ECE 383 – Spring 2017

Final Project - Event A

Edwin Bodge (etb10)

December 15, 2017

I have adhered to the Duke Community Standard in completing this assignment. I understand that a violation of the Standard can result in failure of this assignment, failure of this course, and/or suspension from Duke University.

Contents

List of Figures

1 Introduction

In this report, an approach to robotic system design will be detailed and defended. This robotic system utilizes a 6DOF Robot in the Klamp't Online simulator to accurately sense, detect, and hit thrown balls. These objects are uniquely colored, and are thrown towards a "goal". The robot stands in front of the goal and is tasked with colliding with the balls in order to prevent them from scoring points. Balls that pass through the goal result in points deducted from the final score. The first half of this project involves determining the control of the robot based on sensed objects. For purposes of developing this controller, the simulation was provided with an omniscient sensor, which provides accurate position and velocity readings of each ball.

In order to sense the balls in a realistic fashion, the simulation utilizes a camera, otherwise known as a blob detector. This blob detector constructs a 2D representation of a sensed object, providing the x and y pixel location of the center of the object, the pixel width and height of the object, and the color of the object. This sensed information is processed and sent to the perception stage, which allows the robot to make informed decisions about where to block. The details of this system and the process by which it was constructed is the topic of this paper.

2 Summary

Figure 1 illustrates the high level design of this robotic system. The system relies on 4 separate stages: Sensing, Perception, Planning, and Control.

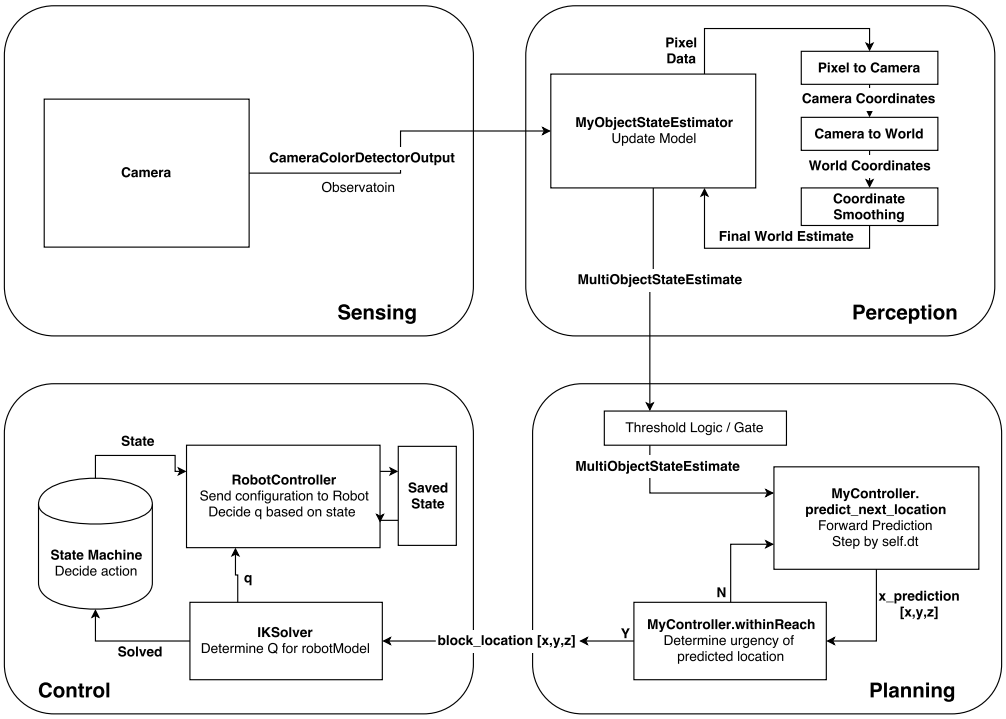


Figure 1: High Level Outline of Robot System Design

As described previously, the blob detector provides 2D information about the location of the ball in its frame of reference, in addition to the ball's perceived width and height. This information is provided in measurement values of pixels, and fed into the perception stage as a *CameraColorDetectorOutput* object.

The perception model transforms the perceived 2D pixel information into 3D world coordinate data; this task comprises the predict stage of perception. At a high level, the 3D world coordinate is determined by mapping the x and y camera pixel locations to actual x and y locations in the camera's reference frame. The 3rd dimension, or camera z-Position, is determined by the scaling factor between the ball's actual radius and

the sensed radius of the blob. This scaling factor determines the depth from the sensor where the object is located. This position is then converted from camera coordinates to world coordinates.

The world coordinates are then passed to a smoothing function. This module relies on stored data of previous measurements to perform two tasks that comprise the predict stage of perception. First, the module utilizes a combination of the new sensor data and the previous measurements in order to determine a smoothed position estimate. This estimate will be highlighted later, however involves constructing a weighted mean of the previous position values. Second, the module utilizes this history of position data to predict the velocity of the object. The velocity is then stored in the model to be used as historical smoothing data in future steps. The final predicted state, containing both the predicted position and velocity, is sent as a *MultiObjectStateEstimate* object into the planning stage.

The planning stage performs three key tasks. First, it determines whether or not the *MultiObjectStateEstimate* object is moving anywhere towards the robot; this is illustrated in Figure 1 as *Threshold Logic/Gate*. Criteria for invalid configurations includes a ball sitting still, a ball with velocity in the positive x-direction (away from the robot), or a ball that has rolled off the table. Second, if the *MultiObjectStateEstimate* represents a ball that may be worth blocking, the planning stage uses a basic prediction model to estimate the projected trajectory of the ball. This prediction takes into account collisions with the ground, and properly handles "bouncing" to predict an accurate trajectory. Third, the predicted locations of the ball are fed into a module that checks whether or not the trajectory will result in a scored goal. If the trajectory is critical, and within reach of the robot, the predicted location of the ball is sent to the controller. Otherwise, the planning model continues to construct the trajectory further in time, up to a certain threshold.

The predicted location of the ball is fed into the Control Stage, as shown in Figure 1 as *block_location [x,y,z]*. This location is fed into an Inverse Kinematics Solver, which produces a configuration q and boolean *solved*. The result *solved* is fed into a state machine, which determines how the robot should behave; namely, whether the robot should update its target location, or if it should remain targeting the same location. The configuration q is sent into the controller, and dependent upon the state of the robot, will be used to set a configuration of the robot.

3 Components in Detail

3.1 Perception Components

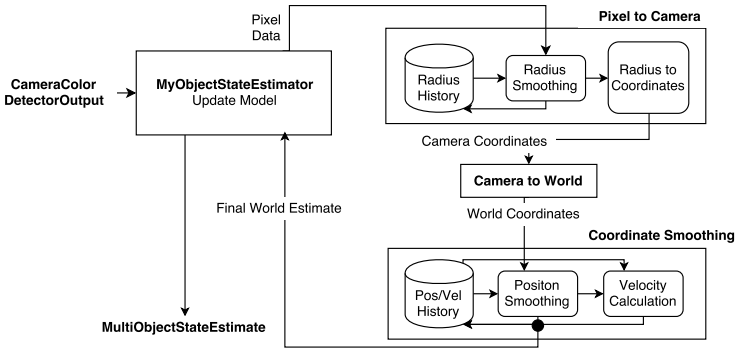


Figure 2: Detailed Components of Perception Stage

The detailed perception components can be found in Figure 2. The perception stage takes in a single input from the sensing stage, a *CameraColorDetectorOutput* object. This object is in the reference frame of the camera, and contains a list of multiple *CameraBlob* objects. Each *CameraBlob* object contains three key pieces of data.

- X and Y locations of the "blob" in the camera's field of view. These values are given in measurements of pixels, where the resolution of the camera is 320 pixels wide, 240 pixels tall.
- Width and Height of the detected "blob".

- Three-digit tuple containing the color of the detected blob.

The module *MyObjectStateEstimator* is the main component within the perception stage as it behaves as an interface between the sensing and planning stages. The subcomponents used to perform perception are controlled by this module, and are implemented as follows.

Pixel to Camera Coordinate Conversion

The *Pixel to Camera* module is responsible for transforming blob data from the camera into position coordinates in the camera's reference frame. The data from each *CameraBlob* object is sent into this module. This data input includes the pixel measurements detailed above in Section 3.1. For this implementation, the *MyObjectStateEstimator* only sends the highest indexed *CameraBlob* object; this is to prevent confusion by trying to target multiple blobs at once. This sometimes sacrifices going after multiple targets, but handles the error where too many objects would confuse the robot. This component is utilized every time that a new *CameraColorDetectorOutput* object is sent to the *MyObjectStateEstimator* module.

The width and height of the blob are averaged in order to estimate the perceived diameter of the ball in pixels; this value is then divided by two in order to determine the radius in pixels. Because the width and height of the object in the blob detector is given with single pixel precision with magnitudes from approximately 5 to 30 pixels, the accuracy and precision of these values are quite low. Therefore, the calculated radius value is run through a historical smoothing module in order to determine a more accurate prediction for radius. The smoothing module utilizes historical radius readings as follows.

$$average = \frac{\sum_{i=0}^{n} i * radius_history(i)}{\sum_{i=0}^{n} i} \tag{1}$$

Older radius history values are given less weight in this calculation, where the newest readings are given the most weight. The history only looks back 4 steps in time from the current reading and recycles data as required; this allows for faster and more up-to-date computations.

Once the smoothed pixel-radius of the ball is calculated, it is fed to to the *Radius to Coordinates* subcomponent. This component uses the pixel-radius of the ball and the actual radius of the ball to calculate the 3D camera coordinates. By switching to "omniscient mode" and printing out the world z-position of the balls as they sit on the playing field, the radius of each ball was determined to be 0.103 units. The law of similar triangles was then used to calculate the x and y locations, as shown below.

$$x_{blob}^{actual} = \frac{radius_{actual}}{radius_{pixels}} * (x_{blob}^{pixel} - x_{center}^{pixel}) = \frac{0.103}{radius_{pixels}} * (x_{blob}^{pixel} - 160) \tag{2}$$

$$y_{blob}^{actual} = \frac{radius_{actual}}{radius_{pixels}} * (y_{blob}^{pixel} - y_{center}^{pixel}) = \frac{0.103}{radius_{pixels}} * (y_{blob}^{pixel} - 120) \tag{3}$$

The ratio between actual radius and the perceived pixel radius was used to determine the actual x and y positions of the ball within the camera's coordinate frame.

The z position of the ball within the camera's coordinate frame was then determined by mapping radius size to z depth in the camera's field of vision. This is done through an algorithm developed and described in Section 5; it follows a similar law of proportions, allowing the ratio between actual radius and pixel radius to determine the depth of the image.

The final output of this subcomponent is a vector containing the objects perceived location in camera coordinates. This value is then fed into the Camera to World module.

Camera to World Coordinate Conversion

This module converts an input of camera coordinates into world coordinates through the provided *Tsensor* transformation matrix. This utilizes the Python command *se3.apply(Transformation, [x,y,z])*.

Coordinate Smoothing

Coordinate smoothing occurs by utilizing a history of position and velocity data to predict the current position of the object. The position is updated first, and relies on an averaging schema similar to equation 1. However, this average includes the model prediction similar to that of a Kalman filter (though without

modeling uncertainty). It utilizes the following operation for each position and velocity pair in memory, where n is the number of time steps that have occurred between the perception of that data point and the current time.

$$\begin{bmatrix} \vec{x}_i \\ \vec{v}_i \end{bmatrix} = \begin{bmatrix} I_3 & n\Delta t I_3 \\ 0_3 & I_3 \end{bmatrix} \begin{bmatrix} \vec{x}_{i-n} \\ \vec{v}_{i-n} \end{bmatrix} + \begin{bmatrix} \vec{0}_3 \\ 0 \\ 0 \\ -ng\Delta t \end{bmatrix} \tag{4}$$

The result of this calculation is used for the arithmetic mean of position and velocity, similar to equation 1. These values are then properly written back to the position and velocity history.

The final output of this module is a 6-index vector, containing the perceived position and velocity of the ball in world coordinates. This is output back to the *MyObjectStateEstimator* module, which then packages the processed velocity, position, and original color reading into a *MultiObjectStateEstimate* object of length one. This object is then passed to the planning stage.

3.2 Planning Components

The planning stage is responsible for determining whether or not the robot should take action to block an incoming ball. It takes an input of a *MultiObjectStateEstimate* object, which primarily contains the color, position, and velocity of an incoming ball in world coordinates. The details of the planning stage are shown in Figure 3

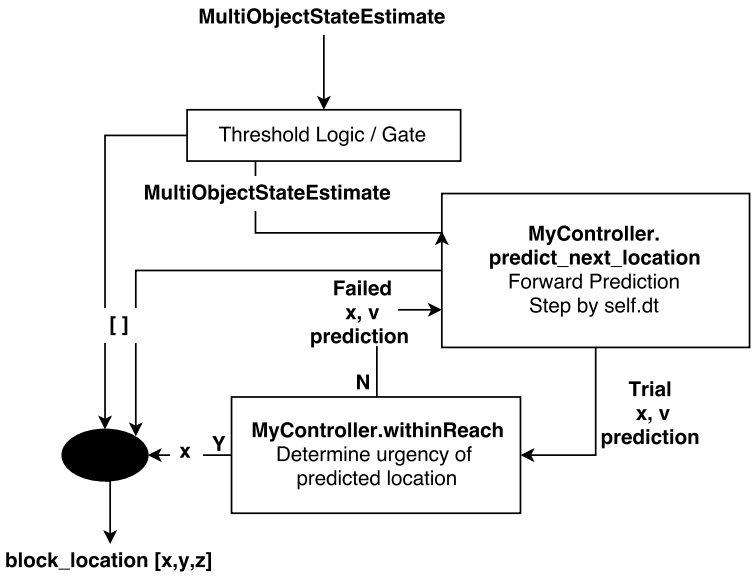


Figure 3: Detailed Components of Planning Stage

Threshold Logic

The *MultiObjectStateEstimate* object is first processed by a simple logical check. This subcomponent checks for three criteria to deem a *MultiObjectStateEstimate* object negligible.

- If the ball's perceived z-position is less than -0.5, it has fallen off the table.
- If the ball's perceived x-position is greater than 3, it has likely not been launched.
- If the ball's perceived x-velocity is positive, it is rolling in the opposite direction of the goal and should be ignored.

If the *MultiObjectStateEstimate* object does not meet these criteria for exemption, it is then passed on to the next module unchanged. If any of the above criteria are true, there is no need to block an object, therefore an empty location is sent into the next module as [ ].

Predict Next Location

This module only runs if the unmodified *MultiObjectStateEstimate* object is passed to it. This module is responsible for predicting the trajectory of the ball based on the input world velocity and position. This module uses the physics displayed in equation 5.1 in order to predict the next location of the ball. This is done iteratively, where the next state is predicted, then immediately passed to the next module. This state is composed of the object's state and velocity. If the next module does not succeed, which will be explained in the next section, this module is passed back its predicted value. It will then process this predicted value in the same manner as it handled the *MultiObjectStateEstimate* object, using equation 5.1. This module and the next module repeat this process for up to 20 cycles, or until the next module is successful. Upon reaching 20 cycles, it is assumed that the ball should be ignored, which will be detailed in the following section. The module then passes an empty list to the output of the planning stage, signifying that the controller should not change configuration.

Within Reach

This module accepts the input position and velocity vectors from the *Predict Next Location* module, which are position and velocity estimates along the trajectory of the ball. This module checks whether or not the predicted location falls within the realm of reachable space for the robot that would allow the ball to score. The criteria for a critical ball position are as follows.

- If x-position is between -2.5 and -1.7 units. This is approximately from the goal to a few units in front of the robot.
- If y-position is between -1.2 and 1.2 units. This covers the width of the goal.
- If z-position is between 0.09 and 1.2 units. This covers the ground up to the top of the goal, though prevents the robot from trying to run its arm into the ground.

If the projected position of the ball is within this range, then the module passes the current position to the output of the planning module. This signifies that the ball is currently on a trajectory to score at this specific position. If the projected position of the ball is not within this range, then the module passes the current position and velocity back to the *Predict Next Location* module in order to continue processing. If this module executes 20 times for a specific perception value without sensing a ball projected to score, this module passes an empty array to the output of the planning module.

In summary, the planning stage outputs a position vector in world coordinates. If this vector has 3 values, then it is a valid location where the ball will be. If the vector is empty, then the perceived ball is predicted not to score.

3.3 Control Components

The control stage is responsible for handling projected ball locations, and consequently programming the robot to accurately move in response to these projected locations. The details of this stage can be found in Figure 4.

The input to the controller is a vector specifying the projected position of a scoring ball. This vector, as described in the previous section, will either contain three entries to specify the location of a scoring ball, or zero entries to specify that there are no balls predicted to score. If the vector is empty, a boolean signal is sent to the state machine that there are no balls to protect against. The state machine handles this response as necessary, which will be detailed later. Otherwise, if the input vector contains three values, then this value is sent to the *IK Solver* for processing.

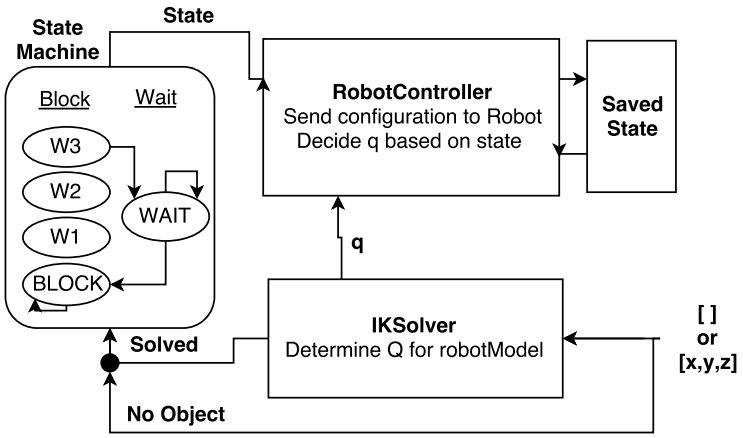


Figure 4: Detailed Components of Control Stage

IK Solver

This module is only called when the control stage successfully receives a position vector specifying the location of an incoming ball. The IK Solver utilizes this position and information about the 6DOF robotModel in order to perform inverse kinematics. The solver seeds the robot with its current configuration, retrieved using the command *robotController.getSensedConfig()*, and performs 100 iterations trying to solve the IK problem where the desired end-effector location lines up with the position vector of the incoming ball. The solver runs up to 100 times on the end-effector, or until it successfully determines a configuration. If this process was unsuccessful, IK is then performed where the ends of other links are compared against the position of the incoming ball. This opens the possibility for other links to act as the goalie, just in case the incoming position is unreachable by the end effector.

The IK solver outputs two signals. First, it outputs the recommended configuration for the robot. In the case of a successfully solved IK problem, this output is a vector containing a configuration rotation for each joint. In the case of an unsuccessful IK problem, this output is an empty vector. These vectors are sent to the RobotController module.

Second, the IK solver outputs a boolean value to the state machine. This boolean represents the success of the IK solver; hence, 1 for success, 0 for failure.

State Machine

This state machine determines what state of control the robot should be under. It takes an input as a boolean variable, which represents whether or not inverse kinematics were successfully solved to intercept a ball's trajectory. This input either comes from the input of the control module, when an empty vector is input, or when the IK Solver is unsuccessful in determining a configuration to intercept the robot.

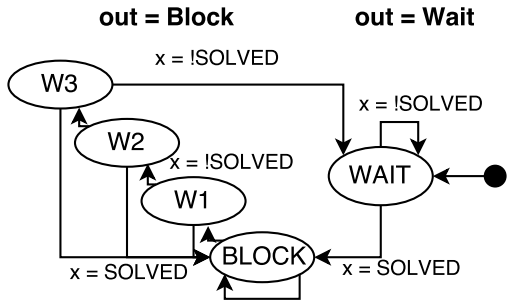


Figure 5: State Machine of Robot Controller

The state machine has 5 different states, as shown in Figure 5. The robot is initialized at the "WAIT" state. With input "Not Solved", the state machine moves towards the "WAIT" state. Upon receipt of a "Solved" signal while at any state, the state machine moves to "BLOCK". It remains at this stage until a "Not Solved" signal is received, when it then moves through the stages "W1", "W2", and "W3". Stages "BLOCK" through "W3" produce an output string of "Block", while the "WAIT" stage produces an output string of "Wait". These signals are sent to the RobotController, which determine the course of action for the robot. The motivation behind this state machine is to require 4 unsuccessful ball location targeting cycles to occur before definitively determining whether or not the robot should be waiting. This will be further explained in the *Planning and Control Strategy* section.

Robot Controller

The Robot Controller module is responsible for programming the robot. It takes two inputs: the state of the robot system, as specified by the previous state machine, and the configuration q from the IK Solver.

If the state input is "Block", then the robot will perform one of two actions. If the configuration q is a valid list of joint encodings, then the robotController will be programmed with the command *robotController.setMilestone(q)*. If the configuration q is an empty list, then the robot will not be programmed during this cycle. This means the robot will stay at configurations programmed in previous cycles; this is generally to avoid programming conflicts due to perturbations in trajectory prediction data.

If the state input is "Wait", the robot will go to a "waiting" configuration. This strategy will be explained in Section 4, however broadly returns the robot to a strategic location to help it become ready to block the next incoming block.

This concludes the description of components utilized in this robot system. The components have been detailed extensively, however the next two sections will illustrate the underlying logic and strategy behind completing the assignment. Much of this information has been touched on during this section, therefore the subsequent sections are written to reduce redundancy.

4 Planning and Control Strategy

The previous sections have highlighted the components utilized to perceive objects, and consequently process their signals in order to plan-for and defend-against incoming objects. The state machine diagram shown in Figure 6 illustrates the high-level methodology used to make planning and control decisions. This section will highlight these decisions, and the precise computations behind them.

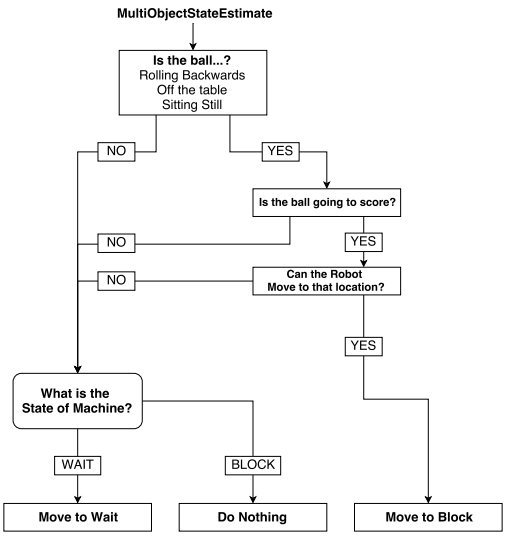


Figure 6: High Level Control State Machine

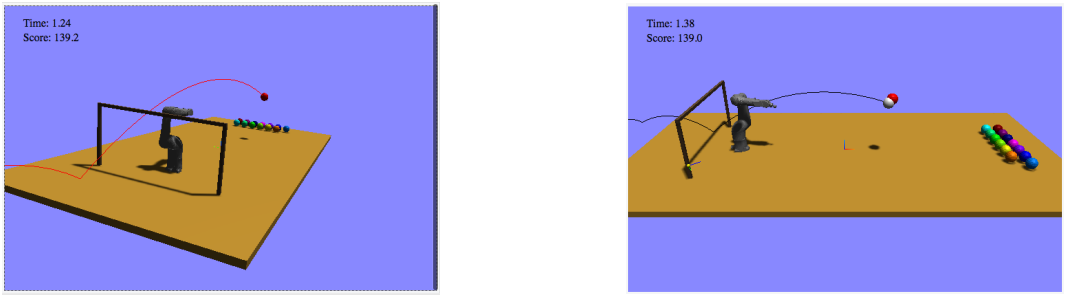
4.1 A Note on Failure Handling

In this model, a "failure" could be triggered by three different phenomenon. Rather than think of these events as "failures", it would be more accurate to consider them as signals not to block a ball. The following discussion will highlight what these events are, and how they are handled.

First, the state (position and velocity) passed to the planning stage in the *MultiObjectStateEstimate* object may violate one of the three criteria highlighted in Figure 6. A ball's state may be rolling backwards, off the table, or sitting still. Any of these states signal that the ball is not a scoring threat, and should therefore not be acted upon. Second, the predicted trajectory of the ball may not be set to score, as determined by the *Within Reach* module described in Section 3.2. In this case, we assume that the ball is not a scoring threat, and we should therefore not act upon this ball. Third, we may fail to properly solve the inverse kinematics problem. This event would actually be considered a "failure", as the ball is predicted to score, however the robot cannot move to block it. In this case, we simply cannot move the robot to a location, therefore allow the state machine to determine the subsequent configuration of the robot.

4.2 Trajectory Mapping

The *Predict Next Location* module detailed in Section 3.2 generates the predicted trajectory for the ball. This module takes into account "bouncing" when the z-position of the predicted state falls to 0 or lower. This allows for accurate prediction of the ball's location even when it is moving far from the robot. Examples of this predicted trajectory can be seen in Figures 7a and 7b.



(a) Omniscient Mode (b) Blob Detector Mode

Figure 7: Trajectory Tracking Examples

Each position in the ball's predicted trajectory is then checked to see if it will score, and if it is reachable by the robot. As the trajectory is calculated, it grows towards the goal and is checked in real time in order to reduce the amount of computation that is required. The first position that could be blocked by the robot is immediately chosen. As shown in Figure 7, the predicted trajectory accounts for bouncing, which will allow the robot to defend against "ground balls".

4.3 IK Solving

The process of Inverse Kinematics solving has primarily been described in the previous discussion of components. The main strategic addition to this module is the allowance of multiple joints to serve as the "goalie" for this problem. As described previously, the IK Solver first tries to solve the IK problem to defend with the end-effector. However, upon failure, the IK Solver then iterates through every other link to look for an unconventional solution.

4.4 State Machine

The motivation behind the state machine detailed in Section 5 is to help filter out bad readings for the robot model. With noisy readings from the previous stages, sometimes a ball is predicted to be harmless while it's truly heading towards the goal. In this case, the state machine will receive the signal "Not Solved", and consequently move towards the "WAIT" stage. Without the 3 buffer stages between "BLOCK" and "WAIT", this noisy reading would cause the robot to move back to its waiting stage, while it should have

been moving towards an obstacle. The state machine provides a buffer between the "BLOCK" and "WAIT" stages to handle the noisy readings from the sensors.

4.5 Waiting and Blocking

There are two main strategies for robot control, which are defined by "BLOCK" and "WAIT". The blocking state has been extensively described in previous sections. The waiting state will return the robot to one of three configurations.

- If the robot is currently oriented a certain amount above of the x-axis, then program to a configuration where the robot will be low and facing forward, slightly above the x-axis.
- If the robot is currently oriented a certain amount below of the x-axis, then program to a configuration where the robot will be low and facing forward, slightly below the x-axis.
- Otherwise, return to a forward-facing, neutral-height position.

The purpose of this control is to allow the robot to return to a strategic location where it will easily be able to reach the next incoming ball. When the robot was programmed to return only to the third reset position, and a ball was launched towards its previous location, it would not be able to turn around quick enough to respond to the incoming ball. Therefore, the robot is programmed to move towards a location that requires the least amount of movement. Allowing this to happen puts the robot in a strategic location without inhibiting the robot's ability to shift directions while in transition to the waiting location.

4.6 Shortcomings

The planning and control for this robot is fairly robust, and I am quite happy with its performance. There are two main shortcomings that could have been improved upon.

First, the robot does not take into account collisions with the soccer goal. It rotates freely as if there are no obstacles in its way, which may cause reduction of points in the simulation. A way to get around this was to ensure the robot was primarily configured with locations that would not collide with the goal. By keeping the robot mostly out in front of the goal, it greatly reduced the probability that a collision would occur.

Secondly, the simulation constantly loses points because the velocity of Joint 6 is deemed outside of the joint velocity limits. This was discussed on Piazza, and seems to be an error with the simulation model. The configuration of Joint 6 is constantly driven to 0 in hopes of eliminating this error, however this does not work.

5 Perception Strategy

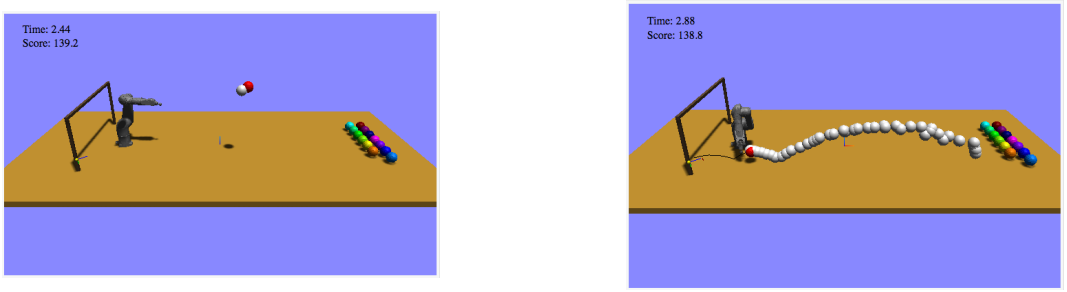
The perception strategy for this robot system has been greatly described in Section 3.1. This section will answer some of the questions provided in the Final Report Guidelines, and highlight a number of design decisions.

5.1 Estimates and Error

The perception for this robot system predicts both the position and velocity for the sensed "blob". It does this through the smoothing function highlighted in both Section 3.1 and equation 1; additionally, it uses the underlying physics behind equation to compute a more accurate position and state velocity. Both the position and velocity are required because they are passed to the planning stage; this stage requires both of these measurements in order to predict the trajectory of the ball.

The blob detector gives the pixel location, width, and height with 0.5 pixel of uncertainty. This is therefore the minimum amount of uncertainty we can consider our estimate to have. By using the previous states of the balls in the smoothing calculation of the new state, this error is generally propagated from state to state. The hope is that the smoothing function provides a counter-weight to these inaccuracies, which it seems like it does. Figure 8 illustrates the perception model's ability to track a ball's location.

Figure 8a visualizes of a single predicted location (white) on the world coordinate system, next to an actual ball (red). Figure 8b visualizes the entire history of predicted locations for a single ball. This shows that as the ball gets closer to the camera, and gains more historical information about the ball's state, the prediction of the ball's state becomes more and more accurate.



(a) Single Object Tracking

(b) Trace Tracking

Figure 8: Blob Detecting Perception Visualization

5.2 Camera Z-Position from Blob Size

The z-position of the ball in the camera's coordinate frame is the most difficult part of the ball's state to perceive. This is done by comparing the detected blob's size in pixels to the actual size of each ball. As described previously, the pixel-radius of the blob detector's output is determined by averaging the width and height and dividing by two. This value is then compared to the actual radius of the ball 0.103 units, to achieve a ratio of actual to pixel.

In order to compute the z-position of the ball, the pixel radius must be utilized in a sensible way. In order to construct the equation that converts from radius to z-position, I tested the robot's predicted configuration with a number of hard-coded values for z-position coming out of the perception model. This meant that x and y positions were accurately represented, however z was constantly set to the same value. The simulation was then run; when the estimated state exactly aligned with the actual ball on the screen, the pixel radius was taken down. The relationships between the radius and z-position created the curve shown in Figure 9. The equation representing this curve is $zpos = 15(radius)^{-1}$. This equation is therefore used to predict the z-position of the ball in the camera's coordinate frame.

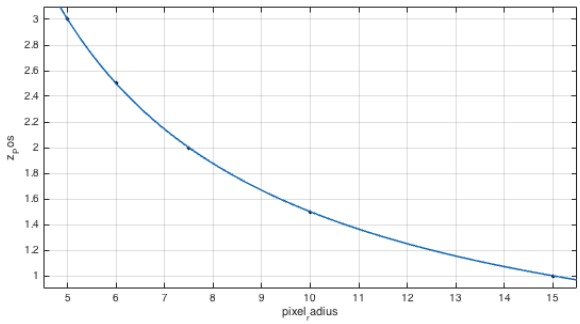


Figure 9: Mapping Radius to Camera Z-Position

5.3 Shortcomings

The implemented perception model does most of what I would like for it to do. The main shortcoming of this implementation is the lack of consideration for uncertainty. The model used to predict the smoothing location does not directly account for the error in pixel-readings. This is largely mitigated by the smoothing that actually occurs, however greatly inhibits the state perceptions at large distances from the camera.

6 Reflection

In general, I am very pleased with how this robot system works. I had a great deal of fun implementing it, and am glad to see that even on harder modes with my own perception state, I am able to defend against a large number of balls.

I've discussed what technical details I would have liked to improve in Sections 4.6 and 5.3. Additionally, I would have liked to improve the logic the robot follows during the "WAIT" stage. The current model adjusts the location to increase the probability that the robot can get to the next location of an incoming ball, however is not perfectly implemented. This could include gaining better insight to how the robot moves, and changing the waiting position based on this performance.

Getting the sensor working was much harder than expected. The readings were quite noisy, and the orientation and conversions took a large amount of time to understand. However, the solutions to HW 5 were quite helpful in understanding the configuration. By drawing the goal, camera, and robot, it made it much simpler to understand; the visual debugging was very helpful as well. I've left a number of the visualizations on for demo purposes, including the projected trajectory and perceived location of the thrown ball.

Should this robot be implemented in the real world, there would be a number of additional challenges this implementation would face. First, the blob detector would have a number of obstacles that get in its way. Currently, the behavior of the camera indicates that it might see through the robot in order to detect the location of the ball. This is purely off of observation rather than empirical data. Second, the blob detector will need to be constructed such that it is highly selective in what it deems an object. The background of the camera image will likely not be solid like the simulation, and the colors of balls will change based on the shadows and highlights in the real world. The solution to this problem would either need to come from improvements to the blob detector, or in improved detection in my perception model of what objects are balls, and which should be ignored. Third, we would need to consider air resistance, ground friction, and even wind as factors in the trajectory prediction of our model. This would cause the trajectories of the balls to vary greatly compared to our simulated model. Lastly, we would need to consider using multiple blob detectors in order to increase the field of view for our sensing stage. This would mainly be the case if the problem statement allowed balls to realistically come from more angles than only head-on.