Robot Goalkeeper

Adam Dziedzic, Ryan Wunderly, Jonas Hirshland, Jingliang Ren

***Abstract*—Our team created an autonomous robot that guards the area behind it, moving to intercept balls that are rolled towards its range and are within the view of its vision system. We successfully combined sensing, reasoning, and acting to make a functional autonomous robot, demonstrating what behaviors and skills are needed for goal keeping.**

I. Introduction

The most impressive robots can interact with a human and educate participants on how world knowledge is created and used. For our project we wanted at the very minimum to foster an interaction between a robot and an audience member at the expo. One of the most basic interactions, which even a young human can participate in, is passing a ball. We decided to keep the interaction on the ground because a two-dimensional space was easier to handle and make a plan for.

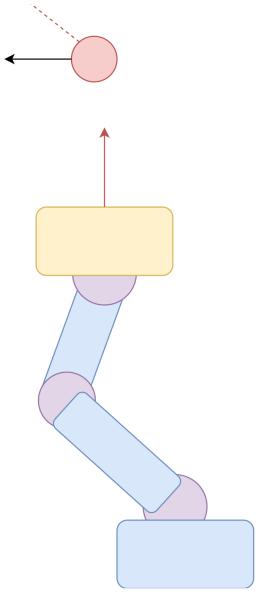


Fig. 1: Our robot arm has 3 important joints and sense balls in the playing field to determine their velocities and projected positions.

There are many ways to complete a pass of a ball, but all of them require a reliable vision system, a way to interpret ball detections, and calculate a plan. We completed the full integration of all of these systems and successfully blocked balls that were coming at the robot from a human.

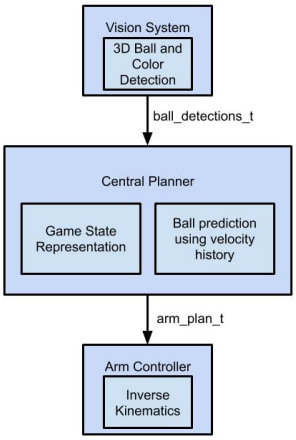


Fig. 3: The overview of the complete control system, which includes all components i.e. vision system, central planner and arm controller.

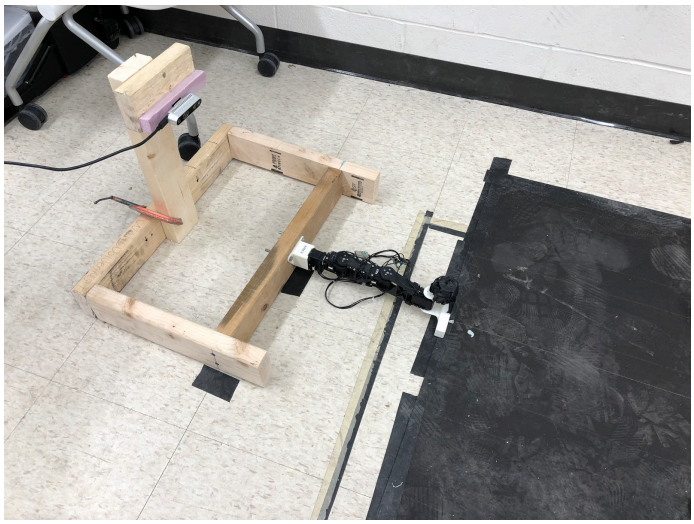


Fig. 2: What our robot ended up looking like.

II. Methodology

The robot goalkeeper needed develop a model for the state of the world, and plan actions to interact with that state. We divided the robot's capabilities into the vision system (sensing), the central planner (reasoning), and the arm controller (acting). In this section we will discuss how we approached our solutions.

- ***Arm Design*** The robot uses Dyanmixel servos and 3D printed parts to .
- ***Vision System*** Uses RGB images and point clouds to detect balls and send them to the central planner.
- ***Central Planner*** Central planner keeps the world representation of balls, their coordinate history, and their trajectories. It predicts the motion of the balls and sends motion plans to arm controller to intercept the ball.
- ***Arm Controller*** Calculates Inverse Kinematics to execute plans and control servos.

A. Vision System

We use Intel Realsense D435 RGBD cameras for RGB and depth images as well as point clouds of the environment. We chose to use these cameras because they solve the 3D vision problem for us, since it performs stereo vision under the hood in the hardware, we don't have to do it in software. Thus cutting out latency and work for our project, and making it more reliable and easy to set up. In addition, the camera was able to function at both 30 and 60 frames per second at a resolution of 640 by 480 for both RGB and depth, which are the settings we operated our vision system at. Our system has been tested to work with up to 3 Intel Realsense D435 cameras, but could theoretically be extended until either the processor or bandwidth is saturated, since each camera works in an independent thread of control.

We needed to aid the Intel Realsense cameras with asphalt felt because we ran tests by pointing the camera at the ground in the EECS 467 lab in the CSRB and Tishman hall and found that the tiles are effectively invisible in the depth image. The depth felt however shows up perfectly in the depth image. We need a proper ground plane for calibration and to have a correct mask in our ball detection, therefore, we lay down asphalt felt in front of our robot in these environments.

1) Ball Detection:

1) Compute mapped and unmapped Point Clouds
2) Convert Point Cloud from Camera to World Space
3) Convert RGB image to HSV image
4) Mask HSV image using x, y, and z of points
5) Color Segmentation of HSV Image
6) Extract point clouds for each contiguous color segment
7) Perform RANSAC sphere fitting on point clouds to purge false positives
8) Return point cloud centroids labeled by color as detections

We use the intrinsics and extrinsics of the cameras provided by realsense to compute a point cloud for each depth frame and map the points to pixels on the color image. We first tried using the pointcloud class provided by librealsense, however, the time it took to compute a point cloud mapped to the color image using their black box took 35 milliseconds, whereas when we did it manually it took less than 5 milliseconds, therefore, we computed the point cloud and mappings manually.

The mapped and unmapped point clouds are then transformed from the camera coordinate frame to the arm coordinate frame using extrinsic parameters that we calibrated. The RGB image from the camera is converted to HSV as well.

Fig. 4: Original image with ball detections drawn on.

The next step is a masking of the HSV image using the mapped point cloud, where only the pixels corresponding to points within a specified rectangular volume in the arm coordinate frame. The volume is configured to mask out all pixels that lie outside of the asphalt felt area and that are too high off the ground as well. Therefore, only the balls would be left. However, due to the depth noise of the Intel Realsense D435 it was impractical to have a tight bound for height, so we had to settle for the asphalt felt bleeding through. This didn't pose an issue due to the asphalt felt's consistent dark black color which does not result in false positives.

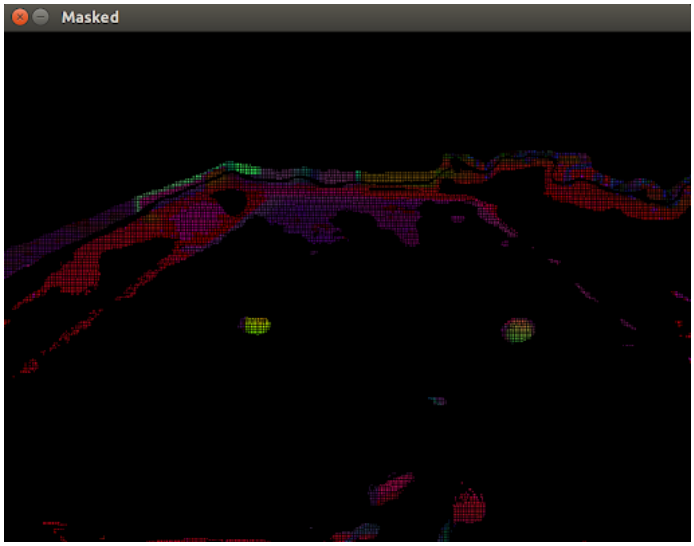


Fig. 5: Masked HSV image using only height based masking.

Next the remaining HSV image is segmented by hue, saturation, and value ranges. Our algorithm creates masks

that contain the areas of bright green, bright blue, and bright orange. Those masks then have a dilation applied using a 3 by 3 kernel. This is because our mapped point clouds have points mapped to every other color pixel because the depth image has a greater field of view than the rgb image. Then a 3 by 3 kernel used in an erosion. This fills in the gaps and allows the removal of noise that follows to work. Next a 5 by 5 kernel erosion is applied to each mask, then a 7 by 7 kernel dilation is applied, followed by a 3 by 3 erosion. The end result of these morphological operations are masks where random noise segments are removed and correct color segments are contiguous and of the correct size. The final results are visible in the mask images below.

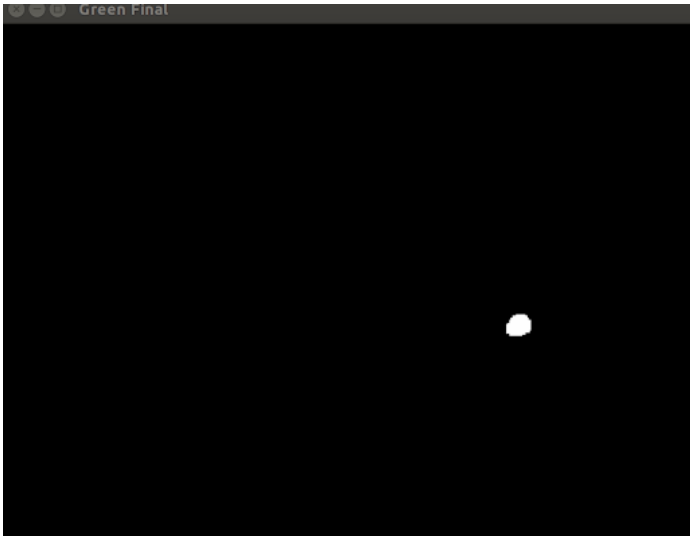


Fig. 6: Green only mask.

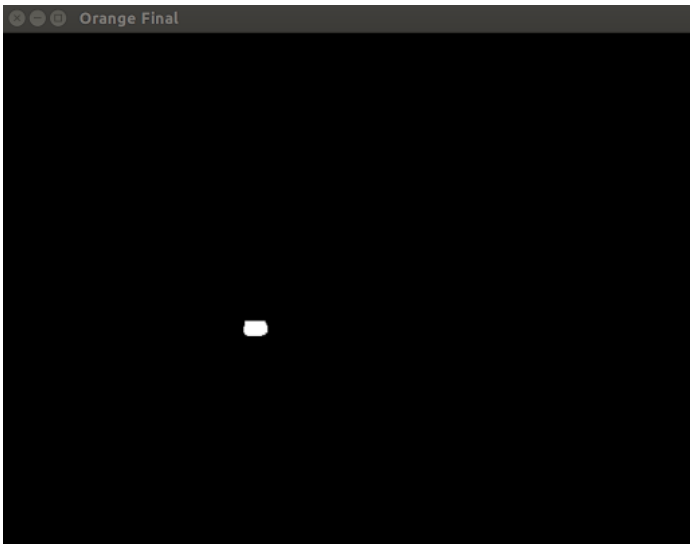


Fig. 7: Orange only mask.

Next each blob in the color segmentation is used to create a pointcloud of the corresponding points which is used in a BallPrototype class. This class represents a potential detection and is made with useful behavior for refining the detections using distance in the world coordinate frame, i.e. properties obtained from the point clouds. The BallPrototypes are combined into new BallPrototypes if they are within a ball radius of another prototype. Then some pruning occurs where the prototype centroids are recomputed and smaller prototypes that are too close to others are removed. The remaining BallPrototypes have their point clouds used for RANSAC sphere fitting from the Point Cloud Library to catch false positives. The BallPrototypes that still remain are converted into detections by taking their color label and world coordinate frame centroid and sending them over LCM to the central planner for tracking.

Fig. 8: The large orange dots are the detections drawn onto the original image from our ball detection visualization.

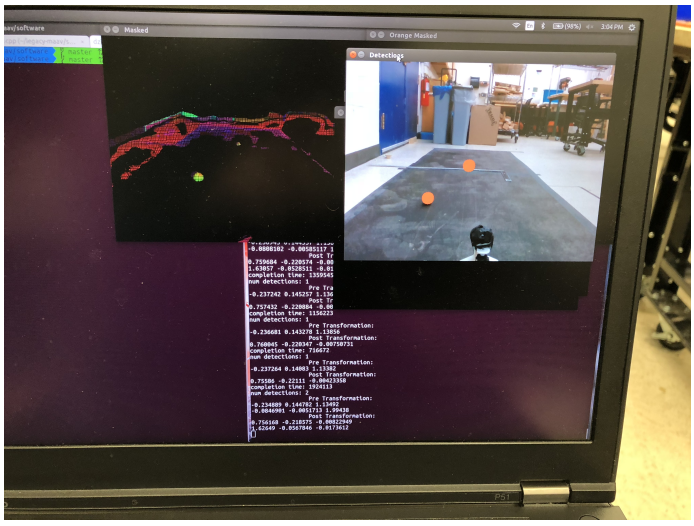


Fig. 9: Our ball detector visualization.

2) Camera Calibration: We created our own custom camera calibration for this project. We use our ball detection

configured to use only height based masking through computing a ground plane as well as the RANSAC plane fitting provided by Point Cloud Library. We first run plane fitting and ball detection until we have reached a minimum of some configured number of detections of ground planes, blue balls, green balls, and orange balls. Usually the configured number ranges from 100 to 250.

We then compute the heights of the camera obtained by computing the distance of the point (0, 0, 0), i.e. the origin of the camera, from the ground planes. We compute the roll and the pitch of the camera using the fitted planes by using the the angles between the normal of the ground plane and the vector $< 0, 0, 1 >$. By projecting the ground plane normal onto the z-y plane, the pitch of the camera is the angle between the two vectors. By projecting the ground plane normal onto the z-x plane, the roll is the angle between the vectors. The roll, pitch, and height measurements are then averaged and outliers are removed from the averaging. The same is done for the detected ball centroids. This reduces the noise in the measurements.

Outliers are rejected by sorting the measurements into clusters where the centroid of the cluster is set, and added points must be less than a set distance, L2 norm, from this point. For points that lie outside of all clusters a new cluster is created. Then the empty clusters are removed. Then the non empty clusters that are too close to larger clusters are pruned. Then the centroid of each cluster is recomputed and points are re-sorted. This runs for 10 iterations. The largest remaining cluster is used to get the averaged measurement.

The difference between the pitch and -PI/2 is how off the camera is from facing directly forward rather than upward or downward. The difference between the the roll and 0 is how far off the camera is from being aligned with the ground plane around the z axis of the camera coordinate frame. Rotation matrices are created that modify the points in the world so that they appear as if the vectors coming out of the sensors of the camera all lie on the same plane that is parallel to the ground plane but not equal to the ground plane. A coordinate frame transform is then applied that makes the previous z axis the new x axis, the previous y axis the new z axis, and the previous x axis the new y axis, putting points into the world coordinate frame with uncorrected rotation around the new z axis.

Next the yaw needs to be computed. The averaged locations of the balls using the previously created rotation matrices are used for this. The vector from the orange to green ball detection is computed. The angle between this vector and the positive y axis is the yaw, i.e. the rotation about the new z axis. A rotation matrix that corrects this is created and applied to the ball detections.

Then the offset between the true location of the balls and the detected positions of the balls are averaged to compute the new x and new y displacements of the camera. The same is done for the heights of the balls to compute the height translation to apply. This is better than using the height at point (0, 0, 0) in the camera frame from the ground plane, because it results in a better height in the area that matters.

Then the rotation matrix is computed as follows:

$$rotation = yaw\_correction * coordinate\_transform * pitch\_correction * roll\_correction$$

The translations to be applied are just the averaged differences between the ground truth ball locations and detected ball locations.

```
roll fix: 0.0070515
previous green centroid: 0.232472 -0.0111394 1.14555
previous orange centroid: -0.20424 -0.00951166 1.14473
0.00217586  -0.308562   0.951202
  0.999975 0.00705144          0
0.00670734  -0.951178  -0.308569
green centroid: 1.09359 0.232387 -0.341327
orange centroid: 1.09136 -0.204302 -0.345552
blue centroid: 1.88933 0.007275 -0.351809
rotation matrix: -0.00293042    -0.308594     0.95119
   0.999973  0.00547572   0.0048572
 0.00670734   -0.951178   -0.308569, translation:  -0.334392
-0.0196213
  0.367229
Filtered Height: 0.340299
Filtered Pitch: -1.88448
Filtered Roll: 0.0070515
Filtered Yaw: 0.0051064
X_diff: -0.334392
Y_diff: -0.0196213
transformed green centroid: 0.758 0.218347 0.0259024
transformed orange centroid: 0.758 -0.218347 0.0216773
^C%
```

Fig. 10: Output of our camera calibration.

B. Central Planner

The central planner interprets the detected ball centroids sent from the vision system to create a state representation of the world. The planner then uses it's state history to predict the balls future trajectory. Once the trajectories are known, the planner sends waypoints to the controller to intercept the ball.

1) World Representation: The robot must abstract the complexity of the world into a more manageable representation that it can use to reason and make decisions in. We created *ball.hpp*, *arm_planner.hpp*, and *prediction.hpp* classes to store this information.

The ball class tracked the following variables:

- color
- odds
- coordinate
- coordinateHistory
- velocityHistory
- bestFitLine
- prediction

The arm planner had the following variables:

- balls
- armOuterRadius
- armInnerRadius

The prediction held the following information:

- ballInRangeTime
- ballInRangePosition

- ballInRangeVelocity

2) Tracking: The robot deals with noisy vision data by keeping an internal list of seen balls and solving the correspondence problem. A detection is matched to a ball by the algorithm below:

If a detection can be matched to the ball. Then the odds of the ball is incremented. If no detections are matched to a ball, then the odds go down. We remove a ball from the list once its odds drop below a threshold. This allows for a robust ball state estimates based on noisy vision data.

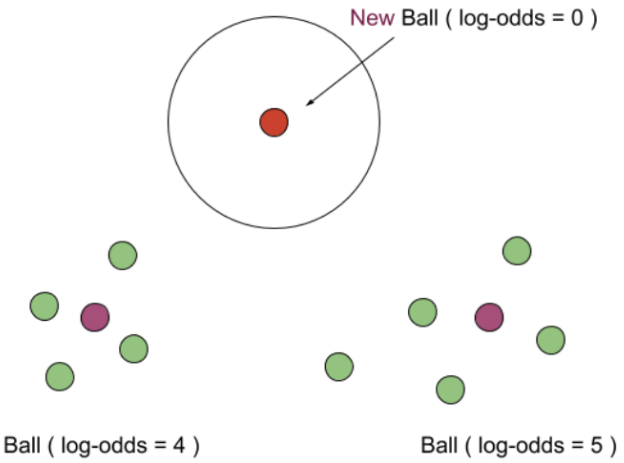


Fig. 11: Ball objects in our state representation are depicted as purple dots, the detection history of that are the green dots that have been corresponded to that ball. The red ball is a new ball because it is not close enough to be corresponded to any of the existing balls. The log odds are also given.

The trajectory and position of a ball is predicted by matching a line of best fit to its past five detections. It's future trajectory is predicted to fall along the line of best fit. The line of best fit is computed using least squares regression.

$$Ax = y$$

We were able to get away with approximating the motion of a ball as linear since our framerate was fast enough, golf balls move straight on flat ground, and our arm's end effector was wide. We had hoped to use an Extended Kalman filter to capture non-linearity in the ball's motion, but failed to get it working. We tried out a standard kalman filter from opencv (which could be extended with a parameter change), but the velocities in the state were never being updated since they were unobservable. We think that the issue was the covariance initialization, but could not solve it. Therefore we made a pivot to define our own physics.

$$v_k = (x_k - x_{k-1})/\Delta t$$

Where x_k is the current state of the ball and v_k is the current velocity. We used the average of the past 5 velocities of the ball for its velocity.

3) Computing a Blocking Plan: After developing a world state and being able to predict the trajectories of the ball, the central planner must decide which ball to target. The planner defines a targt zone as the zone between 2 semicircles. The inner circle has a radius of 10cm and the outer circle has a radius of 17.5cm. The planner computes the time for each ball to reach the outer and inner radius and targets the ball that will reach the inner radius first. The position of where the ball crosses the outer circle is sent to the arm controller to intercept.

C. Arm Design

The arm mimics a human arm playing air hockey, projected to a two-dimensional space. We first used SolidWorks to mock up the configuration of Rexarm parts, and then designed a custom end effector and 3D printed the end effector. The arm uses one AX-12+ and two Dynamixel MX-28 servos to move. We used Computer-Aided Design to develop the end effector and verify that it fit on the servos.

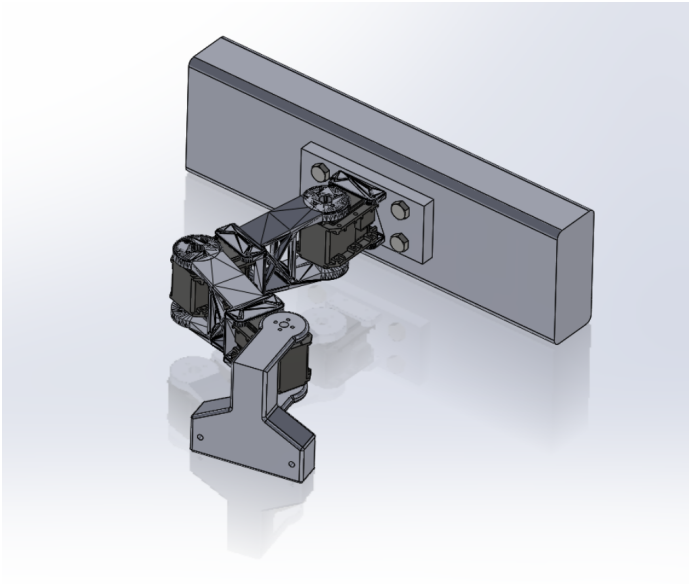


Fig. 12: SolidWorks Assembly including the base and the end effector.

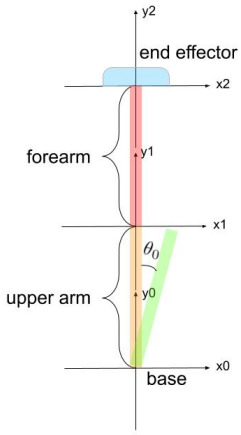
We designed the end effector to not sit directly on the ground, in order to minimize friction. Furthermore, there were through holes to allow different hitting surfaces to be attached. We wanted to have the freedom to attach different materials with more elasticity than the 3D filament.

We also had to create a base and camera mounts that could be easily moved based on whether or not we wanted more cameras or to reposition the camera. We used clamps to secure the camera mount onto the robot.

D. Arm Control

The movement of the arm is controlled by three servos. θ_0 (shoulder) and θ_1 (elbow) control the position of the end effector. $theta_2$ (wrist) controls the angle of the end effector.

1) Forward Kinematics: Forward kinematics are mainly used for debugging. It can calculate the position and angle of the end effector given angle feedback of the servos.



$$p^0 = A_0(\theta_0)A_1(\theta_1)p^2$$

Fig. 13: Diagram of the forward kinematics.

$$\mathbf{A_0}(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & \text{upper\_len}\sin\theta \\ -\sin\theta & \cos\theta & \text{upper\_len}\cos\theta \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{A_1}(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & \text{fore\_len}\sin\theta \\ -\sin\theta & \cos\theta & \text{fore\_len}\cos\theta \\ 0 & 0 & 1 \end{bmatrix}$$

Therefore, when $p^2 = [0\ 0\ 1]^T$, $p^0 = A_0(\theta_0)A_1(\theta_1)[0\ 0\ 1]^T$.

Assume the angle of the end effector in home position showing in Fig. 13 is 0. The angle of the end effector given $(\theta_0, \theta_1, \theta_2)$ is

$$\theta_0 + \theta_1 + \theta_2$$

.

2) Inverse Kinematics: Given a target position (x, y) and target angle ϕ of the end effector, inverse kinematics can calculate the needed angles $(\theta_0, \theta_1, \theta_2)$ of the three servos.

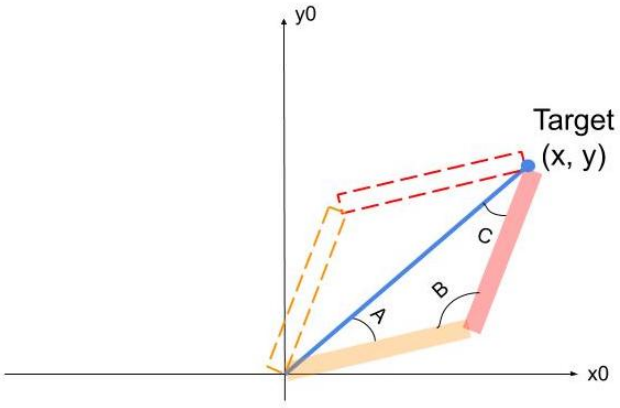


Fig. 14: Diagram of the inverse kinematics.

From the above diagram,

- when $x > 0$,

$$\theta_0 = -(\arctan(y/x) + A)$$

$$\theta_1 = \pi - B$$

Specially, when $y = 0$, $theta_0 = -(\frac{\pi}{2} + A)$
Based on law of cosines,

$$\text{fore\_len}^2 = \text{upper\_len}^2 + \text{target\_len}^2 - 2 \cdot \text{upper\_len} \cdot \text{tar\_len}\cos A$$

$$A = \arccos(\frac{\text{upper\_len}^2 + \text{target\_len}^2 - \text{fore\_len}^2}{2 \cdot \text{upper\_len} \cdot \text{tar\_len}})$$

Therefore,

$$\theta_0 = -(\arctan(y/x) + \arccos(\frac{\text{upper\_len}^2 + \text{target\_len}^2 - \text{fore\_len}^2}{2 \cdot \text{upper\_len} \cdot \text{tar\_len}}))$$

Similar to A,

$$B = \arccos(\frac{\text{upper\_len}^2 + \text{fore\_len}^2 - \text{target\_len}^2}{2 \cdot \text{upper\_len} \cdot \text{fore\_len}})$$

Then,

$$\theta_1 = \pi - \arccos(\frac{\text{upper\_len}^2 + \text{fore\_len}^2 - \text{target\_len}^2}{2 \cdot \text{upper\_len} \cdot \text{fore\_len}})$$

For the angle of the wrist,

$$\theta_2 = -(\theta_0 + \theta_1) + \phi$$

- When $x < 0$,

$$\theta_0 = -\theta_0, \theta_1 = -\theta_1$$

E. Strategy of Speeding Up the Arm Movement

As shown in the Fig. 14, there are two ways for the arm to reach the target. To reach target as the dotted line,

$$\theta_0' = \theta_0 + 2 \cdot A$$

$$\theta_1' = -\theta_1$$

For each target point, there are two ways of movement from current point:

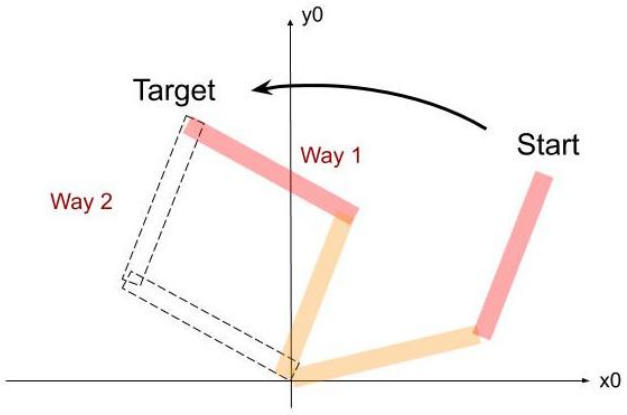


Fig. 15: There are two ways moving from the start point to the target point. See "Way 1" and "Way 2".

Our strategy will choose "Way 1" instead of "Way 2" for two reasons:

- Angle difference of the shoulder servo is less than "Way 2"
- In "Way 1", both shoulder and elbow servos rotate counter clockwise. So they have momentum in the same direction. In "Way 2", shoulder servo rotates counter clockwise, while elbow servo rotates clockwise. So their momentum are against each other and slow down both movements.

From each target point, our strategy choose the movement with less $\Delta\theta_0 + \Delta\theta_1$. If two movements have similar angle difference, we choose the movement that both servo rotate in same direction.

III. Results

A. Vision System

The ball detection takes 14 milliseconds on the Lenovo P51 we use for testing, and the point cloud extraction and transformation takes 7 milliseconds, therefore resulting in a total detection latency of 21 milliseconds allowing us to process each camera stream at 30 frames per second. We found 1 D435 camera to provide adequate results, so for our final demo we didn't add on another camera. We found that the sunlight has a dramatic detrimental effect on the ability of the Intel Realsense cameras to create a depth image. We saw holes in the depth image wherever the sunlight fell on our asphalt felt. We set up vertical asphalt felt in order to mitigate this during our demo, but still had to largely rely on the clouds to provide sufficient protection against the sun. Having done all of our testing in an underground lab without windows, this is not a problem that we foresaw.

B. Central Planner

One of the challenges for the central planner was integrating and testing our predictions with the real world data. For example, we initially were using OpenCV's implementation of an Extended Kalman Filter, but had a hard time getting valid results. We moved over to a simpler, physics based prediction method, that was able to accurately predict the ball's trajectories.

One thing we were missing from the central planner was an easier way to tune the parameters, such as constants like the radius at which we make plans for, the amount of coordinate and velocity histories that we store, and how often we are creating a plan. Parameter tuning would have avoided recompilation time, and would make it easier to debug the robot.

C. Arm Design

We initially wanted to make our own arm design, and actually designed a full SolidWorks Assembly, but realized we would be fine using just a modified Rexarm. It would have been helpful in the beginning to think harder on whether or not we need a custom arm before we spent time designing one. One of the biggest hold ups in our project was actually testing the capability of the arm, which we should have done sooner along in the project. We should have create some more simulations to optimize for arm joint dimensions.

D. Arm Controller

The arm is able to move to any target position within its reach after running the arm controller. But there is latency between sending a plan and receiving a plan, between receiving a plan and moving to the target position. Although we are using the strategy mentioned in previous section to speed up the execution, there is still considerable latency which makes the system cannot respond to a fast-running ball. We solve this by using prediction algorithm in the central planner. As a result, the arm could prepare earlier if the prediction algorithm gives a good prediction. Another way to speed up the movement of the arm is to modify the driver code of it. We could change the PID controller in the driver code or use another mode of the servo. Since we don't have enough time on the project, we didn't try it.

IV. Conclusion

Our robot goalkeeper achieved our goal of interacting with a human audience member, and hopefully teaching them about how we use our vision to hit incoming balls. The planning for our robot idea was where we thrived best, because we made sure that there were different milestones that we could reach and different levels of functionality to follow. We struggled

on starting our code integration earlier in the project timeline, especially because that was one of the best ways to test if the individual parts were working. We also could have written tests for the central planner and controller before we started developing them. One of the problems we had was defining what each piece would exactly do, so integrating earlier would have made the final few debugging sessions go a lot smoother.

Overall we were very happy with the final result, and enjoyed making a robot that could begin to reason about the world. We had all of the pieces integrated effectively and were able to plan based on our representation of the outside world. If we were to continue on this project, we would work on more complex behavior of hitting a movie ball and aiming it either back to the audience member or into a goal.

I participated and contributed to team discussions, and I attest to the integrity of this report.