Robot Batter
ECE 383 Final Project C

Kevin Do
kkd10@duke.edu

1 Summary

The robot's task is to shoot balls into a goal (Figure 1).

The robot has three hardcoded paths that send the ball to the left, center, or right of the goal. State estimates of the world are obtained from the camera/blob detector using predetermined geometric relations. From these state estimates, the robot makes predictions about the world state in the future and determines when to shoot and which stroke to pick. The robot design is modular, and each module performs one of these subtasks (Figure 2).

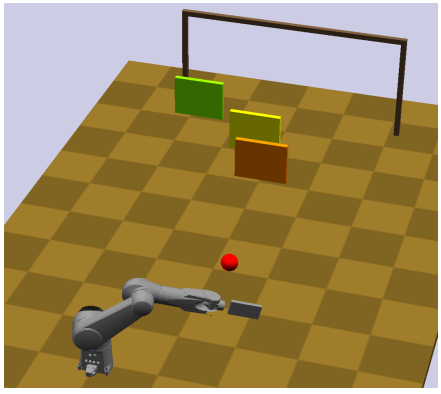

Figure 1: The goal of the robot is to strike the ball (red) and shoot it into the goal (brown) past the goalies (orange, yellow, green).

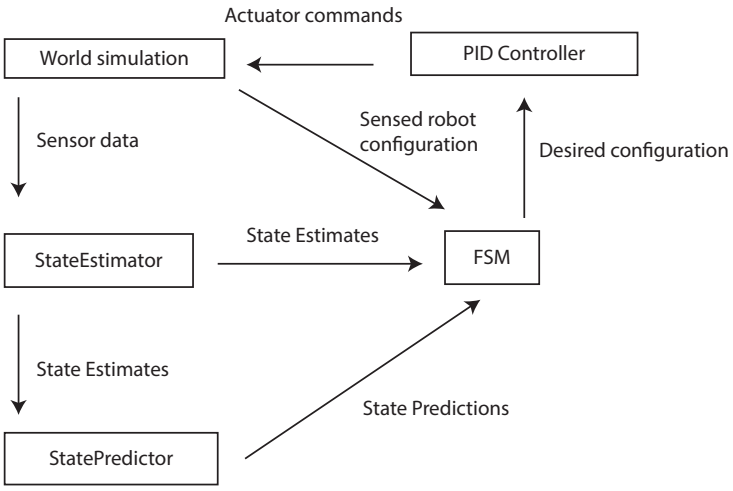

Figure 2: Block diagram of system design.

2 Components

World simulation This module is given in the assignment and simulates the environment and the goalies. Sensor data is generated by the world simulation. The inputs to the world simulation are data files (e.g. XML) defining the world and robot. The output of the world simulation to the robot consists of `CameraColorDetectorOutput` objects containing `CameraBlob` objects. Each `CameraBlob` object has a color, width, height, x coordinate, and y coordinate.

PID Controller This module is built into Klampt. It takes a desired configuration and desired velocity (each a vector of size 7) and uses PID parameters to define torques to send to the actuators. This controller is invoked every time step by the FSM.

FSM The finite state machine contains the majority of the complex high-level logic. Based on its current state, the robot decides between several different subgoals (e.g. strike the ball or prepare for the next strike). The FSM is invoked every step and can be considered the entrypoint of the user-defined logic.

StateEstimator This module takes the `CameraBlog` objects from the blob detector (in camera coordinates) and produces a current state estimate (in world coordinates).

StatePredictor This module takes state estimates and stores it in its internal memory. It then uses this history to provide predictions about the world simulation's state in the future. The most important StatePredictor is the SinePredictor, which assumes that the data is well-modeled by a sinusoid and makes predictions accordingly. This is the most error-prone of the modules, since the curve fit can fail if it is not seeded with good initial guesses for the parameters. This is invoked whenever the robot is in the waiting state.

3 Planning and Control Strategy

The general strategy is to hardcode several loops (Figure 3) into the robot that send the ball into either the left, center, or right of the goal. Having multiple paths to choose from ensures a high probability that at least one of these paths will be free. Then, the robot chooses the path with the greatest distance between the ball's path and the goalies.

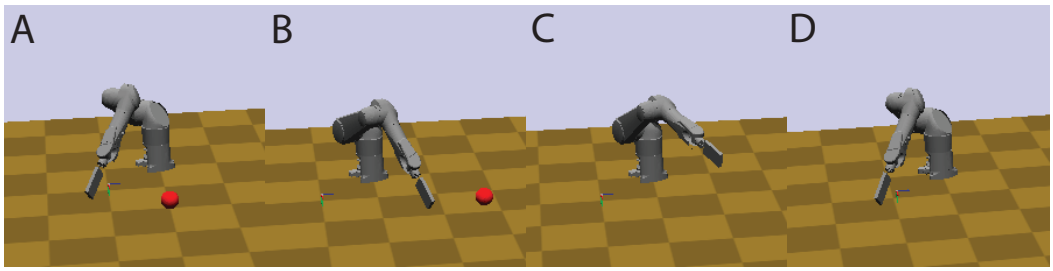

Figure 3: Timelapse of typical robot stroke: prestroke (A), poststroke (B), prerecovery (C), and postrecovery (D).

3.1 Hardcoded path generation

We wanted to avoid the runtime cost and potential lack of solutions that comes with runtime inverse kinematics (IK). Therefore, we hardcoded paths into the robot. To generate the paths, we used Klampt's built-in IK solver to generate solutions that would place the robot arm near the ball's starting location. These solutions were then tuned manually.

In our tuning process for different configurations, we preferred solutions with the second joint up rather than down to minimize the chance of terrain collisions. Furthermore, we preferred ball paths as close to the target (left, center, or right) as possible. Finally, we preferred configurations that yielded low variance in the ball paths (Figure 4).

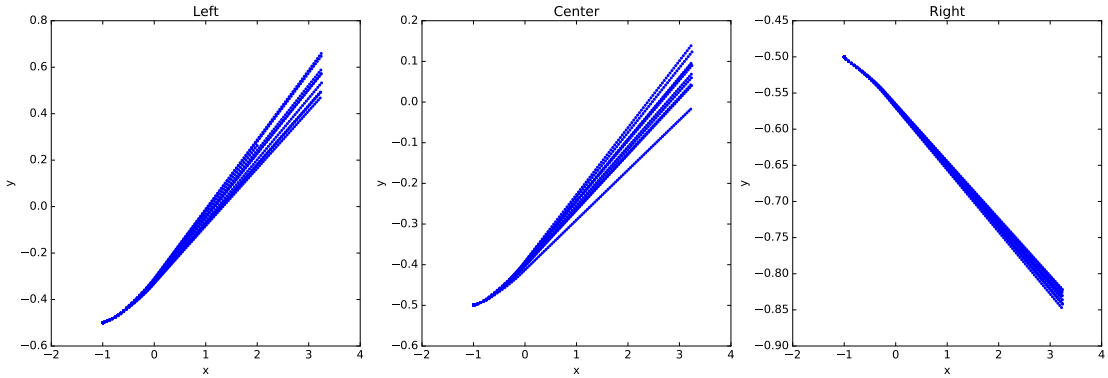

Figure 4: Plots of ball paths following the three different strokes.

3.2 State prediction

A state predictor is a module with two primary functions:

- `addPoint(self, t, o)`
- `predict(self, t, name)`

The first function's purpose is to train the state predictor by adding to its memory a state estimate `o` taken at time `t`. The more states that the state predictor has in its memory, the better predictions it can make about the future. The second function's purpose is to retrieve the predictor's best guess about the state of the object identified by `name` at time `t`.

The robot has two types of state predictors: a linear predictor and a sine predictor. The linear predictor assumes that the object in question takes a linear path, and predicts using linear extrapolation. The sine predictor assumes that the object in question takes a sinusoidal path. Predictions are made by fitting the general sine curve $f(t) = A\sin(Bt + C) + D$ (using scikit-learn's `curve_fit` function) to the history and then plugging in for the desired prediction time. The sine predictor has two failure modes: it can take too long and it can fail to find an optimal curve fit. Both of these failure modes are addressed by caching previous parameter solutions. Doing so helps improve the convergence time for the curve fit since we expect solutions to be similar over time. Furthermore, this caching allows us to still make reasonable predictions if the curve fit fails.

3.3 Choosing to shoot

The robot checks two conditions before deciding to strike the ball. First, the robot checks that the ball is ready: it must satisfy the robot's assumption about its starting position as well as be still. Second, the robot checks that there is an open path. It can do so because the robot has been hardcoded with the ball's traveltimes to the goalies along each of the three possible paths. Based on its predictions (see above) of the goalie's positions, it can decide whether any path is open. If multiple paths are open, the robot picks the one that maximizes the minimum distance to a goalie.

3.4 Finite state machine

The finite state machine (FSM) contains the bulk of the high-level controller logic. Technically, the FSM contains around 100 states, but many of these states can be grouped together. We therefore group the FSM's states into states and substates, where the state is a string and the substate is a whole number.

In an effort to reduce variance in the robot's paths, state transitions are required to satisfy two conditions: (1) the robot's sensed configuration must be suitably close to the desired configuration at the state transition and (2) the robot's sensed velocity must be below a given threshold. Ensuring these conditions means that each state can be considered relatively isolated from the others, since

each state knows to a high degree of accuracy the robot's configuration and velocity when it takes control.

To control the speed of state transitions, we implemented a custom motion queue. The motion queue takes the start, end, and number of frames as parameters and generates intermediate goals to be fed to the PID controller. These intermediate goals are generated through a linear interpolation in configuration space. Though this can lead to unintuitive motion in Cartesian world space, the paths that we hardcoded into the robot assume a linear interpolation through configuration space.

The states themselves are detailed below and in Figure 5.

precycle This is the robot's initial state. This state is similar to the post_recover state.

waiting The robot is waiting to strike the ball. It waits for the ball to be ready and for there to be an open shot before shooting.

pre_stroke The robot is moving to the chosen stroke's prestroke position.

stroke The robot is in the middle of performing the hardcoded stroke and hitting the ball.

pre_recover The robot is moving from the stroke's poststroke position to the hardcoded prerecovery position.

recover The robot is moving from the prerecovery to the postrecovery position.

user The robot has exited the FSM loop and is under user control.

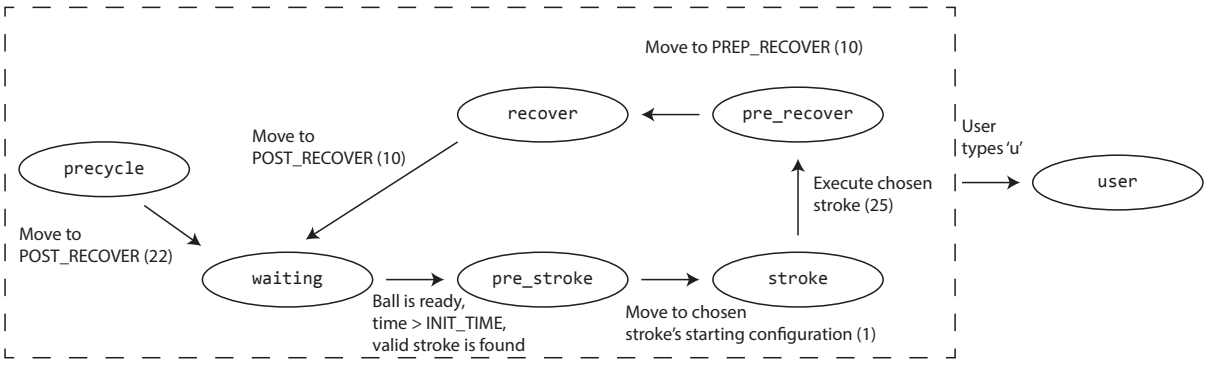

Figure 5: Finite state machine diagram. Numbers in parantheses indicate the length of the state transition in frames for the motion queue interpolator. If the user types 'u' in any state, the FSM switches to the 'user' state.

4 Perception Strategy

The robot's perception strategy is relatively unsophisticated because the problem is not as complicated as the controls problem. This is largely because the perception problem is in one dimension,

the y dimension. The x coordinates of the goalies are fixed at 2.0, 2.5, and 3.0 for each of the goalies, and the z coordinates are not used by the controller at all (but even if they were, they too would be fixed at some constant value.) Furthermore, our robot does not need instantaneous velocity estimates in any dimension, so our perception system does not return them.

Though the camera (and therefore the blob estimates) are noisy, we make the simplifying assumption that the mode of the posterior predictive distribution occurs at the estimate itself (a valid assumption given zero-mean, symmetric noise). This robot therefore does not take into account any variances or covariances. The noise in the blob estimates seems to be relatively low, however, and the curve fit does not fail, so state predictions are still very accurate.

4.1 Goalie state estimation

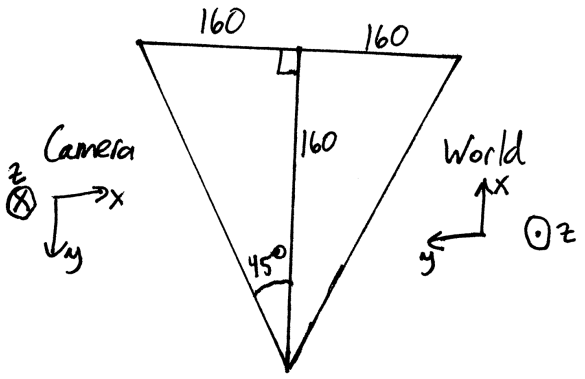

Figure 6: Diagram depicting the geometry used to calculate position from the list of blobs produced by the blob detector. The numbers are in "pixel units", though ultimately the units are irrelevant because only ratios are used for calculation.

The y coordinate of a goalie can be estimated from its blob (Figure 6). Note that the x coordinate of the camera maps linearly to the y world dimension. Furthermore, we know the camera's position to be at $(-1.5,-.5,.25)$. Let (x_c,y_c) denote the center of the blob, x_g denote the known x coordinate of the goalie, $\hat{y}_g$ denote our estimate of the goalie's y position, and α be an unknown. Exploiting the geometry of the situation and the fact that we know the distance to the goalie in the x world dimension, we can drop the z world dimension and write

$$(-1.5,-.5)+\alpha(160,160-x_c)=(x_g,\hat{y}_g) \tag{1}$$

$$\alpha=\frac{x_g+1.5}{160} \tag{2}$$

$$\hat{y}_g=-.5+\alpha(160-x_c)=-.5+\left(\frac{x_g+1.5}{160}\right)(160-x_c) \tag{3}$$

4.2 Ball state estimation

The planning and controls algorithms detailed above do not actually need the ball's current position, but rather only whether or not the ball is ready to be struck. Therefore, in an effort to keep the robot implementation as simple as possible, the state estimator hardcodes the blob attributes when the ball is ready to be struck, and returns a nonsensical ball state if the blob attributes are not within the hardcoded range.

5 Reflection

In testing, this robot seems to work very well. It gets a perfect score of 100 on all three levels (easy, medium, hard). In building this robot, we were surprised by the difficulty of finding suitable strokes in configuration space (i.e. we were surprised by the difficulty of the inverse kinematics). The manual search for good strokes was very time-consuming and tedious, despite not being very conceptually difficult.

This robot could be improved by increasing the number of hardcoded strokes, thus allowing it to be able to exploit more holes. Furthermore, the path space could be searched more carefully to find suitable strokes in configuration space, especially strokes that yield low-variance ball paths. This search in stroke space could be improved by evaluating strokes automatically using some sort of cost function, and highlighting the best paths for human evaluation.

This robot would face significant challenges if it were to be ported from this simulation to real life. A real-life camera could suffer from non-zero-mean noise, which could throw off the curve fit. The robot might not have perfect joint encoders, resulting in errors in the sensed and commanded configurations. If the ground were not perfectly flat, then the ball paths might see more variance, which will throw off the highly-tuned logic to determine whether a certain path is safe.