Basic Autonomous Ball Pick up Using Ur5e

Charlie Caputo

***Abstract* – I was tasked with moving a UR5E robot to pick up a ball and place it somewhere else using autonomous control, starter codes, codes we developed, and simulators. In this paper I will discuss my methods to finding a solution and the roadblocks that I ran into. Throughout the process, I learned techniques that helped me debug code and think of new ways to arrive at a solution.**

I. Introduction

For this project we were given the task of making a UR5E robot pick up a ball and move it somewhere else. Specifically, the goal was to ensure that the robot would make controlled and slow movements through each of the points to complete its task. I did this by using simulated environments to control a simulated robot for testing codes. There are four points that the robot needs to flow through as depicted in (Fig. 1).

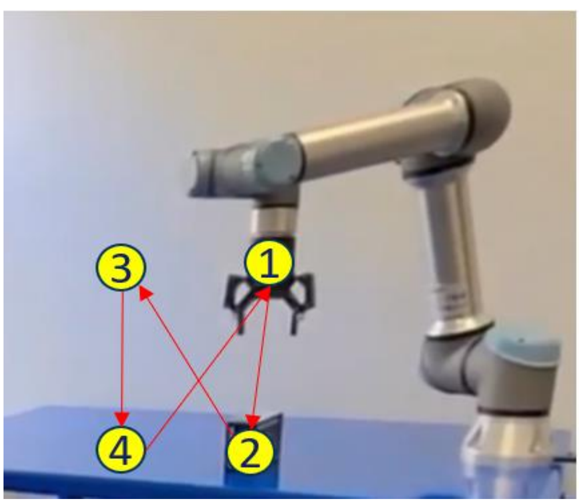

Fig. 1: 1) starting at a safe initial position, 2) Reach a position to pick up the object, 3) Move to a save spot above the target position, 4) Place the object at the target position [1]

In this paper I will first discuss the tools that were given to achieve this goal and the code I created to complete the task. Then how the problem was approached and solved. Finally, the results of the task will be revealed along with an in-depth discussion of the problems that ensued and what could have been done better throughout the process.

II. Methods

During the ISE 360 - CSC 475/592 lectures we were shown a multitude of resources such as starter codes, a simulator, and ROS. The starter codes were written by Dr. Hamed Saeidi both in C++ and Python. The first important code is the "manual_initialization.py" which is used to set the robot to an initial position by sending the joint motors specific positions. While the code is not necessary for the other codes to run, it is still useful as it starts the robot in the same position every time[2].

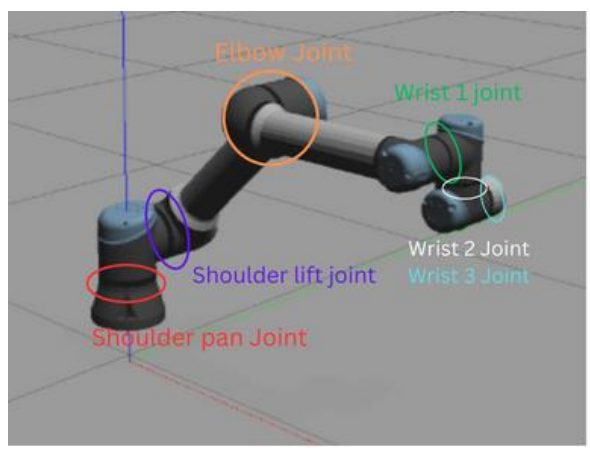

Fig. 2: The joints and their names which are color coordinated. The tip of the robot is designated as the end-effector and is the metal flange after wrist 3 joint

The next important code is the "ur5e_controller.cpp" which allows us to control the robot. It defines a chain of links and joints using the Denavit-Hartenberg (DH) parameters [2]. These parameters essentially tell the code the specifics of the robot based off the base of the robot 'zeroed' position). These are usually given by the company that makes the robot. These parameters allow us to use matrix multiplication to move the robot to desired positions. After defining the DH parameters, the code then goes on to get the initial joint positions, then computes the kinematics (received from "task_space_traj.cpp") of the desired position, and finally publishes control commands to the robot in order to achieve the desired end-effector position (the piece where the tool is mounted) [2].

The next code that is important is the "task_space_traj.cpp". This code is used to receive a set of planned coordinates and orchestrate a trajectory by planning the steps of the trajectory [2]. It then sends this planned trajectory to the "ur5e_controller.cpp" [2]. This code also ensures that the UR5E moves in a smooth and controlled manor by using the "Reflexxes Motion Library" which is an opensource software [3].

The last important starter code is crop_visualize _3D.cpp. This code takes an input image, point cloud data from the camera, and sphere parameters, then processes them to crop point clouds and visual spheres. Lastly it publishes these results to be used elsewhere [4].

The next code is the starter code that we were given called "simple_planner.py" which I renamed to "point_planner.py". We were given this code to alter in order to make the robot function. It creates a "Plan" topic which allows us to plan points by using the following notation: "plan_point#.linear.A = #". The first # symbol is the point number which ranges 1-7 for this specific case since we are only trying to move the end-effector through 7 points with the second point being the location of the ball. The "A" represents the coordinate we are using, cartesian and polar, such as x, y, z, roll, yaw, pitch. The second # represents the specific coordinate we want the robot to reach for or what angle we want to achieve with specific wrist joints. The code uses one of these per for each "A" value available [2]. In the final versions of this code I used the balls position and manual initialization position as the base of many of the points knowing that these would be safe spots to move through to complete the objective

Next, I had to develop a program named sphere_fit.py which uses the crop_visulaize_3d.cpp program to get points of a detected object, which in this case is a tennis ball. The program then fit a sphere to these points and applies a low-pass filter to them. The low pass filter is used to filter out any noise picked up by the camera so that the program sends more reliable points that aren't jumping around. Lastly this program publishes the parameters of this sphere back to crop_visualize_3d.cpp via the /sphere_params node.

The next node that I developed for this problem is named detect_ball.py. This program gets a raw 3d image from the camera and processes. When processing the video, the program uses OpenCV to look for specific hues that the camera is seeing. For this problem the program picks up the neon green/yellow of the ball and highlights the ball white and turns everything else black. It publishes this altered video to the node /ball_2d.

The first tool we were given is a simulator named Gazebo. This simulator allows us to test the codes and see how the robot reacts which will be useful for ensuring that we don't break the real one when we eventually get to it.

The next Tool we were given was Rviz. This allowed us to see the video of the ball and watch the simulated robot tip move with respect to the ball.

The last tool we were given is integrated in ROS as a program called "rqt_gui" this allows us to manually send the robot position commands. This was useful to me as it was the first tool that I used to solve this problem.

This project was split into two parts. First, we were tasked with making the robot move through four points in a simulator. For the second part, we had to detect the ball and move the robot to pick up the ball and set it somewhere else. The following is the process of how each of these parts were solved.

Part I:

I started solving this problem by finding the points that I wanted to send the end-effector to by using "rqt_gui" to do trial and error in order to understand how each value affected the robot's movement. After figuring out each of the desired points/ coordinates, I used the "point_planner.py" to send the four points to "task_space_traj.cpp". After interpreting the plan point, it then sent instructions to the "ur5e_controller" which in turn controlled the simulated robot and moved the end-effector to the specified points. The interaction of these modules can be summed by the nodes circled in red in (Fig. 4).

Part II:

I started solving this problem by creating the ball_detect.py program using Rviz to check if the ball was being highlighted and detected properly. There was a lot of fine tuning in this step, but I didn't fine tune it well enough which caused problems down the road. This will be discussed later in the paper. After that I then developed the sphere_fit.py program to which receives the point cloud data from crop_visualize _3D.cpp and fits a sphere to the detected part that the camera can see using the least squares method. This is a mathematical method that allows gets all the point cloud data and hones in a sphere that is size of the ball. In this case we set up matrices that resemble the equations needed to define a sphere and then it use this method to solve the equations. This gets us the coordinates of the center of the ball with respect to the camera and its radius.

Next, I added low pass filtering to this program in order to control the noise that the camera was receiving. In short, this is another mathematical method that gets averages of the position of the points where the weight of newly received versus old points can be adjusted to help make the points steady and settle down to a defined point. The points were then published to /sphere_params. This result can be seen in (Fig. 3)

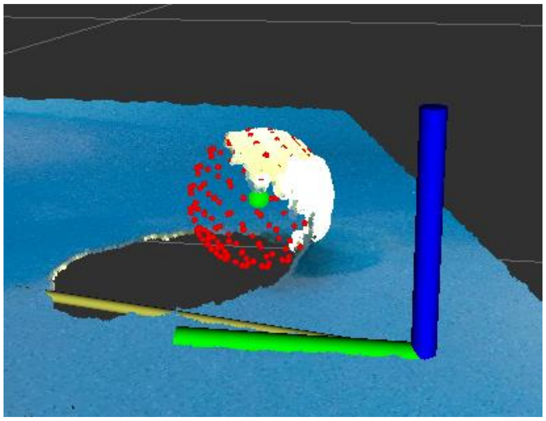

Fig. 3: The detected ball as seen in Rviz with points created by sphere_fit.py

The next thing I did was integrate this into the point planner program. Using a ros method called tf2. I was able to get the points given by /sphere_params and location of the base of the robot with respect to the camera. This allowed me to find where the ball was located with respect to the base. I then made the balls location as point two in the point_planner.py program to pick up the ball. The interaction of these programs can be summed by the nodes circled in blue in (Fig. 4).

Finally it was time to test this code on the real robot. Before doing this though I had to change a couple basic things. First, I integrated a timeout function into shpere_fit.py to stop data collection after a certain period so that when the points converge to the balls location the program sends one solid coordinate to the robot so that the robot doesn't get confused with the small osscilations of the fitting module. Then I had to integrate the grasper code that Dr. Saeidi had made. I had to add a grasper mode point to each point in the planner with a specific integer to tell the grasper what state it needed to be in. 0 was to stay in the same mode it currenlty is in, 1 was to open the grasper, and 2 was to shut the grasper.

After these changes were made, I went to the lab where our TA, Andrew, checked my codes to ensure that I wasn't going to send the robot through the wall or table. We had to make one small change which was to add a rqt_gui boolean value. This allowed to check the points of the plan and then click a box in the rqt_gui to publish the plan so that the robot could move.

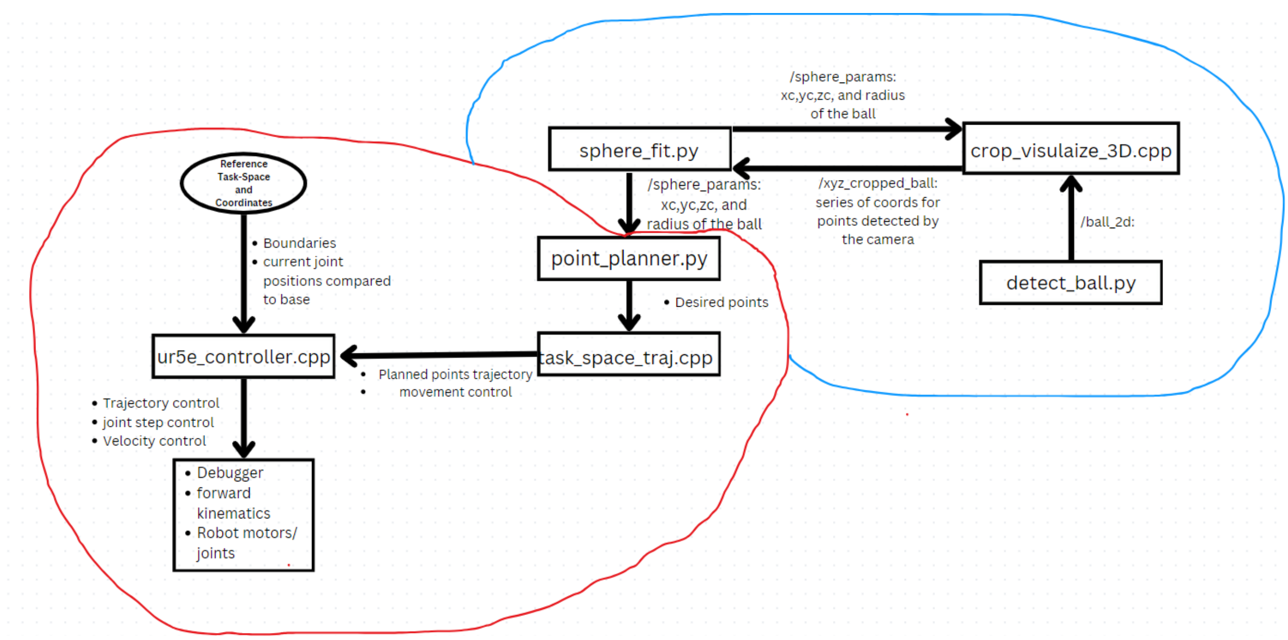

Fig. 4: A diagram of the work flow and how all of the codes are connected red indicates phase 1 implementation and blue indicates phase 2 implementation.

III. Discussions and Conclusions

Throughout the process there were a couple of hiccups. The first problem was that the simulator would crash any time I sent a coordinate that was out of reach for the robot. After the first time and taking 10 minutes to re-initialize the program, I decided it would be smarter to take baby steps from known positions so that the program wouldn't crash again. The second problem was that I didn't realize that I needed to make the point_planner.py in a ROS package so that it could be ran properly with all of the other modules. I had realized my mistake as soon as I ran it the first time and this was a quick fix. I used the "catkin_create_pkg" and made the same messages that my professor used in the starter file.

The third problem was during sphere fitting and was noticed when applying the low pass filter. When applied, the low pass filter should slowly settle on an xyz points which would eliminate noise but this wasn't happening. After troubleshooting and double checking the math, I realized that there was a white blob off to the side of the ball. I then adjusted the detect_ball.py to change the hues that were being detected until the blob was gone. The last big problem I ran into was also while sphere fitting. Every once and a while the program would crash after it was sent a bad array. I solved this by setting up an exception handler to skip that array and keep processing the points. All in all, the project was successful and I was able to get the robotic arm to smoothly to the position of the ball and then move to a new location.

References

[1] H. Saeidi, "ISE 360-CSC 475/592: Robotics Reports and Presentation for Phase: Robot Modeling and Control"

[2] H, Saeidi (2022) ur5e_control [Source code]. https://github.com/hsaeidi-uncw/ur5e_control.git

[3] Reflexxes Motion Libraries: The Reflexxes Motion Libraries – An Overview. Reflexxes.ws. http://reflexxes.ws/software/typeiirml/v1.2.6/docs/page__reflexxes_motion_libraries.html

[4] H. Saeidi (2022) robot_vision_lectures [Source code] https://github.com/hsaeidi-uncw/robot_vision_lectures.git