Collision Avoidance System (CAS) on the Triton Robot

Lingxi Li [1,◆,◇] Andrew Meyer [2,◆,◇] Sahan Prasad Podduturi Reddy [3,◆,◇]

***Abstract*— The primary motivation behind this project was to design a robot that can move from a given start position to the destination area whilst avoiding all incoming moving obstacles. This is an important challenge because mastering this task is essential in real-world applications such as effective autonomous navigation. The main problem being addressed here is how to receive available sensor data and use it to accurately detect moving obstacles in an environment so that measures can be taken to avoid those obstacles. This is necessary for the goal of obstacle avoidance in dynamic environments which is crucial in the task of autonomous navigation and also helps facilitate independent robots working in the future. The main approach that we have come up with to tackle this problem is to first use data received via the camera equipped on our Triton robot to detect moving obstacles in the environment and then calculate the coordinates of each moving obstacle. The YOLOv8 object detection algorithm helps us in this process as it assigns bounding boxes to objects detected and calculates depth distances relative to the robot's location. We then pass the resulting coordinates to the Kalman Filter algorithm to effectively estimate the position and velocity of moving obstacles in real-time to help predict the future positions of the obstacles, enabling the robot to plan its path and avoid potential collisions.**

I. INTRODUCTION

A. Background

Autonomous navigation is one of the most fundamental tasks in Robotics that hasn't been fully tackled yet. The ability of a machine to safely navigate through a dynamic environment where changes are introduced at random points in time is very important if we are ever to go into the age of fully autonomous vehicles. By designing a robot that can move independently from a given starting position to an end position whilst avoiding obstacles in its path, we are promoting progress in robotics applications such as fully autonomous service robots that can be deployed without any human supervision. The purpose of this project is to design a robot capable of such navigation as it moves in a straight line from a given starting position to the destination location. There will be moving balls spawned at random that will be rolled in the robot's direction and the robot must try to avoid these balls as it makes its way towards the destination location. The robot will not have any odometry values to leverage while it is in the environment and it must rely solely on the sensor data available to it to successfully navigate to the end location. Successful demonstration of this task will help to demonstrate the potential of combining advanced sensing, computer vision, and estimation techniques for autonomous and safe navigation in real-world scenarios.

B. Specifications

Our robot is equipped with the Intel® RealSense™ Depth Camera D435i which plays a vital role in achieving our goal. The camera provides depth information that allows our robot to perceive the environment in 3 dimensions. This was extremely beneficial because we were able to estimate the Pose data of obstacles moving in the robot's direction directly by processing camera data and we do not have to depend on other sensor data for our calculations.

We also leveraged the YOLO algorithm in order to detect any objects that appear in front of our robot. YOLO is able to detect multiple objects at once from an image and classify them based on the dataset the model was trained on. Ultralytics provides pre-trained models which are trained to classify balls that are present in an image. This turned out very handy for us as we can utilize this model ('yolov8n.pt') to detect the moving balls in our environment and assign bounding boxes to each ball that is recognized by the YOLO model at a particular time step. Thus, the depth information along with the YOLO algorithm allows for efficient and accurate object detection and pose estimation.

The camera data gives us an estimate of the current location of the ball at a particular time step, we can further refine our obstacle-avoiding capabilities by using the Kalman Filter algorithm. This algorithm takes in the pose data which we calculated by utilizing the YOLO algorithm at an earlier time step to estimate the velocity and movements of the obstacles in real-time. By knowing this information, the robot can adjust its path accordingly avoiding collisions with any moving obstacles that may spawn in the environment.

C. Environment

As in our proposal, we defined our experimental environment as Figure 1. The world should have a starting area and a destination area virtually, which has at least 5 unit distance in between (such that 5 meters in real world or 5 unit distances in Gazebo Simulator). The objective of the task we defined is allowing robot to avoid colliding with red balls while moving towards the destination area. We are working on a

The triton robot can only move in three ways:

- Moving forward, which is having a fixed y velocity and no x velocity.

◆ University of Massachusetts Amherst
◇ Equal contributions
[1] lingxili@umass.edu
[2] almeyer@umass.edu
[3] spodduturire@umass.edu

♠ The GitHub repository of our catkin workspace is publicly available at: https://github.com/lilingxi01/cas_ws

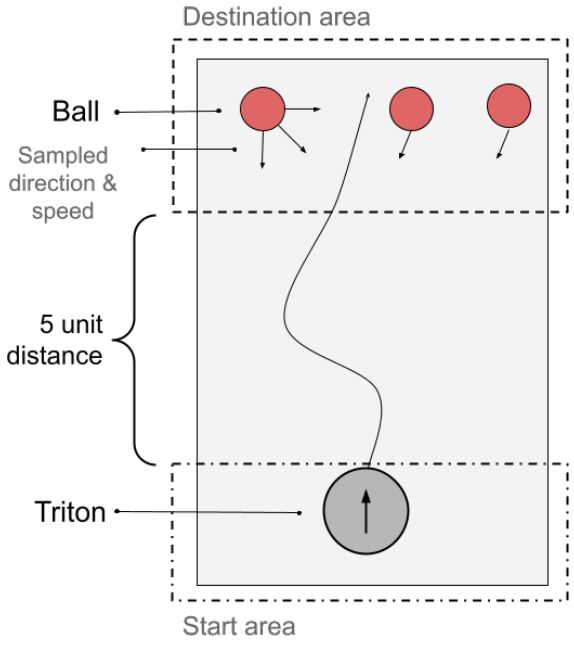

Fig. 1: **Environment Definition.** This is our proposed evaluation environment of CAS. We will also build our simulation platform based on this environment definition.

- Moving forward with horizontally shifting to the left, which is having a fixed y velocity and a negative x velocity.
- Moving forward with horizontally shifting to the right, which is having a fixed y velocity and a positive x velocity.

II. APPROACH

A. Collision Detection Logic

Once we have identified the ball the next step is to track it and predict the movement. For this it is important to have a lightweight algorithm that can process in real time. We used tested two different approaches to see which would work better. First, we thought a Kalman filter would be a good algorithm for this task. This is a common use for the Kalman filter going back all the way to its first major use in the Apollo missions [1]. The Kalman filter [1] is a type of Bayes Filter in that it starts with a prediction step then has an update step. We were also able to find a python demonstration of its tracking and prediction abilities with a thrown ball [2]. We used this source code to implement our Kalman filter. The Kalman filter uses a two step process. First, the algorithm predicts the future state based on a physical model of the system, and second it updates it based on sensor data. For our project the prediction step is based on a velocity model of the ball and the update is based on the coordinate readings received from the camera. By using the predictions for the update step it is able to predict more time steps into the future. This is shown in 2, where the kalman filter can accurately predict the position of the ball many steps in the future.

[1] **Kalman Filter.** The link shows our experimental work to get the Kalman Filter working - `https://t.ly/9cBbm`

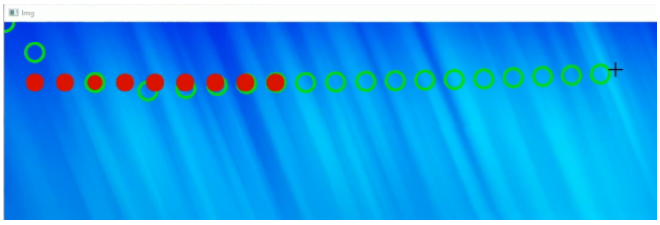

Fig. 2: **Kalman Filter Used for Path Prediction** The red circles are the actual balls positions and the green rings are the predicted positions.

The other method we looked into was using a more specialized model of the ball movement for this scenario. Assuming that the robot is moving at a constant velocity and the ball is approaching at an approximately constant velocity then the motion of the ball relative to the robot could be modeled as a straight line. In this case by knowing two positions of the ball a model of the motion could be predicted as a straight line. So we assumed that the robot's coordinates were at (0,0) then we can find the shortest distance between the line and the origin. If the distance is less than the size of the robot itself plus some buffer area then the robot can make a decision to avoid that collision. There is an equation to use two points of a line and find the closest distance from that line and another point which is shown in equation (1). We did not end up using this technique because it would not be as transferable to other scenarios.

$$Distance = \left(\frac{|(x_2 - x_1)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_1)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \right) \tag{1}$$

B. Simulator

We mainly experiment our architecture and algorithm on Gazebo Simulator with a customized world (map), a modified triton robot model, and a custom-built ball shooting script. The objective of our simulation platform (as known as Collision Avoidance System Simulator, or CAS Simulator) is to mimic the environment we defined in our proposal, including dynamics of balls, dynamics of triton robot, and possible collisions between balls and triton robot. To achieve this, we have taken a lot of considerations into account.

1) Environment Model and Robot Model: We built the world model of CAS Simulator using a few built-in cubes in Gazebo GUI as shown in Figure 4. To avoid unexpected trace of balls in a simulated environment which is bounded by boxes, their physical properties like mass, frictions, and restitution coefficient are changed to allow balls sliding along the wall rather than bouncing off from it. We have considered the difficulty of this simulation environment, so we should eliminate experimental noises like such bouncing balls. Furthermore, we realized that physical interactions

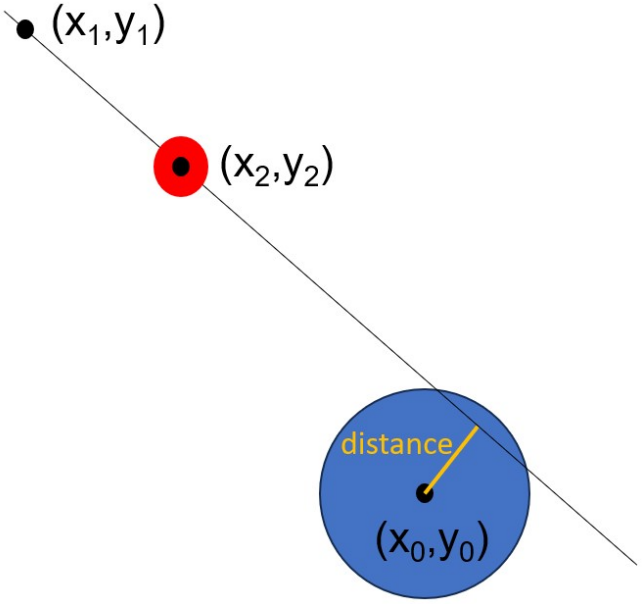

Fig. 3: **Object Prediction Using Geometric Model.** Using the balls current position and previous position a predicted path of the ball can be created and used to anticipate collisions.

among balls and between balls and triton robot are significant under our environment setup, so we have kept the bounce behaviors among triton robots and generated balls. However, some balls shooting directly towards the triton robot might stuck to the 3D model of camera module, which blocks the camera vision stream and interferes our experiment severely. Therefore, CAS Simulator has employed a collision detection mechanism that deletes the 3D model of a ball when it hits camera and has possibilities to get stuck.

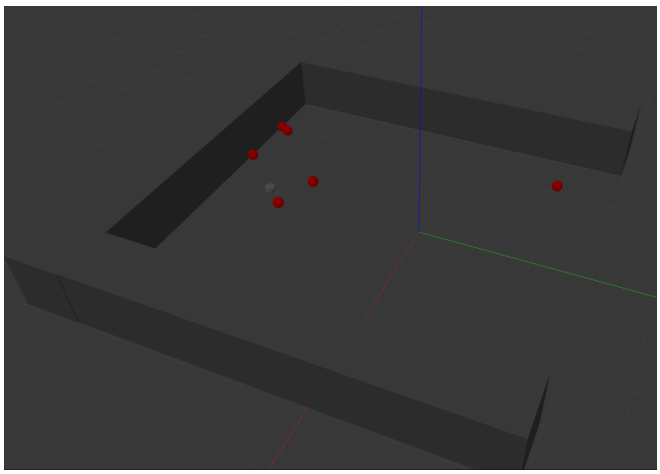

Fig. 4: **Simulation Environment.** Custom made Simulation Environment with Ball shooter

2) Ball Shooting Behavior: In order to create enough of task complexities and enough degree of reality, we made the ball shooting program as a separate script running exclusively on CAS Simulator and does not share any information with our CAS Core package. Once CAS Simulator receives the `True` from `/cas_sim/spawn_balls` topic, balls will be spawned automatically with a sampled position S, as showing in equation (2), and applied a sampled force F on balls for shooting towards the robot, as showing in equation (6).

$$S = \left(S_x \sim \mathcal{N}(0, 2), S_y = 2, S_z = 0\right) \tag{2}$$

$$D_{adv} = \max\left(0, (2.0 - X_y) \cdot 0.2\right) + 0.4 \tag{3}$$

$$\theta = \text{atan2}(X_y + D_{adv} - S_y, X_x - S_x) \tag{4}$$

$$\widehat{\theta} \sim \mathcal{N}(\theta, 2^\circ) \tag{5}$$

$$F = \left(F_x = \cos(\widehat{\theta}), F_y = \sin(\widehat{\theta}), F_z = 0\right) \tag{6}$$

This algorithm has consist of a series of factors:

- X is the true position of the triton robot provided by Gazebo super privilege data. The true location of triton robot is only accessible on CAS Simulator by design and will never be seen in CAS Core to avoid cheating on math models.
- D_{adv} is a value used to point the angle to some degree ahead of the current robot position, as defined in equation (3).
- θ is the angle from spawning position S pointing towards the triton robot, as computed in equation (4).
- $\widehat{\theta}$ is a sampled theta which adds randomness to the final shooting angle, as shown in equation (5).

3) Re-usability: For easier replication and better re-usability, we have modularized the code for CAS Simulator into a separate package other than CAS Core, relying on `stingray_sim` package for providing the triton robot model.

4) Evaluation over CAS Simulator: As shown in Figure 7, the actual screenshot of CAS Simulator in a bird-view angle, balls are all shooting towards the triton robot along with marked arrows. We rely on this system for our evaluation because the randomness in ball shooting provides great diversity over possible situations, where in this graph, the left most ball is shooting directly towards the triton robot and the other two balls will just pass by it. This randomness can be used to evaluate our system (CAS) on both true negatives and false positives.

C. Object Relative Pose Estimation

Our robot is currently equipped with the Intel® RealSense™ Depth Camera D435i. This gives us the added advantage of calculating the depths of moving objects that are detected by the robot instead of having to depend on LiDar to solve our problem. When we load up our simulation environment with the Triton robot, there are many ROS camera topics that are made available to us which we can subscribe to. We are mainly interested in two of those topics - `'/camera/color/image_raw'` which gives us a color image of the camera data and `'/camera/depth/image_raw'` which gives us a

grayscale representation of the camera data. We are leveraging YOLOv8 which is the latest version of the popular object detection algorithm that is available and known for its real-time object detection capabilities and its ability to detect multiple objects in an image or video simultaneously. There are YOLO models available through the Ultralytics library that are pretrained on a dataset to detect and classify a set of objects that might show up in the image data. We utilize one of these YOLO models - `yolov8n.pt` which helps to detect the moving obstacles in our environment and creates bounding boxes for each obstacle.

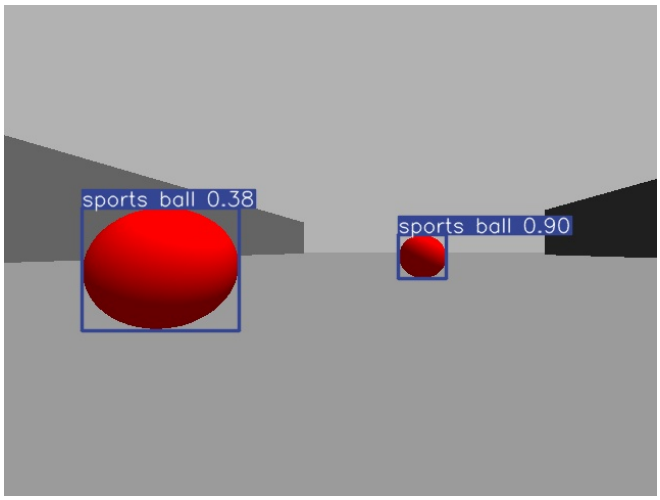

Fig. 5: **Ball Detection.** YOLOv8 detects objects from camera data, assigns labels to each object with a confidence score, and creates bounding boxes for each object.

In order to use the YOLOv8 model, we need to retrieve data from the `'camera/color/image_raw'` topic and transform it into a suitable format. We leverage the OpenCV library's `imgmsg_to_cv2()` function which helps us transform images of type - `'sensor_msgs.msg'` into a Numpy array representing the same image in OpenCV format. This image can then be passed to our YOLO model. The YOLO model creates bounding boxes for every obstacle that is detected and the box information can be retrieved from the model output. We then calculate pixel values corresponding to the center point of every obstacle detected.

We then use the output of the `'camera/depth/image_raw'` to compute the depth measurements by utilizing the obstacle center points that we calculated. We do this by first computing a depth matrix from the topic data which contains depth values corresponding to every pixel of the image outputted by the camera data and then looking up this depth matrix by using the obstacle center points previously calculated. The depth value of an object tells us how far the object is from our robot.

Now that we know the distance of obstacles from our robot, we need to find the angle relative to the center of the robot at which the object is located. This will help us because if we project out a line to the center point of any moving obstacle, and a second perpendicular line from the center of our robot such that they form a right-angle triangle, then we can calculate the (x,y) coordinates of the object relative to the robot using the Law of Sines.

$$\left(\frac{\sin(A)}{a}\right) = \left(\frac{\sin(B)}{b}\right) = \left(\frac{\sin(C)}{c}\right) \tag{7}$$

In order to find out the angle relative to the center of the robot, we needed to first figure out the angle width of the total field of view of the camera equipped on the robot. Not trial and error was required here as the website had the camera width specification at 87°. We always assume our robot is at point (0, 0) and facing in the 0° direction at the start. Since we have pixel data corresponding to the (x,y) values of the center of an obstacle in the environment, we need to scale down the values to be in the range [-43.5, 43.5] to get the angle in degrees.

$$\left(\frac{x}{640} \times 87\right) - 43.5 \tag{8}$$

We can then use this angle to calculate the other two sides of the formed right-angle triangle using the Law of Sines. The values of these two sides will correspond with the x and y values of the obstacles whose pose we are trying to estimate. This data is then passed down to the Kalman Filter to estimate the motion of the obstacles.

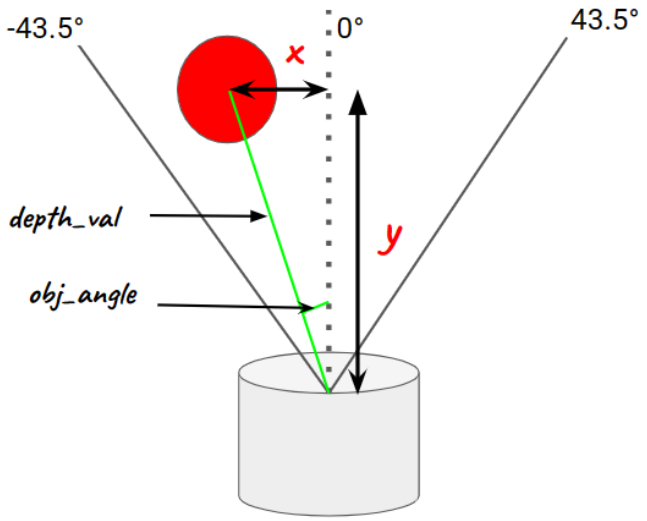

Fig. 6: **Pose Estimation Logic.** Depth value is retrieved from the depth matrix using pixel coordinates and the angle from the center of the robot to the center of the object is calculated using the above formula. Then, we can estimate the pose of the object (x, y) leveraging Law of Sines.

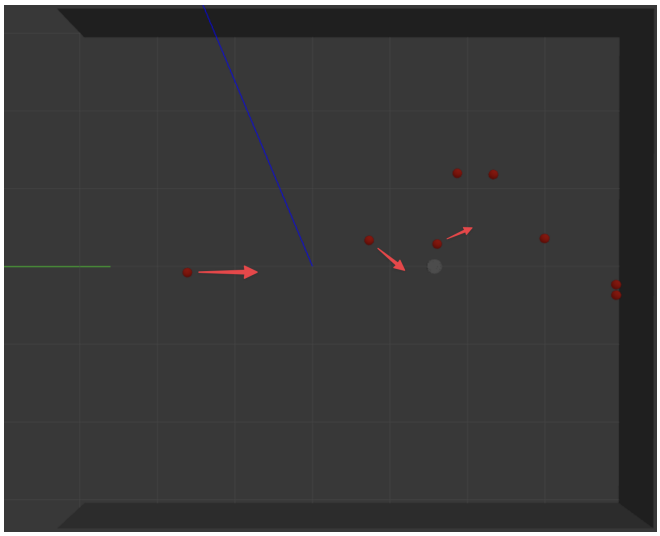

Fig. 7: **Ball Shooting Logic.** Balls are shooting towards the robot with a gaussian distribution for randomness and a dynamic lead calculation to aim the target position of triton robot at time $t + i$ where i is based on the distance from the shooting area to the robot.

D. Frame Mapping

Since we are trying to calculate the object pose at different time steps which gets passed on to the Kalman Filter, we have to find a way to map objects from previous time steps in the previous frame to their current location in the current frame. In order to do this, we compare poses calculated for obstacles in the previous timestep with those calculated in the current time step. For every pose reading calculated in the previous time step, we calculate Euclidean distance to pose estimate values acquired from obstacles in the current timestep. Obstacles with the minimum Euclidean distance between their pose measurements are mapped together. We also maintain a minimum distance threshold so that newly spawned obstacles do not get mapped to obstacles from older timesteps accidentally. [2]

III. EXPERIMENTS

A. Camera Data

To create the depth matrix for our model to calculate the center points of obstacles that the robot perceives, we needed to figure out a way to process the data from the `'camera/depth/image_raw'` so that we can directly lookup depth values based on pixel values corresponding to obstacle's center points. The data is originally in the form of a very long list of values (length - 1228800) which are converted into a Numpy 3D array corresponding to the depth values for every one of the pixels in the 640*480 pixels image. The third dimension will be '4'. Each pixel of the image has 4 values associated with it which represent the depth. We tried different methods to interpret this data and retrieve the correct object depth:

1) We first tried splitting up the Numpy 3D array into 4 separate channels where each channel represents one of the 4 depth values corresponding to the pixels in the image to get a better picture of what the values represent. We tried plotting each of the 4 channels individually to see whether any of them directly represent accurate depth values. If we plot the depth matrix, we expect a grayscale image with areas further away to be lighter in color and areas closer to the robot to be darker such that every pixel representing a particular depth value has the same shade as other pixels with the same depth value.

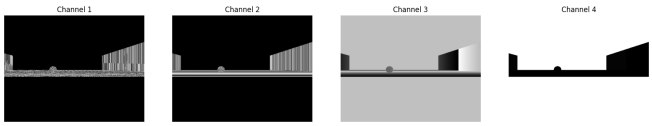

Fig. 8: **Depth Channels.** Each graph depicts one of the 4 channels representing the depth values for each of the pixels.

None of the graphs depict the actual depth matrix that we perform further analyis on the data.

2) Later we tried calculating the mean of the 4 channels which gives us a single Numpy 2D array containing depth values for each pixel of the image data we receive from the `'camera/depth/image_raw'` topic. We tried sampling depth values based on obstacle center coordinates from the resultant matrix but quickly realized that the values are much larger than expected.

3) Finally, we tried converting the original `depth_image` array into a new array that contains depth values as 32-bit floating-point numbers. The conversion involves reshaping the original array into a 1D array of bytes (`depth_image_bytes`) and then interpreting those bytes as float values. Finally, the resulting array is reshaped back into a 2-dimensional array representing the depth map with the same dimensions as the original array. We then compare the resulting graph.

We tried retrieving depth values from this matrix based on obstacle coordinates and notice that the values are accurate. We use this matrix as the final depth matrix.

B. Limitations

While we were able to get a working demonstration of the collision avoidance system in the simulator, we had difficulties with the processing speed. The biggest issue we faced was with the perception problem. In our case that was difficulty detecting the red balls. The detection was unreliable and typically could not detect the balls until they were already too close to the camera. This difficulty was most likely due to the lack of GPU speed when we were running the simulation. In order to counteract this

[2] **Frame Mapping.** The link shows our experimental work to successfully enable frame mapping - `https://t.ly/ZdXd`

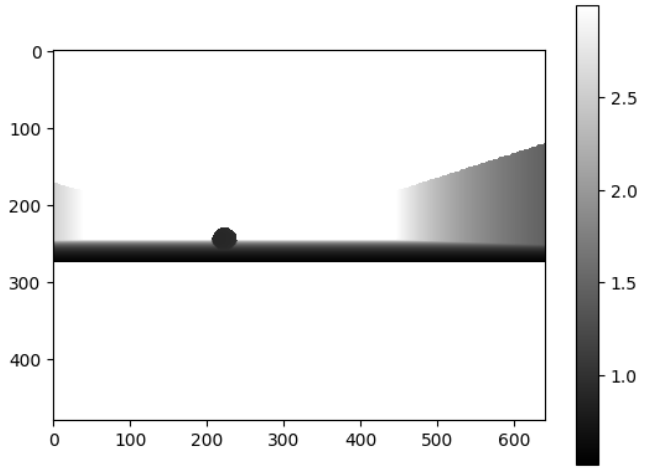

Fig. 9: **Depth Matrix.** Plot of the Depth Matrix.

problem we slowed the robot and balls down drastically. This wouldn't be realistic in the real world, but was a good proof of concept. To compound this the simulation environment was noisy and YOLO algorithm would sometimes identify the walls as objects.

The other biggest limitation we had was being unable to run the system on the real Triton robot. With the time constraints we had with this project we wanted to focus on getting the CAS system working and by the time we did we didn't have time to test on the real robot.

IV. CONCLUSION

This project helped expand our knowledge of ROS and work on a more holistic robotics project than our previous work in this class. We developed a system that can take in the camera data, process it in order to identify an object, and calculate the predicted future positions of that ball. We were not able to get the system working on the real Triton robot, but we developed a realistic simulation in order to more easily test and iterate on the code. We learned how to design a robotic system from top to bottom. We had to think about all of the three major aspects of robotics: perception, cognition, and action. Our project also opened our eyes to the difficulty of navigation in a dynamic environment. We used a very simple example due to the time constraints of the project. Our robot was moving forward at a constant velocity, and the ball follows a straight path as well. If we had more time on this project we have some future goals we would focus on. The main goal would be to try the CAS system on the real Triton robot. This would require some calibration and fine tuning to get to work in the real environment, especially concerning the camera data in a much noisier environment. A few longer term goals would be to rework the system so it could drive in any direction at any speed. We would need to change how we get the coordinates of the balls. Another big step would be to be able to avoid objects that can change speed and direction, like a human or another robot. This project has given us a holistic view on what it takes to design a robotic system using ROS.

REFERENCES

[1] McGee, Leonard A., and Stanley F. Schmidt. Discovery of the Kalman Filter as a Practical Tool for Aerospace and Industry. National Aeronautics and Space Administration, Ames Research Center, 1985.

[2] Canu, Sergio. "Kalman Filter, Predict the Trajectory of an Object." Pysource, 2 Nov. 2021, pysource.com/2021/11/02/kalman-filter-predict-the-trajectory-of-an-object/.

[3] Mathew, J. (2022, May 2). Estimating depth for Yolov5 object detection bounding boxes using Intel® realsenseTM depth camera... Medium. https://medium.com/@jithin8mathew/estimating-depth-for-yolov5-object-detection-bounding-boxes-using-intel-realsense-depth-camera-a0be955e579a

[4] Davies, D. (2021, September 20). Yolov5 object detection on windows (step-by-step tutorial). W&B. https://wandb.ai/onlineinference/YOLO/reports/YOLOv5-Object-Detection-on-Windows-Step-By-Step-Tutorial—VmlldzoxMDQwNzk4

APPENDIX

A. *Workloads*

- Lingxi (33.33%): Built CAS Simulator and launch files; wrote frame mapping component (`frame.py`); designed relative position mapping with Sahan; fine-tuned Kalman filter function with Andrew; finished hacking version for demo; and done final integration of different components.
- Andrew (33.33%): Implemented Kalman Filter, Worked on Robot Decision making, Contributed to writing report and presentation
- Sahan (33.33%): Implemented YOLOv8 Algorithm. Contributions to writing the project report. Implemented Pose Estimation using Intel RealSense Depth Camera data.