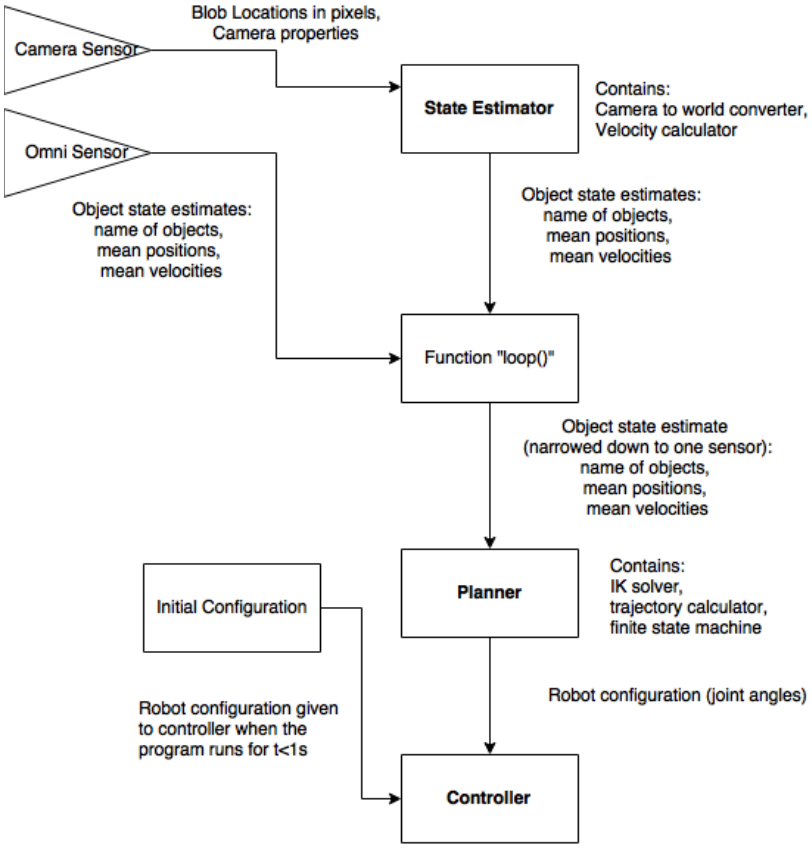
ECE 383 Final Project

System Design - Event A

Pritak Patel (pup), December 19, 2016

I. Summary

The design of my overall system was very similar to the outline provided. It contained a controller, state estimation/perception, and planner as its main components, and some other minor functions/ components that assisted the main components. A block diagram is shown below depicting all of the main connections between components to better understand my code.



The function blocks in bold are the main functions of my design, each of which have several sub-components that I will describe in further detail in future sections. In general, data from the camera sensor is given to the state estimator, which calculates position and velocity and sends it to the function "loop()". If using the omniscient sensor, position and velocity values are given directly to the loop function, which then narrows data down to a single sensor. The data from the sensor is given to the planner, which calculates the trajectory (end position) of the of the ball and sends that point to the IK solver which returns a robot configuration (joint angles). The robot configuration is then given to the controller, which moves the robot. When the robot is initialized, it moves to a starting configuration within the first second to assist the IK solver and to prevent terrain collisions.

II. Components

A. State Estimator

a) Purpose: To convert sensor data from the camera into world coordinates so they can be used by the rest of the program. Additionally, to calculate velocity estimates by using 2+ mean position estimates and dt values so that trajectory calculations in the planner component is accurate.

b) Input: observation (a CameraColorDetectorOutput reading, constructor in common.py). CameraColorDetectorOutput reading -> which is a list of CameraBlobs specifying regions of detected blobs -> A CameraBlob is a blob on screen in image coordinates that includes color (3D tuple) of the blob, x and y positions and width and height in pixel units (floats).

c) Output: MultiObjectStateEstimate (constructor in common.py) MultiObjectStateEstimate -> which is a list of ObjectStateEstimates, corresponding to all of the objects currently tracked by the state estimator -> an ObjectStateEstimate contains the name of the object and x (mean position and velocity estimates as a 6D vector). *Note that covariance was not used in my program so it was never instantiated using the constructor.

d) How is it invoked?: It is invoked every time step (dt) by the function "loop()"

e) Implementation: This is a custom implementation based on my own observations of the problem. My code uses a global dictionary that stores all of the state estimates of a blob as time passes under one key (the blob's color). The state estimates that are put into the dictionary are the positions [x,y,z] as a list so that the value of any key is a list within a list. To calculate these state estimates, I had to make observations and calculate values using trigonometry (which I will describe in more detail in a future section). Once x,y,z values were correct in world coordinates (by applying Tsensor given to my calculated points using se3.apply()), I then used them to calculate velocity. Velocity components were calculated separately by simply taking the last two readings, subtracting them and dividing by dt. In the end a single vector was put together in the format (x, y, z, xvel, yvel, zvel) and used to create an ObjectStateEstimate object. This object was then added to a list of other objects as a for loop goes through all of the blobs seen by the camera in a single reading. To assist my planner component, blob keys were deleted entirely once their xposition values exceeded -2.2 in world coordinates (once the planner no longer needs the values) so that the dictionary doesn't get too large and values don't interfere with other blobs being sensed. An additional check that I used was that if there was less than 2 readings in the dictionary for a given key, the object should not be added because velocities wouldn't have been able to be calculated.

f) Potential failures: My state estimator sometimes fails with velocity estimates when the readings are not very accurate. This however doesn't effect the planner usually because my planner only reads from the state estimator once, so the chances of the error being in that single time step is very slim. Additionally, I created a check in my planner to check if the velocity readings look "reasonable" and if not to take a previous reading instead. Another potential failure case is if many balls are moving at once and the previous blob hasn't yet moved past -2.2 in the x direction, which would create problems for the planning component.

B. Planner/Control

a) Purpose: To take in object state estimates and use a finite state machine to convert them into robot configurations (joint angles) when needed that can be sent to the controller. The planner also includes two sub-components, the IK solver and the trajectory calculator which will be explained in their own sections.

b) Input: MultiObjectStateEstimate (list of ObjectStateEstimates containing position and velocity estimates for all seen objects - 6D vector), dt (time elapsed since last call - float), RobotController (only to send a command to it after calculations are completed - object instance), sensorReadings (used for parsing the name "blobdetector or omniscient" to see which sensor is being used).

c) Output: No returned values. A robot configuration (7D vector of joint angles) is sent to the robot controller.

d) How is it invoked?: It is invoked every time step (dt) by the function "loop()"

e) Implementation: This is a custom implementation based on my own observations of the problem. There are 3 states in my finite state machine: initialization, switch, and moving. The state machine will be explained in more detail in a future section. As an overview, the robot is put into the initialization state when the robot is reset and in this state it also moves to any specified start position. In both the moving and initialization states, the planner is going through all of the object state estimates every time step and determining if the next ball has begun its ascend, at which point a global index is incremented so that the robot controller won't get confused when two balls are moving at once. Once the index is incremented at any time, the switch state is activated and the object state estimate data is sent to a trajectory calculator function (described later) which calculates the final position of the ball. This final position is then given to an IK solver (described later) that returns the joint angles required to get the robot controller to that position. These joint angles are then sent to the controller using either setMilestones or appendLinear (depending on if we want to maximize number of balls blocked or number of points earned in the game).

f) Potential failures: There are some flaws in my planner in that it doesn't fully cover every possible scenario, but does a good job of working for about 90% of trial runs. Some of the times it fails are when it returns joint angles that cause the robot to collide with terrain (happens very rarely because of some bounds I will explain later), or if the object state estimates themselves are not very accurate, at which point the joint angles given to the controller will not always work.

C. Trajectory Calculator (sub-component of Planner)

a) Purpose: To take in position and velocity from a single object state estimate and use them to calculate the final expected position of the ball including bouncing.

b) Input: position 3D vector (floats), velocity 3D vector (floats)

c) Output: position 3D vector (floats) of the final expected position of the ball

d) How is it invoked?: It is invoked once every time the planner moves to the "switch" state

e) Implementation: This function is implemented recursively (to work with bounces). I wanted my robot to hit the ball at -1.75 in the x plane in world coordinates order to minimize terrain collision and joint velocity problems. Therefore, I needed to calculate

when the ball would hit x = -1.75. To do that I used the following equation (all of these are basic manipulations of kinematics equations):

$$t = (xPosition+1.75)/abs(xVelocity)$$

Then, I calculated the final position (3D vector) using this equation and plugging in:

$$S_f = S_o + V_o t + 0.5at^2$$
$$FinalPositions\ (3D) = InitialPositions\ (3D) + InitialVelocities\ (3D) * t + (1/2)(-9.8)(t^2)$$

This would be the final position vector I needed if there was no bouncing, but because of bouncing I needed to check if the z-value in FinalPositions was less than 0.102 (ground) at this location. If it was, I needed to calculate further steps. I now needed to know where the ball hits the ground so that I can recursively put that position and those calculated velocities back into the function to recalculate the trajectory until it hits the x = -1.75 plane. To do so I would first need to calculate the time it requires the ball to reach its peak height via the following calculation:

$$V_f = V_o + at,\ V_f = 0$$
$$t = -zVelocity/-9.8$$

With t, it is now easy to calculate the peakHeight using the same formula used before but instead with peak height time, and in this case we only want the z position:

$$S_f = S_o + V_o t + 0.5at^2$$
$$FinalPositions\ (3D) = InitialPositions\ (3D) + InitialVelocities\ (3D) * t + (1/2)(-9.8)(t^2)$$
$$FinalPositions[2] = zPosition$$

Now, to prevent the program from recursing infinitely when a ball starts bouncing very low, I have a check to see if the zPosition is greater than 0.108 (slightly higher than ground). If not, we return the original FinalPositions calculated before we even considered bouncing because its not worth it for such a minute difference in end position). If it is higher than 0.108 however, we continue by calculating the time it will take for the ball to fall to the ground from its peak height location:

$$S_f = S_o + V_o t + 0.5at^2,\ S_f = 0,\ V_o t = 0$$
$$t = sqrt[(2*(0.102-peakHeight))/-9.8]$$
$$totalt = t + peakHeightTime$$

Now that we have the total time required for the ball to reach the ground from its starting position, we simply use the same equation again to calculate the final position:

$$S_f = S_o + V_o t + 0.5at^2$$
$$FinalPositions\ (3D) = InitialPositions\ (3D) + InitialVelocities\ (3D) * t + (1/2)(-9.8)(t^2)$$

Now, in order to use this new position in the function we need to also calculate the new velocities using the restitution coefficient (k) of the ball (which was found to be 0.9, but 0.7 ended up working best in practice):

xVelocity = k*initialXVelocity
yVelocity = k*initialYVelocity
zVelocity = -9.8*k*(t-peakHeightTime)

The new position and velocity values were then recursively sent to the same function, which would continue this process until the ball hit the x = -1.75 plane, at which point the finalPosition is returned.

f) Potential failures: There are some flaws in my trajectory planner which appear in very rare circumstances. If the x velocity of the ball is very high compared to the other velocities, the bounce predictor won't be very accurate because the restitution value was tweaked a little to help it better run with the rest of my controller. Additionally, if the ball bounces too early, the bounce calculator will eventually get an incorrect estimate because slight errors in calculations will continually add up recursively until they get big. To create a fix to some of these problems, I set a counter that can be tweaked to set how many maximum bounces the function should calculate, so that if there are too many it should simply return a z value that is 0.104 (barely above the ground), so that the rest of the robot arm can hopefully block the ball.

D. IK Solver

a) Purpose: To take a point from the trajectory calculator and set it as the end effector position for the last link of my robot. Once an objective is created, the goal is to create a configuration vector (joint angles) that can be set for the robot controller to use.

b) Input: point in world coordinates (x, y, z floats)

c) Output: No returned values. A global variable "self.qdes" is set to equal the robot configuration (joint angles) that were calculated.

d) How is it invoked?: It is invoked once every time the planner moves to the "switch" state

e) Implementation: Most of the steps are taken directly from the available functions in the klampt IK class. First, the end effector point of the robot was calculated using guess and check with kviz. This was then used to create an ik objective using the following line, constraining the end effector to the point given:
obj = ik.objective(self.links[linkIndex],local=(0.0,0.142,0.0285),world=(point[0],point[1],point[2])).
ik.solve_global was then used to solve the ik objective with iters=1000, tol=1e-3, and numRestarts=50. Once the ik was solved, the self.qdes was set to equal self.robot.getConfig(), which was a set of joint angles.

f) Potential failures: If the point is not in range of the end effector of the robot, this code would return an IK failure and cause the robot to move to an unspecified location which could cause terrain collisions. Additionally, if the point is too close to the base of the robot an IK failure would not be reported, but the robot would cause a terrain collision.

E. Controller (Low-Level)

a) Purpose: To take in a robot configuration (joint angles) and move the robot accordingly in a specified manner to prevent joint velocity and torque errors.

b) Input: Either robot configuration joint angles (7D vector of floats only, or joint angles and dt (float) depending on if setMilestones or appendLinear is used respectively.

c) Output: No returned values.

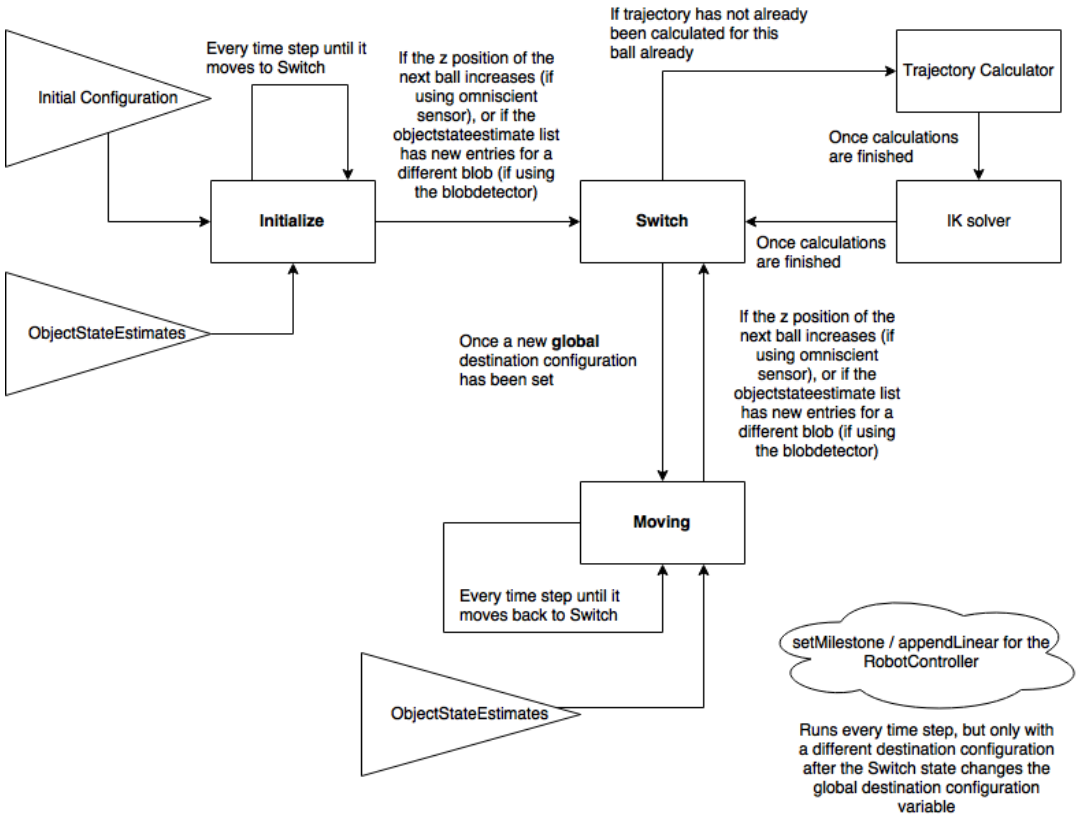
d) How is it invoked?: It is invoked every time step, but joint angles are only different when the finite state machine in the planner is changed to "switch"

e) Implementation: This is implemented exactly as was given already in the code. The only differences are which function is chosen and when the configuration joint angles are given. Currently the system is using setMilestones to prevent joint velocity out of bound errors, but this causes not all balls to be defended because the robot moves too slowly. If the goal is to block all balls, setMilestones can be changed to appendLinear with dt as the time step. Additionally, another specification is that my code only gives the controller configuration joint angles once every time it needs to switch for another ball - this makes the transitions faster and smoother.

f) Potential failures: Since all movements are left up to the joint angles given and what the base implementation of the controller applies, there are some scenarios when terrain collisions occur or the robot gets stuck near the goal, but it all depends on where the balls are thrown to make the system move into those locations. Nevertheless, errors in the controller don't happen that frequently.

III. Planning and Control Strategy

My planning and control strategy followed a very basic finite state machines with several sub-states. I have created the diagram below to help assist in my explanation. I will give a detailed explanation of the state machine without the additions of planning failure checks, and reserve that for a separate section. Note: bold blocks are the states of the finite state machine in the diagram below.



Initialize State
Upon reseting the program, initial configuration joint angles are changed in "Initialize" so that on the next time step the robot moves into that configuration before any balls are thrown. Also in the initialize state, ObjectStateEstimates are being read every time step to see when to move the state machine into the switch position. The state will change from initialize to switch when the z position of the next ball increases past 0.103 (ground) when using the omniscient sensor because it has access to the positions of all the balls. The state will change when using the blob detector when a new color ball is detected as moving.

Switch State
The finite state machine only stays in the switch state until the IK solver is complete, ensuring that the Robot Controller doesn't get overloaded with too many configurations. Once in the switch state, the robot checks to see if the trajectory for this color ball has already been calculated - if it has then it won't proceed and simply use the same previous configuration. If a new trajectory needs to be calculated, the state machine sends the position and velocity vectors to the trajectory calculator (explained in greater detail in the previous section) which returns the end position of the ball at the x = -1.75 plane. This end position is then given to the IK solver, which sets the global robot configuration variable to a new value. Once the IK solver has returned from its call, the state machine moves into the "Moving" state.

Moving State
The moving state is very similar to initialize except that it doesn't set an initial configuration. When in the moving state, the same checks are being completed as initialize to see when the state machine should move back to the switch state.

Note: setMilestones or appendLinear are called every time step but only have different configurations once the switch state is complete.

Handling Planning Failures
There were several failures that could have occurred in planning, for which I implemented some quick checks to prevent them:

1. If a ball is blocked but then it hits the rest of the balls before they could be launched, the system never throws those balls so the indexing of which ball is next would get ruined in my planner when using the omniscient sensor
Solution: Check to see if any ball in the range from the current one until the last one moves in addition to checking if the next indexed ball moves to make sure the planner can sense any ball moving, regardless of order for the most part.

2. When the balls are initialized, they start at a position of 0.15 on the z axis and then drop to the ground. This ruined my planner because it was constantly checking if balls were moving above ground, and if it senses that all balls are above the ground the planner no longer knows what to do.
Solution: Only check if positions of balls are higher than ground if total time of program execution is past 1.0 seconds (since that is how long it takes before the first ball is thrown anyways). To calculate global time, I simply created a global tie variable and incremented it by dt every time step.

3. IK failures - on occasion I do get IK failures but they tend to be rare. When they do happen however, I have no implemented any sort of solution because I found that the robot tends to move in the general correct position anyways - still successfully blocking the ball.

There were no other errors that my planning system encountered, and if there are possibly in the future, there are no current checks in place to prevent the system from running for that instance or any solution of that nature. I found that any minor errors such as slight errors in trajectory calculations or IK failures tend to still work because the robot moves in the general correct position, and since it only needs to hit the ball away from the goal it doesn't need to be exactly correct.

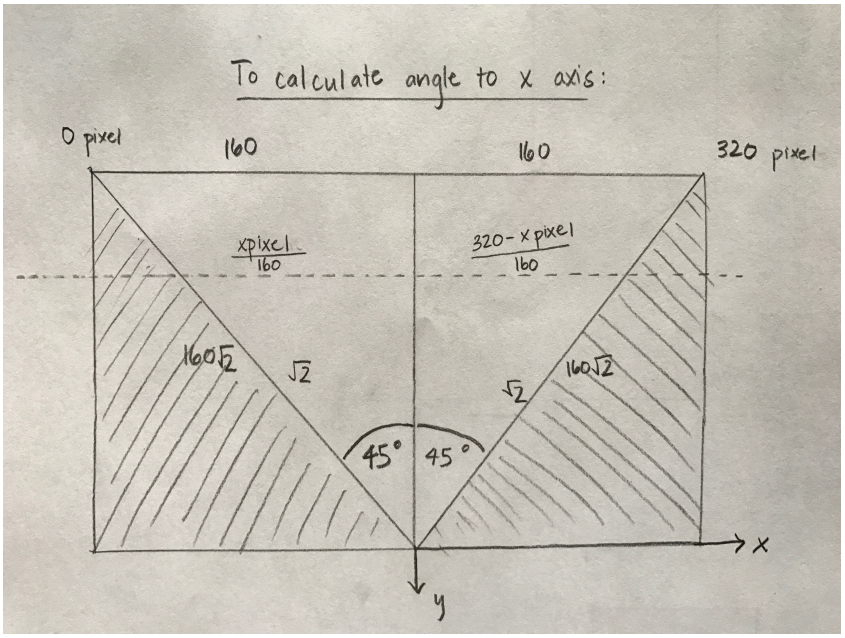
IV. Perception Strategy

My state estimation component was essentially a converter of camera sensor outputs to world coordinates and velocity calculator, as position and velocity were the only pieces of information that my planner needed. Not much more was needed, such as filtering, because I found my position estimates to be relatively accurate and since my task was only to hit the ball away using any part of the robot, accuracy was good enough. In the following steps I will explain how I converted camera sensor output into world coordinates.

1. Calculating the distance of the camera to the detected blob:

$$\text{FocalLength} = \tan(0.5 * \text{FieldofView}) * \text{totalPixelWidth}/2$$
$$\text{Distance} = (\text{FocalLength} * \text{blobActualRadius}) / (\text{blobSeenWidth}/2)$$

2. Calculating x and y angles to solve for distance to the z plane from the camera. Noticing that the field of view is always 90 degrees, I drew out the field of view and performed some trigonometry. First, I needed to calculate the blob's angle from the x-axis:



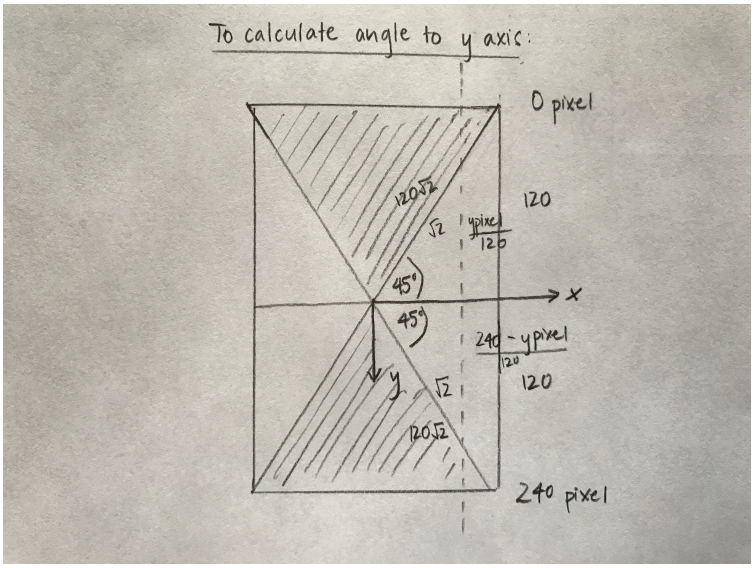
I created a ratio that relates the 45-45-90 degree triangle with 320 pixels wide to a triangle that could vary with z-depth (which the ratios would be multiplied by later). Therefore, the angle from the x-axis would be calculated as:
If xpixel $<$ 160:

$$\text{arcsin((xpixel/160)/sqrt(2))} + 45 \text{ degrees}$$

If xpixel $>$ 160:

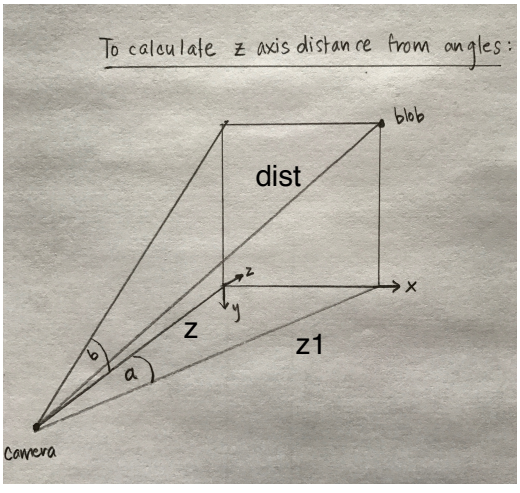
$$\text{arcsin((320-xpixel/160)/sqrt(2))} + 45 \text{ degrees}$$

To calculate the y angle, a similar calculation was performed, but instead using the max y resolution of 240 instead of 320, and using ypixel instead of xpixel. The diagram below shows how they are similar:



With x and y angles calculated, the z-axis depth could be calculated by simply multiplying distance to the blob by cos(x angle) and cos(y angle). The diagram below depicts why that is possible:

Let: a = x angle we calculated previously
b = y angle we calculated previously



It is easy to see that the segment from the camera to the blob (dist) multiplied by cos(b) would give us z1, and z1 multiplied by cos(a) gives us z. Therefore:

$$z = dist * \cos(a) * \cos(b)$$

Now that we have z-depth we can simply multiply that with xpixel/self.w to get the x coordinate, and ypixel/self.h to get the y coordinate. These x,y,z points were then converted to world axes using se3.apply(Tsensor, (x,y,z)). The world coordinate points were checked and they seemed accurate enough for my system to operate with, since my task was only to hit the ball away so it didn't need to be too perfect.

Velocity
Now that we have the position of the blob in world coordinates, we can simply calculate velocity by waiting until two entries have been appended to the list (which is being continually updated every time step as a global variable). Once there are two, we can subtract the new from the old to get the distance travelled, and divide that by dt (time step). These velocity estimates were somewhat accurate but there were times when they became negative when position readings returned errors. If the planner picked up the point at those negative velocities, then it causes problems with the planner, but this happened rarely - still accurate enough for the system to run somewhat decently.

With velocities and positions calculated, I created an ObjectStateEstimate object with the color of the ball and a 6D vector of the positions and velocities. For each of the blobs being seen by the camera, I created a new object and put together a list which was then returned by the state estimator as a MultiObjectStateEstimate.

*Note: When running the state estimator with my planner, it may seem like the robot is moving opposite of where the ball is actually heading, but that is only because the robot moves too slowly and I am telling it to get the ball at a distance x = -1.75 on the x axis. Therefore, the robot is moving in the opposite direction because it is trying to get to that -1.75 point even though the ball has already passed that part so it seems like the system is not working correctly, but it is just based on the parameters i've input. I decided not to change the x position that the robot tries to get the ball at because it was optimized for the omniscient sensor, so I didn't feel comfortable changing it since the parts are being tested separately. Additionally, the final traces that the program leaves (which are shown to be very inaccurate) for the state estimates can be ignored because only one reading is taken in the middle of flight path.

V. Reflection

As a whole I thought my system was quite successful when tested separately. In other words, when using the omniscient sensor, my planner/control was very good at blocking the balls. However, when my state estimator was connected, the robot was no longer able to block any of the balls, partly because I think it was slow to receive readings as well as velocities not being accurate enough for the trajectory calculator to get a good enough end position. If I could improve any part of the system, it would definitely be the state estimator by using a Kalman filter perhaps to increase the accuracy of the velocity estimates so that my robot can both move into the position faster and move into a

position that is slightly closer to the balls so that it doesn't miss them. This project was definitely harder than I anticipated in the implementation portion. There was a steep learning curve in terms of understanding how the different classes interacted with each other and what each of the different objects were. It was also quite difficult finding the kinks when something wasn't working in the implementation that required a lot of trial, error and observation.

If I had to design this robot in real life as opposed to a virtual implementation, there would be a lot more factors to consider and areas for problems to arise. Problems could arise from physical hardware failures, joint velocities/torques would have to be more closely watched, terrain collisions would be a disaster because the robot could get stuck in a position, camera sensors may have a lot of noise, and conversion from camera coordinates to world coordinates would be harder to calculate because the camera could move if a ball hit the goal post or the camera itself. With so many more factors to consider as well, when failures in the implementation occur it would also be even harder to debug since there would be so many more possibilities to check.