many degrees of freedom allowed the arm to avoid objects in its environment, such as tables or humans. Lasty, the wheels were used to move the robot closer or further from the objects as well as rotate the robot to zero-in on its desired location.

B. *Software*

The software used was largely Python scripts with a few launch files, which are ROS specific files used to open multiple nodes at once. Initially, there were some scripts already provided that had some functionality for the arm moving and obtaining distances to the objects, but in the end, we had so scrap most of the existing code. This was because the scripts were written in a way that made running them as a cohesive ROS system could not work.

The most important code for the design goal was designed to take data from the stereo cameras and compute a distance to every pixel in the image. This map of distances was then overlayed with the visual image to give each visual pixel a distance value. The image data was also passed into an algorithm called YOLO that returned an image with a box drawn around objects it could identify. In our code, we would only look at boxes drawn around either bottles or persons, depending on which object we were looking for. The previous code used to find the distance from the object to the robot would take each pixel in that box and average the distances to obtain one distance value. A major flaw was that some pixels within the box were not the object, and were in fact many meters behind the object, causing erroneous readings. This caused the distance value to be unusable for precision applications such as picking up a bottle. Our simple fix was to take only the reading from the center of the box as this was guaranteed to be a pixel on the actual object and would give us accurate and precise data to work with.

After identifying and obtaining a distance value, the code then used other nodes to move the robot to within grabbing distance. The movements would be triggered by massages from the distance computing node to either rotate the robot or drive the robot. Sometimes data received would be off by many factors and the distance was computed as all zeros. To fix this we would merely move the robot back if any distance was less than the grabbing distance.

The arm movement code was then designed to transform the distance value and coordinates into robot coordinates. It is important to note that the coordinate system the camera uses can be linearly transformed into the robot coordinate system. Zones for the objects were placed in the robot's 3-D map so that when the arm was planning its route to move from a stored state to a grabbing position, it would not hit anything in the real world.

Gripper code was designed as an action server instead of standard nodes and topics. This was immensely useful because a maximum effort could be set, translating to a squeezing force on the bottle, allowing any sized bottle to be picked up without extra coding. A goal of 0.01 meters was used as the desired distance between grippers, and if the maximum effort was exceeded before reaching that goal, it would not squeeze any harder.

The last piece of code delt with the head movement. In testing, it was discovered that the head would move around left and right and the camera would move up and down. This movement made the coordinate systems offset in ways we could not account for. An action server was used to zero the head and camera position. Any time the head experienced a force, when the robot rotated or moved, the head would move slightly, the action server would see this difference in the current state and the goal sate and autonomously move the head back the zero position.

VI. Design Procedure used

Initially lacking any ROS development background, our team used the first few weeks to understand and experiment with the Fetch's ROS system. Obtaining a clear understanding of ROS and how it functions was monumentally important in developing and troubleshooting. The next step was then to outline what tasks needed to be done and divide them up into chunks of code, nodes, that could be developed and tested individually. After writing a few functioning nodes, developing communication lines between the nodes, topics, and using standard programming logic to deduce when certain actions needed to be performed led to a function ROS system. Experimenting with offsets and parameters that

adjusted purely theoretical code to fit the real world allowed for a function Fetch system.

VII. Obstacles Overcome

A. Connecting to the Fetch

Initially, we intended to use our laptops to program and control the fetch robot. Both of us had some level of familiarity with virtual machines, or multi boot setups, and so we spent the first several weeks attempting these methods. We found that we were able to communicate with both the robot and the server from our devices, however, we were unable to receive certain types of data from the robot. Unfortunately, this data included the position of the robot's arm, and the images taken by the robot's camera, which were essential for the project.

After spending some time trying to fix this issue, which was likely a result of the complexities of routing messages to a VM, we decided to just use the server directly instead of our laptops. While this did mean only one of us could utilize the keyboard at a time, it was the only way we could progress and so it was a worthwhile decision. We used this method successfully for the duration of the project.

B. Recovering functionality

When we first began this project, we were under the impression that the Fetch robot already contained the code necessary to drive, move its arm, and interpret camera data properly, and that we would merely be responsible for designing a program that drew upon these pre-built functionalities. This was partially correct, as the scripts for interpreting the camera data were essentially complete. The rest however, were not. The robot's arm could move, but the gripper could not be opened or closed, and any drive scripts beyond the base level keyboard input script were non-functional.

We spent the majority of the semester attempting to fix and finish these scripts and 'restore' functionality to the robot. Ultimately, we were successful in getting several of these scripts working individually. It wasn't until two weeks before the end of the semester that we realized none of the scripts worked together. Many of them used unconventional practices, creating overlapping move groups and an incredibly hard to follow web of topics and nodes. With the deadline closing in, we decided the only way forward was to ditch the old scripts entirely and rewrite them anew, purpose built to work together in pursuit of our project goal.

This choice did cause significant crunch for the following two weeks, but it enabled us to progress at a rate that simply wasn't possible with the previous code. We were able to complete the project on time, so it was clearly the right choice.

VIII. Project Logistics

A. Project Schedule/Gantt Chart

Our project was split into 12 weeks of development. The first few weeks revolved around becoming familiar with ROS and its unique aspects of coding. Followed up by a few weeks of developing functionality for the Fetch robot and the server. The last few weeks revolved around the demo and finalizing and tweaking the design so that it was successful every time the robot attempted to pick a bottle up.

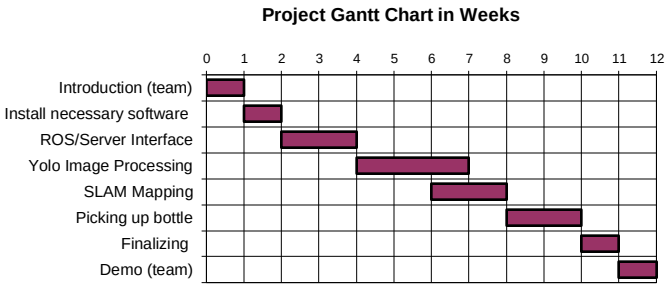

Figure 5: Project Gantt Chart

B. Division of Labor

The work was split evenly between our team. Sometimes working collaboratively on the same piece of code to develop and troubleshoot, other times working on individual pieces of code and reconvening to incorporate them together. Jonathan Feitel worked on much of the development for moving the robot and gripper and Dylan Falzone developed much of the code that integrated the code together with messages. In the end, both team members contributed to every part of the project. This led to each member having a full understanding of the project and the robot's abilities and limitations.