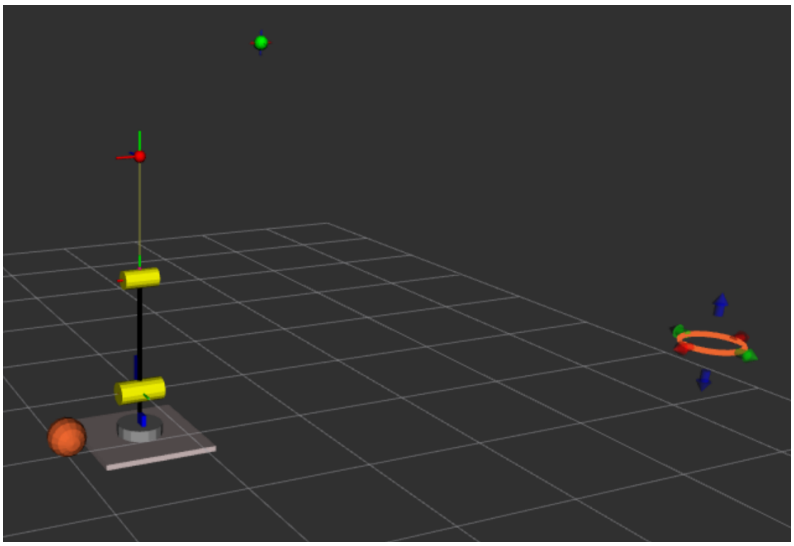
ME/CS 133a Final Report

Kyle McCandless, Daniel Wen

Basket “Never misses except at singularities” bro

1. **Overall Description / Introduction:**

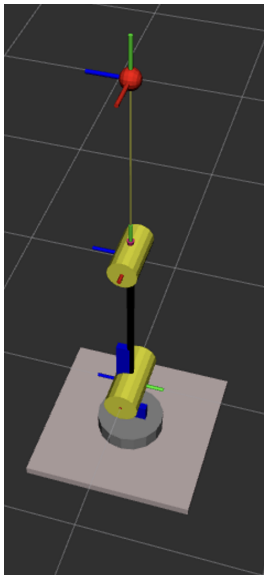
Our final project was to create a robot arm (Basketbro) that shoots a basketball into a hoop given a hoop location and a release point, both of which are defined by the user. Since we implemented joint position limitations onto the robot arm, there are certain requirements, such as the fact that the release point must be within the workspace of the arm. Additionally, since we also implemented joint velocity limitations, there are implicit requirements as well. For example, if the hoop is too far away or the configuration of the release point and hoop are infeasible to shoot the basketball within the limitations, then the robot will simply miss. We are using RVIZ to simulate our robot. However, our final project deals with the interactions between different objects, as the robot must pick up the basketball and then release it through the shooting motion. In order to account for this, we added flags that would simply bind the basketball to the tip of the robot arm when the basketball is picked up, then release the basketball from the tip when the robot reaches the release point.

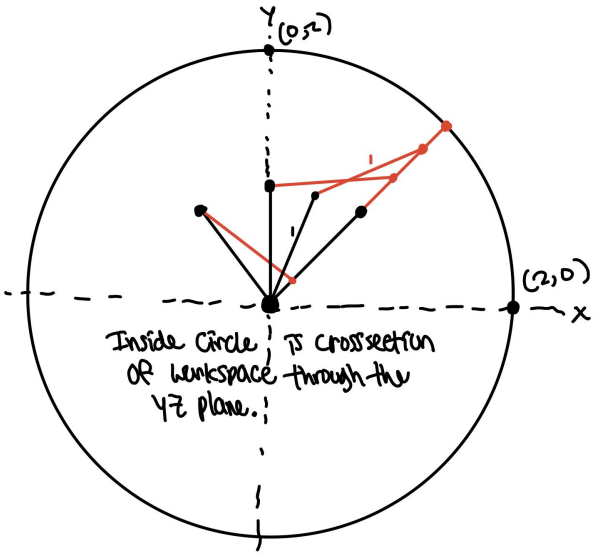


Our final project works as follows: in the environment, there is the robot arm, an orange ball signifying the pile of basketballs, a green point signifying the robot’s point of release, and a hoop that the robot is aiming for. At the beginning, the arm starts in a home “base” location and is not moving. Both the release point and the hoop locations are free to move and to be defined by the user. The user can drag these markers within the RVIZ environment. Then, once the user is satisfied with those locations, they can press the Enter key. The robot listens to keyboard inputs which will start the robot’s motion. The robot’s motion and tasks can be broken up into

separate intervals. For the first 3 seconds, the robot arm will spline to the pile of basketballs and pick it up. For the next 3 seconds, the robot arm will spline to the release point and release the basketball, ending with the desired velocity that will result in the basketball going into the hoop. Finally, the robot arm will follow through and return to its home position, ready for the next time it is called to throw the ball into the hoop.

2. **Describe your robot/system:**



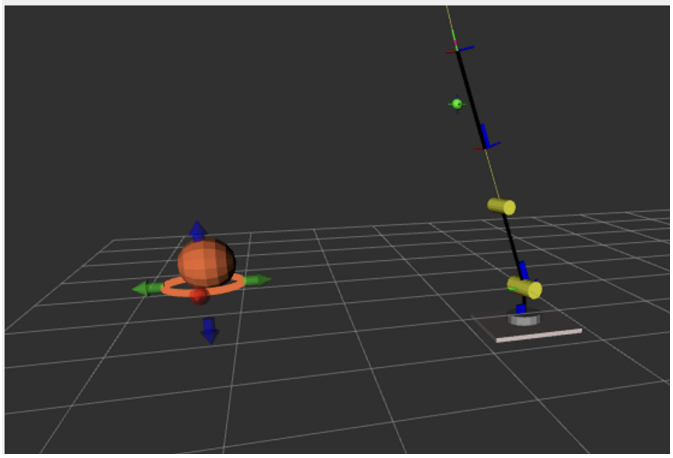


We use a 4DOF Robot that is in the shape of an arm (above left). Going from the fixed base to the tip, it has a pan rotating joint (gray), then a tilt rotating joint (yellow), followed by a prismatic joint (black rod) in the upper arm, and finally another tilt rotating joint (yellow). The upper arm (closer to base) is 1m long when the prismatic joint is at 0m, and the lower arm is fixed at 1m long. We chose these lengths because they allow us to have a full spherical workspace minus singularities, including the points near the center of the workspace which would be unreachable if the arms were unequal length (more on workspace below).

We run the robot in two modes: with the prismatic joint locked, and without. When the prismatic joint is not locked, we set a maximum length of 3m, making the workspace a 5m radius sphere centered at the first rotating joint. When the prismatic joint is locked, the workspace is a 2m radius sphere centered at the first rotating joint (above right). To make the movement slightly more realistic, we also constrained all of the rotating joints to stay in the interval [-pi, pi].

In all of our various debuggings and because we accept user input for the release point of the basketball, we explored a lot of the redundancies of this robot that we learned about in class. On different throws, we have seen the pi + theta_pan redundancy, the negative prismatic joint redundancy (when it is not limited), and different elbow up / elbow down configurations with the final rotating joint. Because we send the robot to a “home” joint configuration after each throw, any redundant configurations we end up in do not persist. This was a huge benefit of operating in joint space instead of task space.

3. **Describe your task.**



The task is to shoot a ball into a given hoop location from a given release point, under joint position and velocity limitations. Since we don't care about the orientation of the ball, there are three task coordinates: (x, y, z). We achieve the task in two steps. First, we must calculate the desired velocity of the ball at the release point using kinematics. Then, since the ball stays on the tip until the release, we need to do inverse kinematics to figure out how to get to the release point with that velocity as the final tip velocity. To visualize the task, we wrote code to stop the ball once it is in the hoop plane, but only if it is near enough to the hoop that it will go in (see above).

Because we have a 4DOF robot and a 3DOF task, we played around with adding a secondary task to move the joints towards their "home" positions, but we found that this made the motions unnatural looking (see plots in section 6 and video). Instead, we explored optimizing the robot for efficiency: choosing a throwing velocity that was as small as possible, and heavily limiting the joint velocities to force the robot to be efficient (see analysis in section 6).

Finally, the task of throwing the ball is not always achievable, for two main reasons. First, the release point could be outside of the robot's workspace. We dealt with this by including a max number of iterations in our Newton Raphson loop, and informing the user that the release point cannot be reached if we reach that max number of iterations without convergence. Second, the hoop could simply be too far away from the release point for the throw to be possible with the set maximum joint velocities. Then, the robot will just throw as well as it can and the shot will miss. In both of these cases, it is realistic for the robot to fail because we are asking it to do something that is impossible, so we decided to make sure the code doesn't throw any errors and allow the user to reselect the release point and hoop position. We talk more about this in section 5.

4. **Algorithm and Implementation:**

Basketball Kinematics:

The basketball has four different states. First is the 'ground' state, where it represents a "pile of balls" on the floor that the robot picks up with its magnetic tip. In this state, the basketball just publishes a constant position and 0 velocity. The second state is the 'holding' state, where the ball has the same position and velocity as the tip of the robot. Immediately after the 'holding' state is the 'free' state, where the ball is in free fall. Each dt, we integrate its velocity with the gravity acceleration vector g = [0, 0, -9.81m/s^2] and its position with its velocity, causing smooth parabolic motion. The last state is the 'stuck' state, when the basketball is frozen in space for some time if it is in the horizontal plane of the hoop and its x and y coordinates are within 0.1 of the hoop. We bumped up the frame rate to 100 frames per second and the basketball flies much smoother.

Desired Velocity Calculation:

For **p**_hoop = current hoop position, **p**_0 = release point position, and **v**_0 = desired initial velocity, we have **p**_hoop = **p**_0 + **v**_0 * t + 0.5 * **g** * t^2. This gives us **v**_0 = (**p**_hoop - **p**_0 - 0.5 * **g** * t^2) / t. There is more information on how we played with this calculation in section 6.

Robot Kinematics:

Once the kinematics of the basketball are set, then we implement the kinematics for the robot. We decided to implement trajectories in the joint space, since this would best simulate an accurate throwing motion, as throwing is usually done in an arc rather than in a straight line. As shown in the video, we also implemented the trajectories in task space, and found that they looked a little bit unnatural. The actions of the robot are multiple actions concatenated together, with splines to make the actions appear smoothly after one another. The actions can be broken down into the following: robot picks up the ball, robot throws the ball by reaching the release point with the precalculated desired velocity, robot follows through and returns to home position. After picking up the ball, the ball will be bound to the tip of the robot arm. After reaching the release point, the ball is no longer bound to the tip of the robot arm and the ball starts following its own free-fall kinematics. Each action lasts 3 seconds.

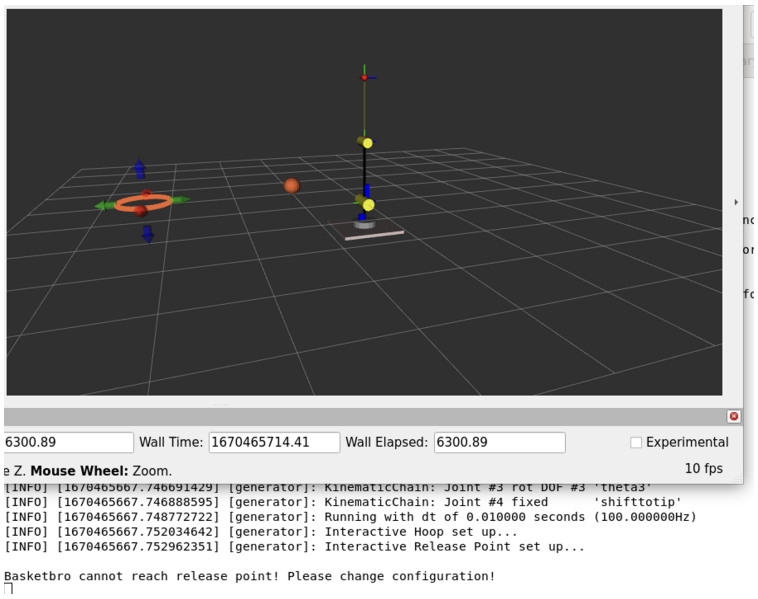
We are given the positions of both the pile of basketballs and the release point of the basketball. We also preset the velocity at the basketball pile and and previously calculated the velocity at which tip of the robot arm must be moving upon reaching the release point, which is equal to the initial velocity of the basketball. However, the position and the velocity are given in the task space, not the joint space. So, in order to convert these to joint space, we must implement inverse kinematics. We can convert the positions in task space to joint space using the Newton-Raphson algorithm. We use an initial guess that is equal to the joint values at the home position, as we probably want to find a solution close to those values. We keep iterating and updating our guess until the error vector is sufficiently small. Once we have that, then we can simply use the fact

that $\dot{q} = J^{-1}(q)\dot{x}$ to find the target joint velocities at each location. From there, we can create splines in joint space.

However, we also must implement joint position and velocity constraints on our robot. It is quite possible that the splines may take us out of these limits. So, in order to take these into account, we would set $\dot{q}$ equal to what the spline calculated, then set for each index i in $\dot{q}$, $\dot{q}[i] = \max(\dot{q}_{min}, (\min(\dot{q}_{max}, \dot{q})))$ to ensure that each joint velocity was within its limits. Then, we would increment q by this new $\dot{q}$, and do the same procedure with q, but with the joint position limits.

This would ensure that the joint positions and joint velocities would stay within their respective bounds. But, if we were to continue to follow the predetermined spline calculations even if we were veered off path by the limitations on the joints, then the final position and velocities would not be accurate. In order to mitigate this, we would have to recalculate the spline based on our current position and velocity, with a decreasing spline duration for each timestep. This way, if a joint gets veered off path due to its limitations, it can readjust by recalculating the spline at every time step based on its current position and velocity.

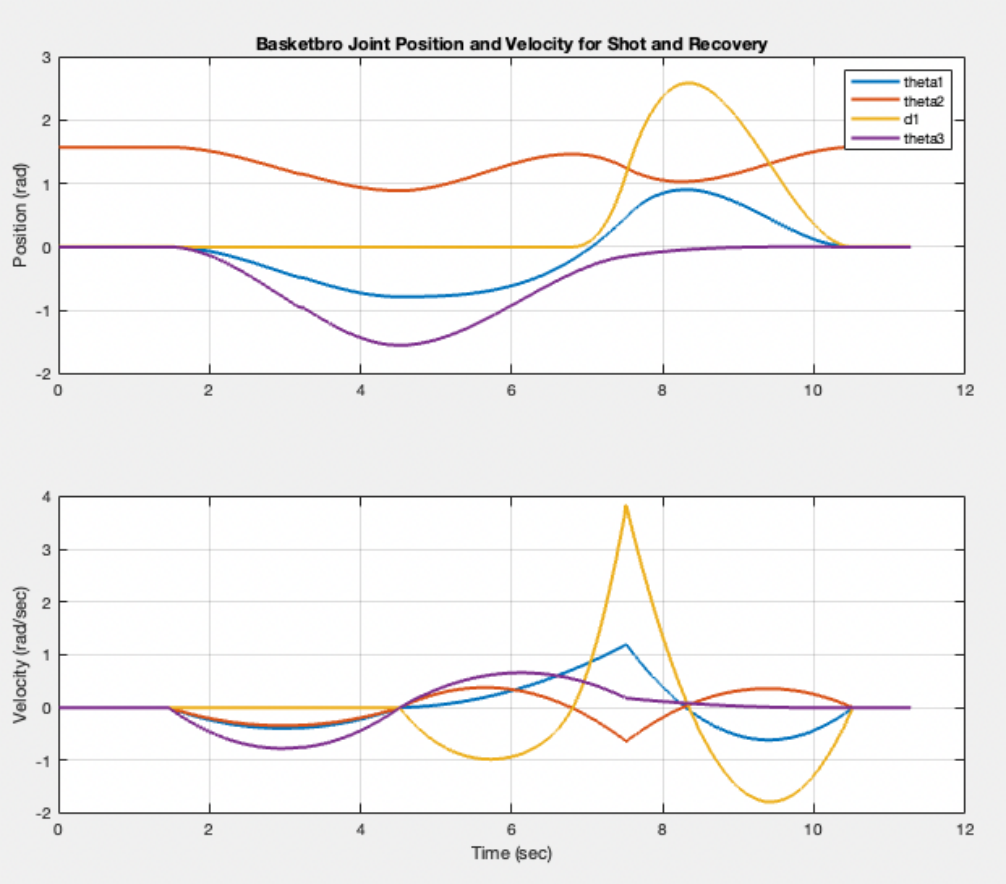
5. **Particular Features:**



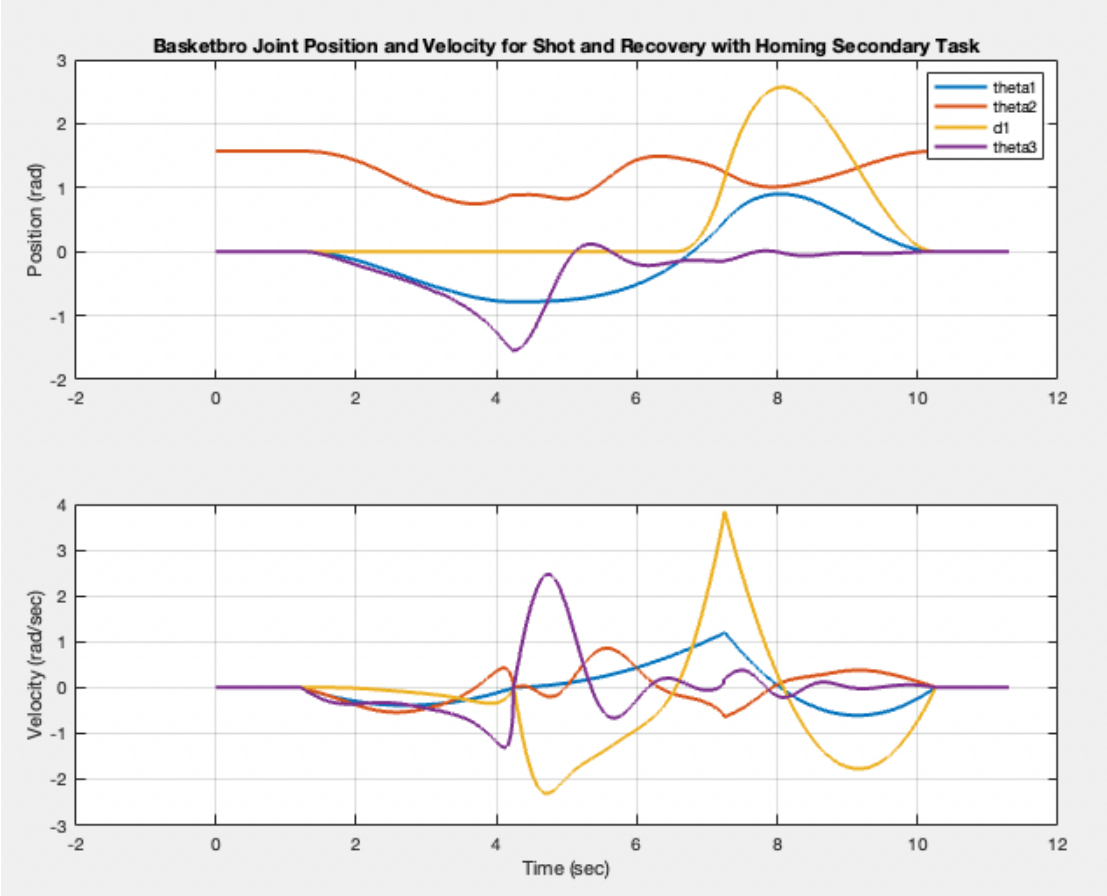
Some of the special features that we have in our robot project is the amount of user interactivity that it possesses. The user is able to customize many integral parts of the environment, such as the locations of both the release point and the hoop. We thought it was really cool to pick a random hard looking configuration and see the robot nail the shot. However, this also leads to problems, as the user may define unsolvable configurations. For example, the user may place the release point of the ball outside of the work space of the robot arm, or the trajectory that the robot must travel in goes through a singularity. In those cases, our program must be able to detect such problems. One way that we do this is through Newton Raphson. If our error does not converge within a reasonable amount of iterations, then we can assume that the desired tip position is not achievable given our joint limitations. Or, if the inverse of a Jacobian is not invertible, then we have arrived at a singularity. In these cases, we can simply reset the environment by setting the robot arm back to its “home” position, then printing out a message that tells the user that the current configuration is not achievable and requesting them to modify it.

One cool thing we would try if we had time would be implementing avoidance of singularities. Professor Niemeyer talked about it in class, but we are not sure how it would work. Maybe we could include a primary task to avoid certain points and balance it with our original task.

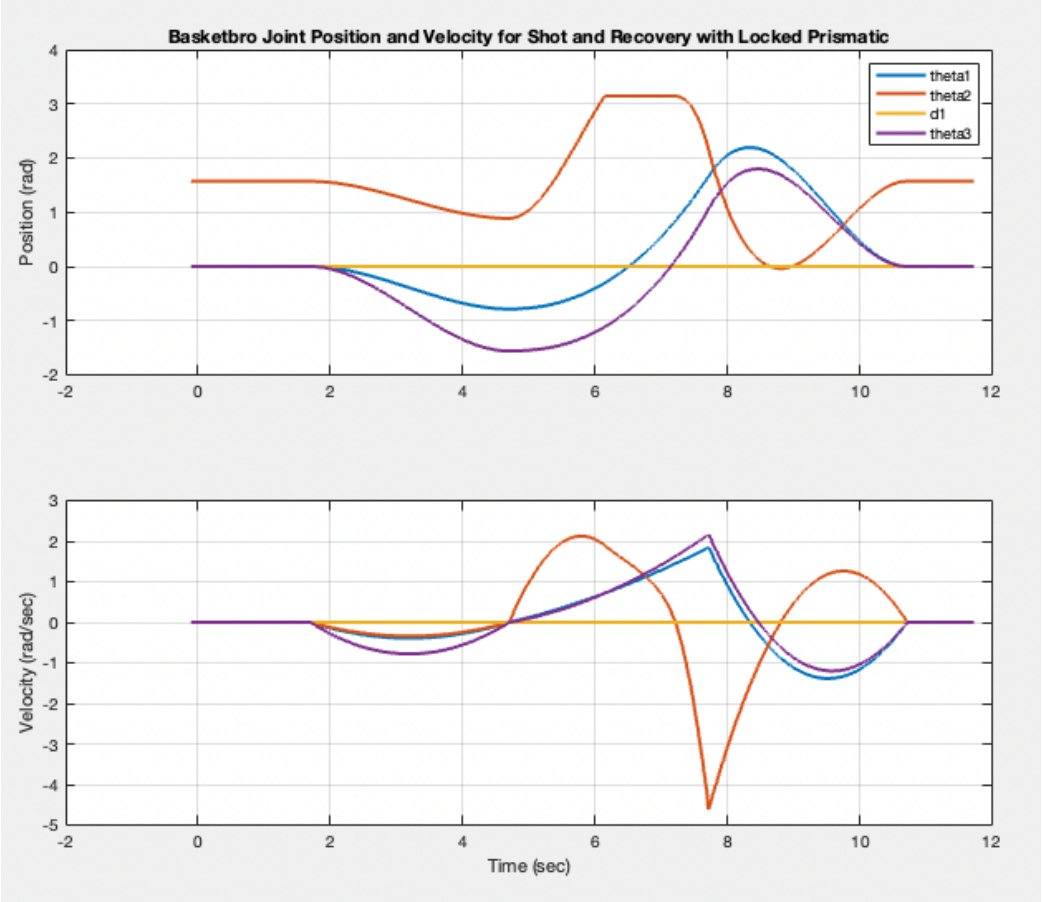
6. **Analysis, Plots/Images, or other Helpful Material:**



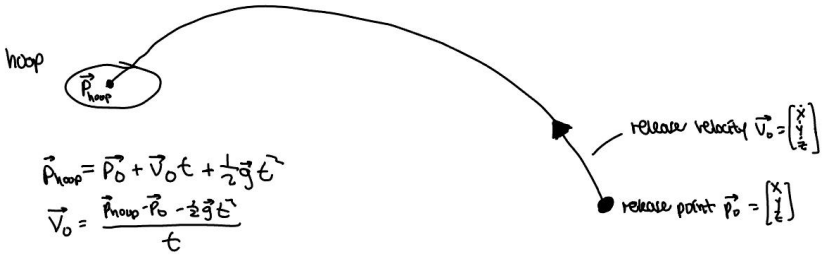
Above is a plot of a very standard shot for Basketbro. We can see that Basketbro uses the prismatic joint heavily when shooting, giving it the shot put like motion we saw in the video. We can see where Basketbro releases the ball and begins going back to the home position, because that is a corner that is a transition between splines. Because the velocity is still continuous and we are dealing with simulated robots, we focused on singularities and locking joints rather than choosing to deal with this. A cool expansion goal would be to switch to quintic splines to keep the acceleration continuous as well.



As mentioned in section 3, we played around with adding a secondary task to keep joints towards the home position. When comparing these plots to the previous ones, we see some slight changes that move the joints towards the home position. Qualitatively when watching this motion, we thought it looked less natural, so we decided to get rid of this secondary task.



This was one of our favorite results of Basketbro. Because we have a 4DOF robot with a 3DOF task, we locked the prismatic joint that Basketbro had been relying on heavily to make shots. We included a portion in the video, but what we can learn from these plots is that quantitatively the prismatic joint is indeed locked, and that the robot relied heavily on theta2 for this particular shot instead of d1. When locking the prismatic joint for the first time, we got a good look at handling singularities and points outside of the workspace (section 5), because both the ball pile and the release point we configured were outside of the new workspace. These were our personal favorite shots, because the robot always had very interesting configurations when shooting.



$$\vec{p}_{hoop} = \vec{p}_0 + \vec{v}_0 t + \frac{1}{2}\vec{g}t^2$$

$$\vec{v}_0 = \frac{\vec{p}_{hoop} - \vec{p}_0 - \frac{1}{2}\vec{g}t^2}{t}$$

Naive approach: Fix parabola Time t and plug in to get $\vec{v}_0$

Efficient approach (attempt): Use calculus to pick minimum $|v_0|$

$$v_x = \frac{hoop_x - release_x}{t} \qquad v_y = \frac{hoop_y - release_y}{t} \qquad v_z = \frac{hoop_z - release_z + \frac{1}{2}\cdot 9.8\cdot t^2}{t}$$

min of $v_x^2 + v_y^2 + v_z^2 = (hoop_x - release_x)^2 t^{-2} + (hoop_y - release_y)^2 t^{-2} + (hoop_z - release_z + \frac{1}{2}\cdot 9.8 t^2)^2 t^{-2}$

derivative $t^{-2}\left((hoop_x - release_x)^2 + (hoop_y - release_y)^2 + (hoop_z - release_z + \frac{1}{2}\cdot 9.8 t^2)^2\right)$:

$$t^{-2}\left(9.8t\left(2(hoop_z - release_z + \tfrac{1}{2}\cdot 9.8 t^2)\right)\right)$$

$$+ -2t^{-3}\left((hoop_x - release_x)^2 + (hoop_y - release_y)^2 + (hoop_z - release_z + \tfrac{1}{2}\cdot 9.8 t^2)^2\right) = 0$$

$$2\cdot 9.8\, t^{-1}\left(h_z - r_z + \tfrac{1}{2}\cdot 9.8 t^2\right) + -2t^{-3}\left((h_x - r_x)^2 + (h_y - r_y)^2 + (h_z - r_z + \tfrac{1}{2}9.8t^2)^2\right) = 0$$

$$19.6 t^2 (h_z - r_z + 4.905 t^2) - 2(h_x - r_x)^2 - 2(h_y - r_y)^2 - 2(h_z - r_z + 4.905 t^2)^2 = 0$$

Four solutions, pick positive and real:

$$t_{opt} \approx 0.225762\sqrt{4h_z^2 - 8h_z r_z + 4h_z + 4r_z^2 - 4r_z + 8h_x^2 - 16h_x r_x + 8r_x^2 + 8h_y^2 - 16h_y r_y + 8r_y^2 + 1} - 2h_z + 2r_z + 1$$

$$\vec{v}_{0_{opt}} = \frac{\vec{p}_{hoop} - \vec{p}_0 - \frac{1}{2}\vec{g}t_{opt}^2}{t_{opt}}$$

Finally, we explored the idea of "efficiency" for Basketbro. As mentioned in section 3, we solved the equations to choose the time of flight for the ball that would minimize the magnitude of the release velocity. After playing around with this, we realized that if the hoop is above the release point, the minimum velocity solution uses a path that reaches its apex right at the hoop position. Intuitively, we thought this makes a lot of sense, because given two points and a path, as t increases, the apex just gets higher and higher which costs a lot of energy and initial velocity (image at right). However, what this practically meant for us is that the optimal energy path isn't correct to kinematics: the ball always passes through the hoop if the hoop is higher than the release point. Since it would require a lot of complicated kinematics to implement the ball-hoop interactions correctly, we chose to focus on the robot and stuck with the naive velocity formula in section 3.