# Final Project

## Introduction

Throughout the labs we have completed this semester, we observed that the built-in feedback control of the servo motors was not sufficient for many of the applications that the Lynx could be used for. There are many applications in which you desire variable speeds and greater positional accuracy for a robotic arm. For example, almost any joint configuration of the Lynx had a noticeably strong gravitational effect. As a result, our goal for this project, was to implement dynamic control and PID feedback control in software to create a more robust Lynx system.

Earlier this semester, we enjoyed working on the Rapidly-exploring Random Tree (RRT) planner but saw the opportunity to implement improvements on it. Therefore, we explored kinodynamic programming to automatically compute smoother trajectories. More natural motion is useful for many applications, such as those that require precise and smooth movements.

## Simulated Feedback Control

### Methods

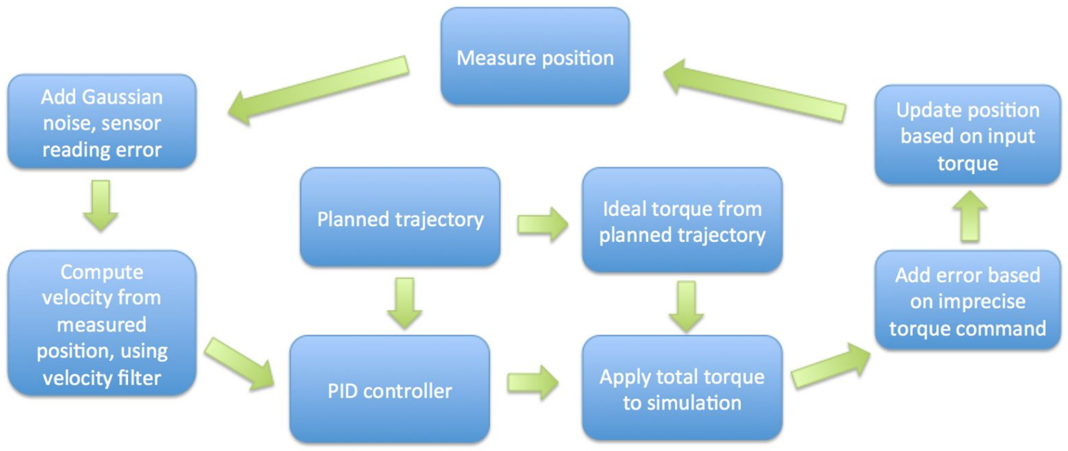

Measure position
Add Gaussian noise, sensor reading error
Update position based on input torque
Planned trajectory
Ideal torque from planned trajectory
Compute velocity from measured position, using velocity filter
Add error based on imprecise torque command
PID controller
Apply total torque to simulation

**Figure 1**. flowchart of control algorithm when algorithm is used in simulation

<u>Algorithm Overview</u>

The main script is controls_test.m script. It requires hardware_flag to be set to determine whether the noise will be manufactured by the program or if the noise will originate from the joint value readings from the physical robot. Our goal was to focus on implementing a robust system in simulation before moving onto involving hardware. For the sake of simplicity and being able to recognize the ideal positions, velocities, and accelerations, we chose to simulate sinusoidal motion over time as planned trajectory to follow. This allowed us to easily visualize what the planned trajectories should look like and any effects of noise on the actual trajectory. We simulated system and sensor noise using a fixed-magnitude Gaussian noise that was added to the applied torque and measured position. The standard deviation of the sensor reading magnitude can be found in table 1. This magnitude of error is fit for the magnitude of movement between ±1 radian. The theta, theta_dot (angular velocity), and integral error values are computed from the ideal and measured values and used for the PID controller. The component of torque added by the controller is combined with the amount of torque required to follow the desired trajectory. Gaussian error is added to the torque to simulate imprecise movement of the robot arm ( $Tau_{error}$ ); the standard deviation of this noise can be found in table 1. This total torque is then used to solve for the acceleration and velocity terms that will simulate the new simulated position. Figure 1 describes the steps of the algorithm in a flowchart.

**Table 1**. simulated error magnitudes

| | |
|---|---|
| Standard deviation of sensor reading noise magnitude | 0.01 radians |
| Standard deviation of torque command noise magnitude | 2000 * $10^{-6}$ Nm |

<u>Proportional-Integral-Derivative (PID) controller</u>
Theta is acquired, the method depending on whether we are running the code in simulation or with the physical robot. Next, the experimental velocity, the derivative of theta is calculated using a velocity filter (with weight = 0.6), as previously implemented in Lab 5. Next, the error between simulated and and ideal theta is calculated, and the integral error term is computed with a trapezoidal approximation which is added at each time step to the total integral error. The PID constant values were tuned by iteration; they are system dependent and can be tuned for different ranges of motion magnitude and frequency, as well as different error magnitudes.

**Equation 1**. PID controller calculation

$$Tau_{control} = K_p\,(error) + k_d\,(\tfrac{d}{dt}error) + k_i\,(area\ under\ error\ curve)$$

**Table 2.** tuned PID controller constant values

| | |
|---|---|
| $k_p$ | -10000.00 |
| $k_i$ | -800000.00 |
| $k_d$ | -1000.00 |

<u>Torque required for following trajectory</u>
To determine the torque required to move joints at desired speed, we need to consider the inertia of the physical links of the robot (M), the Coriolis and centrifugal forces (C), and any other outside forces. For our system, the only outside force we are considering is gravity (N).

The inertia of a rod depends on the axis of rotation; the coordinate frames are defined for each link's center of mass, modelled to be at the center of the ideally cylindrical link. For the axis that is aligned with the length of the rod, equation 2 is used to calculate the moment of inertia for rotations with respect to that axis. The moment of inertia about the other two axes will be calculated by equation 3, they are the same because a rod is rotationally symmetrical.

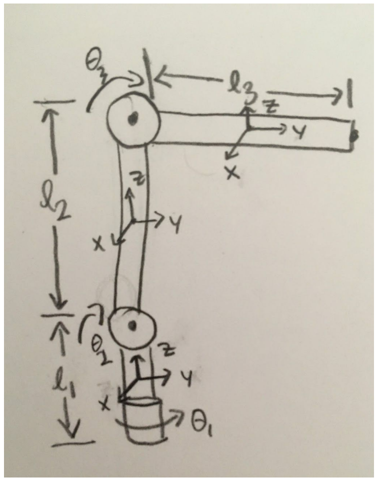


**Figure 2**. schematic displaying axis definitions for the links' center of masses

**Equation 2**. moment of inertia for rotations about the line through the center of the cylindrical links, parallel to its length, where m = mass, l = length, and r = radius of link

$$I_{\text{along length}} = \frac{mr^2}{2}$$

**Equation 3**. moment of inertia for rotations about axes that are perpendicular to the link, where m = mass, l = length, and r = radius of link

$$I_{\text{not along length}} = \frac{m}{12}(3r^2 + l^2)$$

**Table 3**. variable definitions for equation A

| | |
|---|---|
| $I_{pi}$ | moment of inertia about the p axis, of link i |
| $r_i$ | radius of link i |
| $c_i$ | $\cos(\Theta_i)$ |
| $c_{ij}$ | $\cos(\Theta_i) + \cos(\Theta_j)$ |
| $s_i$ | $\sin(\Theta_i)$ |
| $s_{ij}$ | $\sin(\Theta_i) + \sin(\Theta_j)$ |
| $l_i$ | length of link i |
| $m_i$ | mass of link i |

To calculate the moments of inertia, we use equation 3a; the elements of the matrix are described by equations 3b. The variables used in equations 3b are described in table 3. The equation for calculating the Coriolis and centrifugal forces can be found in equation 4 and 5. Lastly, the matrix used to represent the effect of gravity is described by equation 6. These matrices are calculated in the computeMNC.m file, which takes trajectory velocities and trajectory accelerations as inputs.

**Equation 3**. inertia matrix for each link in each axis

$$M(\theta) = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix}$$

a)

b)

$$\begin{aligned}
M_{11} &= I_{y2}s_2^2 + I_{y3}s_{23}^2 + I_{z1} + I_{z2}c_2^2 + I_{z3}c_{23}^2 \\
&\quad + m_2r_1^2c_2^2 + m_3(l_1c_2 + r_2c_{23})^2 \\
M_{12} &= 0 \\
M_{13} &= 0 \\
M_{21} &= 0 \\
M_{22} &= I_{x2} + I_{x3} + m_3l_1^2 + m_2r_1^2 + m_3r_2^2 + 2m_3l_1r_2c_3 \\
M_{23} &= I_{x3} + m_3r_2^2 + m_3l_1r_2c_3 \\
M_{31} &= 0 \\
M_{32} &= I_{x3} + m_3r_2^2 + m_3l_1r_2c_3 \\
M_{33} &= I_{x3} + m_3r_2^2.
\end{aligned}$$

**Equation 4**. Coriolis and centrifugal forces are defined as:

$$C_{ij}(\theta,\dot\theta) = \sum_{k=1}^{n}\Gamma_{ijk}\dot\theta_k = \frac{1}{2}\sum_{k=1}^{n}\left(\frac{\partial M_{ij}}{\partial\theta_k} + \frac{\partial M_{ik}}{\partial\theta_j} - \frac{\partial M_{kj}}{\partial\theta_i}\right)\dot\theta_k.$$

**Equation 5**. Matrix entries for the Coriolis and centrifugal forces

$$\begin{aligned}
\Gamma_{112} &= (I_{y2} - I_{z2} - m_2r_1^2)c_2s_2 + (I_{y3} - I_{z3})c_{23}s_{23} \\
&\quad - m_3(l_1c_2 + r_2c_{23})(l_1s_2 + r_2s_{23}) \\
\Gamma_{113} &= (I_{y3} - I_{z3})c_{23}s_{23} - m_3r_2s_{23}(l_1c_2 + r_2c_{23}) \\
\Gamma_{121} &= (I_{y2} - I_{z2} - m_2r_1^2)c_2s_2 + (I_{y3} - I_{z3})c_{23}s_{23} \\
&\quad - m_3(l_1c_2 + r_2c_{23})(l_1s_2 + r_2s_{23}) \\
\Gamma_{131} &= (I_{y3} - I_{z3})c_{23}s_{23} - m_3r_2s_{23}(l_1c_2 + r_2c_{23})
\end{aligned}$$

$$\begin{aligned}
\Gamma_{211} &= (I_{z2} - I_{y2} + m_2r_1^2)c_2s_2 + (I_{z3} - I_{y3})c_{23}s_{23} \\
&\quad + m_3(l_1c_2 + r_2c_{23})(l_1s_2 + r_2s_{23}) \\
\Gamma_{223} &= -l_1m_3r_2s_3 \\
\Gamma_{232} &= -l_1m_3r_2s_3 \\
\Gamma_{233} &= -l_1m_3r_2s_3 \\
\Gamma_{311} &= (I_{z3} - I_{y3})c_{23}s_{23} + m_3r_2s_{23}(l_1c_2 + r_2c_{23}) \\
\Gamma_{322} &= l_1m_3r_2s_3
\end{aligned}$$

**Equation 6**. Matrix to represent gravity force acting on the links

$$N(\theta,\dot\theta) = \frac{\partial V}{\partial\theta} = \begin{bmatrix} 0 \\ -(m_2gr_1 + m_3gl_1)\cos\theta_2 - m_3r_2\cos(\theta_2+\theta_3)) \\ -m_3gr_2\cos(\theta_2+\theta_3)) \end{bmatrix}$$

The output M, N, and C matrices are used to compute the torque required to provide a desired acceleration at that point in time given the current position and velocity, as described by equation 7. This was used to find the torque required to follow the desired trajectory.

**Equation 7**. summing the forces on the arm to compute desired torque

$$Tau_{desired} = M * acceleration_{ideal}' + C * velocity_{ideal}' + N$$

<u>Computing simulated acceleration and position</u>

Once the torque required has been computed for the desired trajectory as well as the controller torque, gaussian noise is added, as is described above. The simulated new acceleration is computed by equation 8 using the same M, N, and C matrices computed in the previous step as well as the current position and velocity. This acceleration that results is then used to compute velocity, and then position for the next simulated time point.

**Equation 8**. computing new simulated acceleration based on torque

$$Acceleration_{sim} = (M \setminus (Tau_{control} + Tau_{desired} + Tau_{error}) - C * velocity_{sim}' - N)$$

## Evaluation

### Software Feedback Control

To evaluate the efficacy of our feedback control system, we selected controller gain values that may not be representative of the actual Lynx system, but allow great illustration of how our controller works. Given our initial selection for error standard deviation magnitudes, we see that the desired and actual trajectories diverge without feedback control. Through tuning the PID controller, we made interesting observations about how different error magnitudes and controller gains affected our results.

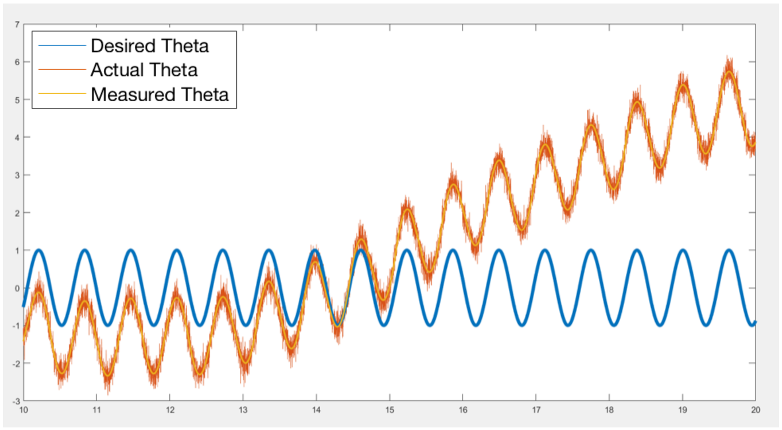
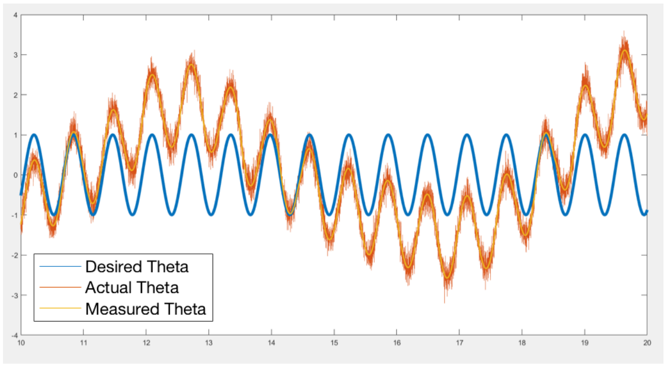


**Figure 9**. simulated result of system over 10 seconds without control, std of sensor reading noise = 0.2 radians, std of torque noise = 2000* $10^{-6}$ Nm

a) 

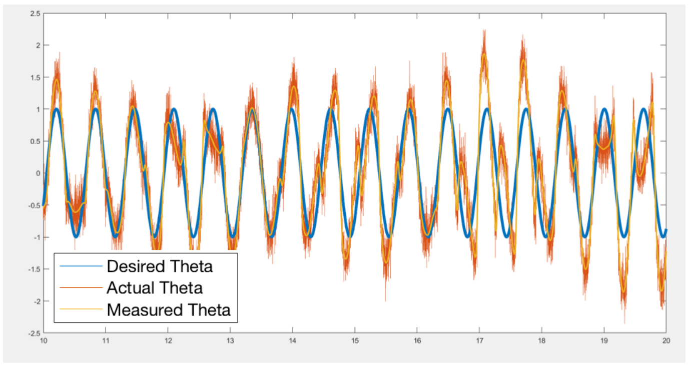
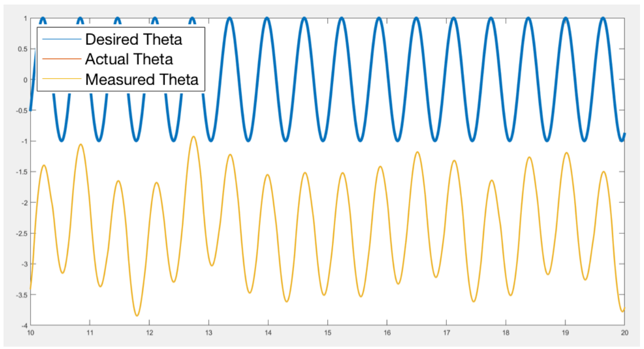
b) 

**Figure 10**. simulated result of system over 10 seconds with proportional gain, magnitude of sensor reading noise = 0.2 radians, magnitude of torque noise = 2000* $10^{-6}$ Nm, a) proportional gain is too low ($k_p$ = -100), b) proportional gain is too high ($k_p$ = -50000)

As we explored values for $k_p$, we looked for a value that would keep the desired and actual trajectories close to each other, but would not cause oscillations in the actual theta curve. We found that if the proportional gain was set too low, as in figure 10a, it was too slow acting and would oscillate around the desired trajectory. On the other hand, if proportional gain is too high, the actual curve will converge with the desired curve more quickly, but it will amplify small errors in theta, causing oscillations of the actual theta curve around the desired theta curve.

a) 

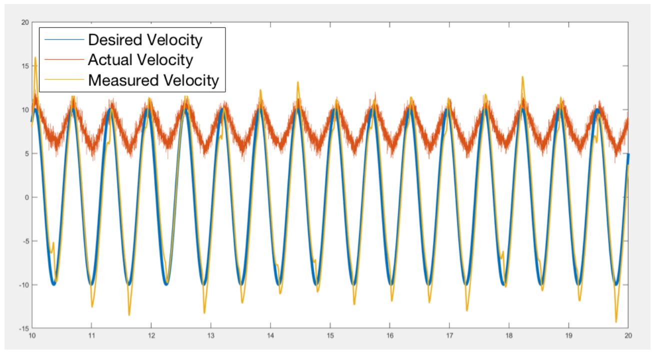
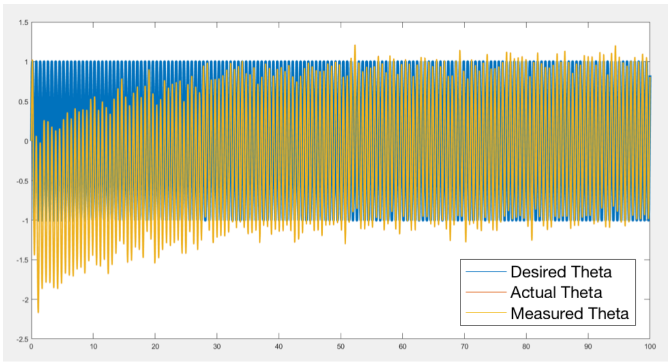
b) 

**Figure 11**. demonstration of how choice of $k_d$ and extreme velocity filter weight can affect results ($k_d$ = -300, w = 0.2), magnitude of sensor reading noise = 0.2 radians, magnitude of torque noise = 2000* $10^{-6}$ Nm, a) Theta trajectories, b) velocity curves

a) 

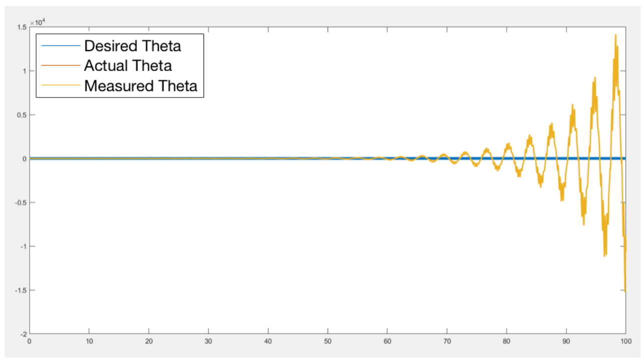
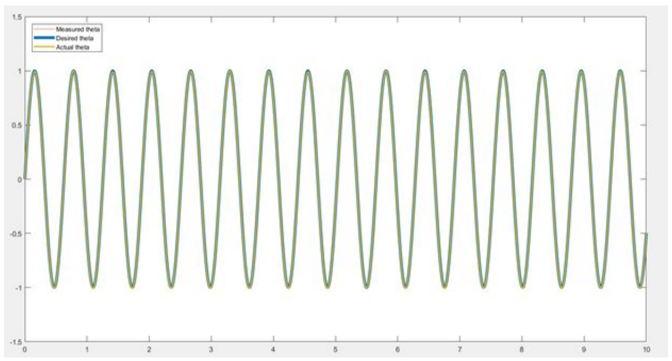
b) 

**Figure 12**. comparison of different $k_i$ values, magnitude of sensor reading noise = 0.2 radians, magnitude of torque noise = 2000* $10^{-6}$ Nm, a) low integral gain will slowly correct the shift ($k_i$ = -50), b) high integral gain will eventually cause oscillations ($k_i$ = -700)

With derivative gain, and a low weight on the velocity filter we found high dependence on the initial condition of our velocity (figure 11). We saw that with this low velocity filter weight, the range of measured theta is shifted significantly lower than the desired theta. This occurred because adding a filter caused the velocity curve to shift up or down on average depending on its initial condition. On the other hand, we observed that the velocity filter did a great job of smoothing the noise introduced by actual velocity. Therefore, we needed to find a balance for the velocity filter where the velocity smoothing effect would be sufficient, but the initial condition would not have too much bearing on future positions. The velocity filter weight in our final simulation was 0.6, more heavily weighing the most recent result. But even after this adjustment, there was still a noticeable downward offset of the actual thetas. At this time, we turned to increasing the $k_i$ value to correct for the shift.

When experimenting with integral gains, we observed that the integral gain acted slowly and eventually corrected for the shift in position over time. At the low extreme of integral gain the position curve could take a long time to converge (figure 12a) to a more accurate position while at the high extreme integral gain caused instability in the system (figure 12b). At this point, we selected a midway integral gain term that relatively quickly (within a few cycles of our sine wave), corrected the shift.

a) 

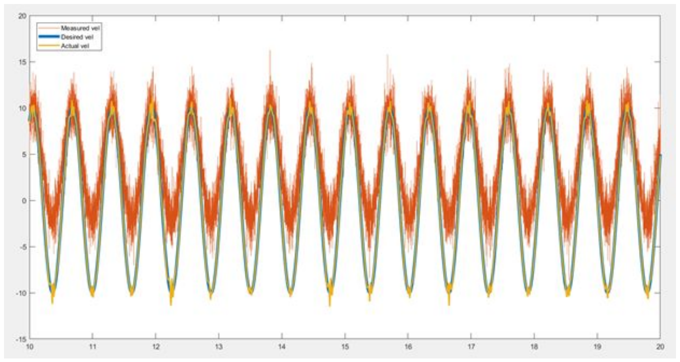
b) 

**Figure 13**. our final controller with, magnitude of sensor reading noise = 0.002 radians, magnitude of torque noise = 2000* $10^{-6}$ Nm, $k_p$ = -800000, $k_d$ = -1000, and $k_i$ = -10000

Putting together the benefits of each of the parts of the controller, we designed a final controller (figure 13) that accurately tracked the desired position despite sensor and system error that was simulated by high gaussian noise. This controller used the velocity filter to lower the effect of the sensor noise on our velocity reading, but was still able to converge to the desired trajectory relatively fast and remain stable over a 100-second period.

## Analysis

As we started this project, we found that the method for computing torques was different than what we had implemented for the haptics lab. Rather than solving for the joint torques based on a torque applied to the end effector, all joint angles had a specific dynamic trajectory to follow. We attempted to solve this problem using linear and angular jacobians, but to no avail. We imagined that each joint would have an individual joint effort stemming from the movement of all the other joints, and once they were computed, we would simply sum the values. But this method would not allow us to easily specify the desired velocity and did not account for the inertia of the links. Therefore we instead used dynamic methods to calculate the relationship between torque and acceleration given a position and velocity. Our consideration of inertial terms as well as external forces allowed us to determine the precise torques we needed to control our system which would then interact with the PID controller.

Once our PID controller was completed, we observed strange results while tuning the values. We observed that with the addition of a non-zero $k_d$ value, the entire actual theta trajectory was shifted downward, as shown in figure E. Naturally, we assumed $k_d$ was the culprit, but after creating plots of the velocity curve along with the theta trajectory, we noticed that the initial velocity might have been greatly influencing the differential gain. By changing the values of the weight in our velocity filter we realized that it was the real culprit. Fortunate circumstances led to this discovery, as if we had chosen a cosine wave to be our desired theta, the initial velocity would have been zero, completely eliminating the change in theta, but not allowing us to create a fully robust system. At this time, we adjusted the weight for the velocity filter to improve the performance and added integral control to correct for the drift in the steady-state of the system.

Next steps for the software controller would be obtaining accurate error magnitude, mass, and length values from the actual robot; we modelled the system as perfect cylinders, whereas the actual robot has mass and volume due to the motors. This may significantly change the moment of inertia of each link. Another simplification in our system was that error in torque was Gaussian, whereas this may not be the case in a physical system. There may be systematic errors resulting in differences between individual robots and factors such as hardware wear. Regardless, the integral gain of our PID controller would be helpful in combating this type of error. In addition, we would need to quantify the torque limits of the robot joints, and incorporate these maximums to limit our controller. Lastly, we could test our controller on new trajectories that more accurately represent potential applications of our system, and use them to evaluate the performance of our control system before integrating it with our kinodynamic planner. In preparation for this, we have included code for hardware_flag = true and the LynxGetAngles function to acquire non-simulated observed joint angles from the robot arm.

## Kinodynamic Planning

### Methods

#### Robot and Simulation

We developed a kinodynamic planning algorithm for our robot which plans in state space, considering the position as well as the velocity of our robot at any given time point. We created our simulation for the three link robot because we required the function of the haptic Lynx setup with joint angle feedback from each joint. Due to this, we adjusted our IK and FK methods from previous labs to compute the correct values for the modified configuration. In addition, we edited the robot simulation plotting code to only display the mobile joints and connecting links in our model.

```
Kinodynamic Planner Algorithm:
  tStart = tree containing starting configuration
  tGoal = tree containing ending configuration

  if (start or goal configs collide with an obstale)
    return false

  while (path from start to goal has not been reached):
    potentialNew = randomly sampled point in state space

    for each tstart and tend:
      find closest point to potentialNew in tree
      while (new point is not reach + no collisions):
        for i = 1, 100
          generate random torques
          evaluate the rho value
        select the torque with the lowest rho value
        move in the direction of the torque for dt

        if collision:
          break and find a new rho value

        else if distance from point < epsilon:
          add point to tree

    if (potentialNew was added to both trees):
      return path from start configuration to goal
        configuration
```

**Figure 14.** pseudo-code for the kinodynamic planner

## Planner

Our implemented kinodynamic planner is similar in its basic strategy to the RRT from Lab 3 as they both use randomly sampled points to determine the path from starting point to goal. Unchanged from the RRT planner, the algorithm used two trees which attempt to find a connecting path to each randomly sampled point. After repeatedly augmenting the paths with new random points, the start and goal paths will converge when a single point can be connected to a point on both trees. Compared to the RRT planner, the benefit of the kinodynamic planner is that it considers the joint velocities which reduces velocity discontinuations, resulting in a smooth trajectory.

The new planner is accounts for the complexity of state space by randomly sampling torque inputs into the system. First, a target point in state space is chosen. Then, 100 torques are randomly sampled and each is evaluated with a “rho” metric which computes whether or not the resulting position and acceleration is closer in state space to the target point. Rather than simply checking for a collisions in the straight and direct path from the tree point to the new sampled point in configuration space, as was done in the RRT we used for lab 3, we need to evaluate the result of each small step in state space as a result of these torque values. The chosen randomly-sampled torque command at each step must move from the closest point on the tree towards the new sampled point until a collision or successful connection occurs. The algorithm runs continuously until the two trees starting at the start and end states are able to both reach the same target point. Then, a path between the two trees is calculated from this intermediate node. Because of the random nature of the planner, the time it takes to generate a path varies greatly, though it can be controlled to an extent by metrics such as ρ and ε described below.

With the addition of velocity, we move from planning in configuration space to planning in state space, but note that collisions are still checked in configuration space, since velocity cannot cause collisions. The robot arm can instantaneously stop as the inertia can be overcome by the motors in contrast to other systems, such as vehicles with high inertia and limited braking power.

As a result of randomly sampling control inputs rather than points in configuration space, the computation time is greater than our previous RRT planner. Because the computation of the path is done prior to execution; for some applications it may be acceptable that the time to discover a path is longer. In addition to added computation time, the kinodynamic planner also requires more data to be stored. With each time step and randomly sampled input, a new node is created that has an additional property that indicates the amount of torque to apply to the joints for a given constant timestep, $\Delta t$. The random sampling of inputs thus requires much more memory usage in contrast to our old RRT planner which generally needed to sample just a few configurations to find a valid path.

To determine how close two points were in state space, we used equation 9, referred to as ρ. This metric is crucial for our algorithm as it is used to select the optimal torque to apply to try to connect points in the tree to a new potential point.

**Equation 9**. metric used to determine distance between two points in state space. Our final algorithm used $\text{weight}_\rho$ = 0.71

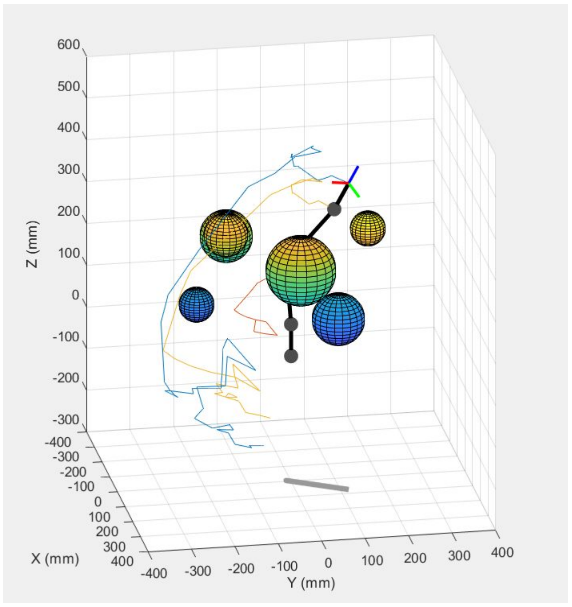
$$\rho = weight_\rho * (norm(difference\ in\ position)) + (1 - weight_\rho) * (norm(difference\ in\ velocity))$$

In addition to $\text{weight}_\rho$, another important tuned parameter is ε -- it determines how low the ρ value must be to consider two points as connected. This parameter was tuned by trial and error. It was observed that when ε was too large, many velocity discontinuities would exist, not very different from our original RRT planner. While when ε was too small, the planner takes an extremely long time to discover a path, a small ε requires more iterations, $\Delta t$ each to reach an acceptable distance away from the new point in state space. Therefore, when coming up with the final version of our path planner, we experimented with different ε values to find a balance of speed and small velocity discontinuities.

## Evaluation

Our kinodynamic planner was successful in creating a smoother path between two specified beginning and end states. An example of this can be seen in figure 15b. Because the planned trajectory considers physically realizable torque values, the resulting path is smoother and considers acceleration and deceleration of the arm as it moves between configurations. In contrast, our previous RRT planner required the arm to make sharp turns at each randomly-sampled configuration, resulting in jerky motion which limits its real uses. There is a limitation when joining the start configuration and goal configuration trees, when the two points are iteratively moving towards the final point, there is a small jump that is reminiscent of the discrete velocity motions exhibited by the RRT planner. This jump can be reduced by reducing the size of epsilon; this is a tradeoff between smoothness at this point and running time required to compute the path. Perhaps determining ε as a function of current velocity would allow for an improvement here.

A very limiting factor of our new kinodynamic planner is the time it takes to find a valid trajectory. Average computation times for valid trajectories of the planner compared to our planner from lab 3 are show in in table 4. For our old planner it took much longer to generate a collision matrix in configuration space due to a higher number of joints being used, but the new planner has to both do that for three joints and perform many more random samples when creating the trajectory. Overall, whether or not the tradeoff is worthwhile depends on specific applications of the robotic arm.

a) 

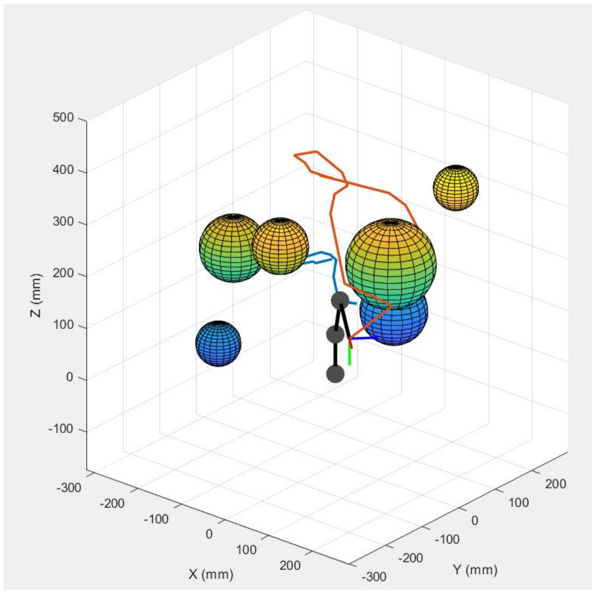
b) 

**Figure 15**. comparing RRT with kinodynamic planner when traversing a path: a) path from RRT, b) path from kinodynamic planner

**Table 4**. comparison of time each planner takes to find a path, found using MATLAB”s tic/toc command

| | **Time elapsed (seconds)** |
|---|---:|
| **RRT Planner from Lab 3** | 0.78 |
| **Kinodynamic Planner** | 312.46 |

## Analysis

Given the smoother trajectory, this kinodynamic path planner would be safer to implement on the physical robot. When we designed our RRT planner, the jerky portions of the path was a concern for commanding the physical robot. Being able to move the robot arm more smoothly can be crucial for precise tasks, such as pipetting or assembling small parts. This ability is also important in the design of robotic systems with more natural, “human-like” motion. The design of the algorithm can be extended other robotic arms, with more or different degrees of freedom.

Future steps to improve on our planner include, selecting an optimal ρ metric. Currently we use a simple function that simply weights the norm of the difference vector of q values greater than the norm of the difference vector of the velocities. Given the randomness of each trial, it was difficult for us to tune the weight to the optimal value. Furthermore, the euclidean norm might not be the best intermediate metric to use. We are definitely interested in exploring different ways of formulating ρ to produce more favorable results.

Finally, after the feedback controller’s parameters are tuned for the physical robot, we would pass the trajectory planned by the kinodynamic planner through the controller, and observe the effect with and without the controller. We expect the accuracy of the trajectory when used on the physical robot to be improved by the feedback controller. The controller would correct for errors in the position and velocity of the robotic arm as it completes the trajectory. This would bring the entire project together to provide smooth trajectory with minimal errors to the Lynx.

## References

LaValle, S. M., & Kuffner Jr, J. J. (2001). Randomized kinodynamic planning. The International Journal of Robotics Research, 20(5), 378-400.
Murray, R. M., Li, Z., Sastry, S. S., & Sastry, S. S. (1994). Chapter 4. A mathematical introduction to robotic manipulation. CRC press.