ME699 Final Project

ME699 - Robotic Modeling and Control
Department of Mechanical Engineering
University of Kentucky
Lexington, KY 40506, USA

Benton Clark (brcl223@g.uky.edu)
Ethan Howell (ethan.howell@uky.edu)
Brian Moberly (brianmoberly1@uky.edu)

April 30, 2020

Table of Contents

List of Figures

1 Introduction

Objective

The objective of our project was to take a 3 degree of freedom (DOF) robot manipulator and move an object of unknown mass from an initial state to a specified goal state, and then return the manipulator to its original position. Three primary methods were experimented with in this project. The first involved developing a method to estimate the mass matrix of the manipulator, with which a controller was designed to track the arm to the initial state, and back home when the object had been dropped in the goal state. The second part involved using a Kalman filter to estimate the positions of the initial state, goal state, and the current joint configurations of the manipulator. The last part was developing an adaptive controller to account for the unknown mass of the object to move it successfully from the initial state to the goal state.

Ancillary Tasks

While not the focus of our project, several other components were required in order to complete the described task. These include the following:

- Path planning algorithm (RRT*)
- Trajectory planner
- Collision detection
- Inverse kinematics to map Cartesian space target into joint space

These items are mentioned here for completeness sake, but are not discussed in details in the report as they were not the primary objective.

Mass Matrix Modeling

Modeling the mass matrix of a robotic manipulator becomes increasingly difficult as the DOF of the system increase. Our attempt to avoid this complexity was to model the mass matrix with a neural network. To further decrease the dimensionality of the problem, it is well understood that the mass matrix is a positive definite (PD) square matrix. Thus the neural network output only the lower triangle of the matrix, and the remaining values were copied from the output accordingly. Once the mass matrix neural network was trained, it was used in simulation to develop a trajectory tracking controller.

Uncertainty in Configurations and Perception

To model the uncertainty in the system, a “camera” was assumed to be in the world, providing the position of the initial state and the goal state. The readings given by the camera sensor were assumed to be noisy, having zero-mean Gaussian white noise about the positions desired. In order to correctly estimate the desired positions for the arm to reach, a Kalman filter was implemented. Since this task simply required estimating a stationary target, the filter estimated the position while converging towards the ideal target.

Once the desired positions were established, the arm plotted a trajectory. However, the joint readings from the arm while moving were also noisy. These too had a zero-mean Gaussian white noise affecting the readings, with these readings being fed directly to the full-state feedback controller. A Kalman filter was used once again to reduce the noise from the joint readings. This task was significantly more challenging though since the joint positions were not stationary, so a dynamic model of the arm was produced to accurately update the Kalman filter throughout the process.

Accounting for unknown parameters

With the goal of picking up an object of unknown mass, this introduces an unknown parameter to the overall system. The effect of this unknown parameter propagates throughout the dynamics of the system throughout its interaction. In order to account for this term, we look to the field of adaptive control. Classical control systems assume that the system dynamics are either known, or able to be approximated. Because of this assumption, there is a shortfall with classical methods when unknown parameters are introduced to the system. Similarly to classical control, adaptive control requires that we pick a control law, often chosen to cancel out the undesirable terms associated with the dynamics, which gives us the ability to select gains to change how the system behaves. In addition to this, we must also select an adaptation law, which allows our control to morph to the evolving conditions at hand. Using this methodology, we are then able to design our control in such a way that we can stabilize the system and achieve desired states.

To begin this project, the first step was to scour the internet and look for implementations of adaptive control models in 2-link manipulators. We quickly learned that such models either don't exist, or are not publicly available. An abundance of classical PD control examples were available, and these models served as a benchmark to measure the performance of our Adaptive model.

2 Theory and Methods

Mass Matrix Modeling

The general equation of motion for a robotic manipulator can be expressed as:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(g) = \tau \tag{2.1}$$

where $M(q) \in \mathbb{R}^{n\times n}$ is the mass matrix of the joints, $C(q,\dot{q}) \in \mathbb{R}^{n\times n}$ is the Coriolis matrix, $G(q) \in \mathbb{R}^n$ is the conservative force vector acted on the arm by gravity, and $\tau \in \mathbb{R}^n$ are the torques commanded to each joint [**spong**]. Each of the terms on the left hand side of the equation can be directly modeled when the manipulator joint information is known. However, when this information is not available, a neural network can be used to estimate the value of these matrices.

To model the mass matrix with a function approximator $\hat{M}(q)$, we must model a loss function to train the network. This proves more difficult, as the mass matrix must be isolated on the left hand side of the equation. However, if the number of acceleration samples taken at a given configuration q satisfy $rk(M(q)) = n$, we could then construct an acceleration matrix $\ddot{Q} = [\ddot{q_1}, \ddot{q_2} \dots \ddot{q_n}]$ which satisfies $rk(\ddot{Q}) = rk(M(q)) = n$. Properties of linear algebra now guarantee that a unique inverse of this matrix must exist, and the equation can then be expressed as:

$$M(q)\ddot{Q} + C(q,\dot{Q})\dot{Q} + G(Q) = \mathrm{T} \tag{2.2}$$

$$M(q) = \ddot{Q}^{-1}(\mathrm{T} - G(Q) - C(q,\dot{Q})\dot{Q}) \tag{2.3}$$

where $G(Q) \in \mathbb{R}^{n\times n}$ is a one dimensional matrix where each column satisfies $G_i(Q) = G(q)$.

The problem with this method is that the Coriolis term is still in place, and we have no estimate for this value. However, if we take our $\ddot{q}$ samples at the initial point when we first begin accelerating, we should see that $\dot{q} \approx 0$, and thus Eq. (2.3) can be expressed as:

$$M(q) = \ddot{Q}^{-1}(\mathrm{T} - G(Q)) \tag{2.4}$$

which now allows for a loss function to be formulated. This method assumes that the gravity vector is known, or that a close approximate may be made to cancel this term on the right hand side of the equation.

The Kalman Filter

The Kalman Filter is used to mitigate the noise present in a system as the system moves. The goal is to know what the robot is doing at the next time step, using the knowledge of the current time step. The motion of the system is described by the following uncertainty model:

$$x_{t+1} = \mathbf{A}x_t + \mathbf{B}u_t + v_t, \tag{2.5}$$

where $\mathbf{A} \in \mathbb{R}^{n\times n}$ is the transition matrix of the dynamic system, $mathbfB \in \mathbb{R}^{n\times m}$ is the control matrix for the inputs, $x_t \in \mathbb{R}^n$ is the current estimate of the state, $u_t \in \mathbb{R}^m$ is the current input to the system, and v_t is a Gaussian noise variable pertaining to the uncertainty in the system behavior.

The measurement taken in by the filter follows the uncertainty model

$$y_t = \mathbf{C}x_t + w_t, \tag{2.6}$$

where $\mathbf{C} \in \mathbb{R}^{m\times n}$ is the observability matrix of the system and w_t is a Gaussian noise variable describing the uncertainty in the process.

The Kalman Filter proceeds with a motion update that represents the motion of the model with noise:

$$\mu_t^{pred} = \mathbf{A}\mu_t + \mathbf{B}u_t \tag{2.7}$$

$$\Sigma_t^{pred} = \mathbf{A}\Sigma_t\mathbf{A}^\intercal + \mathbf{R}_w \tag{2.8}$$

After completing the motion, the filter makes a measurement update:

$$\mathbf{K}_t = \Sigma_t^{pred}\mathbf{C}^\intercal(\mathbf{C}\Sigma_t^{pred}\mathbf{C}^\intercal + \mathbf{R}_w)^{-1} \tag{2.9}$$

$$\mu_{t+1} = \mu_t^{pred} + \mathbf{K}_t(y_t - \mathbf{C}\mu_t^{pred}) \tag{2.10}$$

$$\Sigma_{t+1} = (I - \mathbf{K}_t\mathbf{C})\Sigma_t^{pred}(I - \mathbf{K}_t\mathbf{C})^\intercal + \mathbf{K}_t\mathbf{R}_w\mathbf{K}_t^\intercal \tag{2.11}$$

where $\mathbf{R}_w \in \mathbb{R}^{m\times m}$ is the measurement uncertainty matrix. While an alternative form for the Σ_{t+1} matrix exists, it is considered to be numerically unstable, so it was avoided in this implementation [**bucy-joseph-filtering**].

Implementation

For the stationary targets, there is no input to affect their position, so the matrices are derived as follows:

$$\mathbf{A}_s \triangleq \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \ \mathbf{B}_s \triangleq 0, \ \mathbf{C}_s \triangleq \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note that this corresponds to a stationary, fully observable system, where the measurements are taken corresponding to the estimated x, y and z coordinates in Cartesian space.

For estimating the joint configurations, a more complicated system was required. The robot had three joints, each with a corresponding position and velocity. Thus, the following system matrices were derived:

$$\mathbf{A}_j \triangleq \begin{bmatrix} 1 & \Delta t & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \ \mathbf{B}_j \triangleq \begin{bmatrix} \frac{1}{2}\Delta t^2 & 0 & 0 \\ \Delta t & 0 & 0 \\ 0 & \frac{1}{2}\Delta t^2 & 0 \\ 0 & \Delta t & 0 \\ 0 & 0 & \frac{1}{2}\Delta t^2 \\ 0 & 0 & \Delta t \end{bmatrix}, \ \mathbf{C}_j \triangleq \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

where Δt was the discrete time step used between each sample. Again, the entire system was fully observable, and the inputs were the joint accelerations, calculated from each previous torque input commanded to the system.

Adaptive Control

We can simplify Eq. [2.1] to the following

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q)q = Y(q,\dot{q},\ddot{q})\theta = u \tag{2.12}$$

where $Y(q,\dot{q},\ddot{q})$ is known as the "regressor" function containing only state variable terms, and θ is a vector function of the unknown parameters. from Eq. [2.12], we can select the control law, u, as the following

$$u \triangleq \hat{M}(q)(\ddot{e} + K_d\dot{e} + K_p e) + Y(q,\dot{q},\ddot{e})\hat{\theta} \tag{2.13}$$

where $\hat{M}$ is the estimation of the mass matrix, $\ddot{e} = \ddot{q}_d - K_p\tilde{q} - K_d\dot{\tilde{q}}$ is error in the joint acceleration, and $\hat{\theta}$ is the estimated vector function of unknown parameters. Substituting into Eq. [2.12], we can evaluate the error dynamics as

$$\ddot{e} + K_d\dot{e} + K_p e = \hat{M}^{-1}Y(q,\dot{q},\ddot{q})\tilde{\theta} = \Phi\tilde{\theta} \tag{2.14}$$

where $\tilde{\theta}$ is the error in the parameter estimation. Since we do not know the vector function, it is impossible to know the error associated with it. We do know however that as long as both K_d and K_p are positive definite, our error decays to zero and satisfies the following ODE.

$$\dot{e} = Ae + B\Phi\tilde{\theta} \tag{2.15}$$

In order to determine θ, we resort to Lyapunov analysis with the candidate function

$$V = e^T Pe + \tilde{\theta}^T\Gamma\tilde{\theta} \tag{2.16}$$

Lyapunovs direct method yields the adaptation law which allows us to directly determine $\hat{\theta}$ for our control law given in Eq. [2.13], which we find to be

$$\dot{\hat{\theta}} = -\Gamma^{-1}\Phi^T B^T Pe \tag{2.17}$$

Where Γ is an arbitrary positive definite, symmetric matrix.

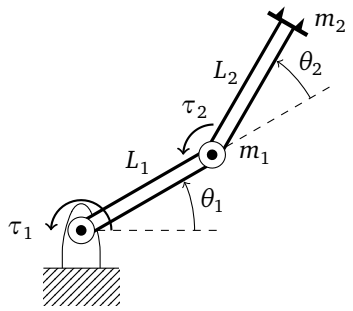

Figure 2.1: 2-Link Robotic Manipulator

To apply this method to our particular case, consider the 2-Link manipulator shown in Fig. 2.1. Following Example 6.2 − 1 in [1], the regressor function, $Y(q,\dot{q},\ddot{q})$, is given as

$$Y(q,\dot{q},\ddot{q}) = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} \tag{2.18}$$

$$Y_{11} = l_1^2\ddot{q}_1 + l_1 g cos(q_1) \tag{2.19}$$

$$Y_{12} = l_2^2(\ddot{q}_1 + \ddot{q}_2) + l_1 l_2 cos(q_2)(2\ddot{q}_1 + \ddot{q}_2) + l_1^2\ddot{q}_1 - l_1 l_2 sin(q_2)\dot{q}_2^2 \tag{2.20}$$
$$- 2l_2 l_2 sin(q_2)\dot{q}_1\dot{q}_2 + l_2 g cos(q_1 + q_2) + l_1 g cos(q_1)$$

$$Y_{21} = 0 \tag{2.21}$$

$$Y_{22} = l_1 l_2 cos(q_2)\ddot{q}_1 + l_1 l_2 sin(q_2)\dot{q}_1^2 + l_2 g cos(q_1 + q_2) + l_2^2(\ddot{q}_1 + \ddot{q}_2) \tag{2.22}$$

To complete the parameters needed for both the adaptation law, and control law, we choose the matrices **A**, **B** and **P** as

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ -K_p & -K_d \end{bmatrix}, \qquad \mathbf{B} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}, \qquad \mathbf{P} = \frac{1}{2}\begin{bmatrix} (K_p + \frac{1}{2}K_d) & \frac{1}{2} \\ \frac{1}{2} & 1 \end{bmatrix} \tag{2.23}$$

For simplicity, the third joint which rotates the base of the 2-Link manipulator, is controlled by a traditional PD Controller. Thus the control law for the 2-Link manipulator shown in Fig. 2.1 is given as

$$\tau_1 = Y_{11}\hat{\theta}_1 + Y_{12}\hat{\theta}_2 \tag{2.24}$$

$$\tau_2 = Y_{21}\hat{\theta}_1 + Y_{22}\hat{\theta}_2 \tag{2.25}$$

$$\tau_3 = -K_d\dot{e} - K_p e \tag{2.26}$$

3 Experiments

One primary experiment was run to test the effectiveness of the mass neural network, the adaptive controller, and the Kalman filter. This was the task proposed in the introduction, consisting of three phases. Each of these phases provided different data to be collected and measured to see how well each method performed. In addition, the loss metrics of the mass neural network during training will be plotted and discussed.

Controls Experiment

Both the adaptive control and the mass matrix neural network control were tested against each other in the three different phases. These phases consisted of the initial trajectory towards the target object, moving the object of unknown mass to the goal state, and finally returning to the original position the arm started in. Both of these methods were compared to a baseline of a standard PD trajectory tracking controller as well, to serve as a baseline for performance. The metric for these experiments was the error in tracked trajectory for both joint configurations and velocities under control.

Mass Experiment

The adaptive control was also tested multiple times in the second phase (moving the object of unknown mass) where the mass was varied. The performance under differing weights was of particular interest. The metric for this experiment was the error in tracked trajectory for the three joint configurations and velocities during the control.

Kalman Filter Experiment

To measure the effectiveness of the Kalman filter, both the stationary and moving filter results were examined. The error in actual position (known due to simulation) versus predicted position was used as the metric. If effectively implemented, the error in predicted position should begin to converge towards the actual position as more samples are acquired and the confidence of the filter increases.

The experiments will use white noise samples from the Gaussian distribution $n \sim N(0, 0.1)$. This variance provides enough noise to show the filter working effectively, but not so much that it cannot provide accurate estimates within the short time span of the trajectory.

Control Models

To validate our control models, it is important to evaluate the error dynamics associated with the controllers. To do this, we begin with the comparison of the error convergence between the Adaptive Control model and the classical PD Control model. This comparison was made with a fixed, known weight, as well as no noise in the joint values.

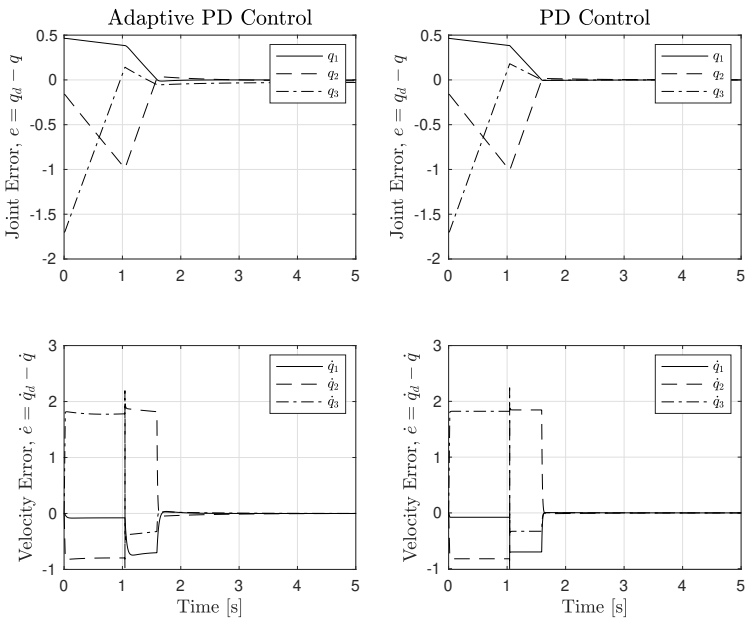

Figure 3.1: Convergence of controller error dynamics

As shown in Fig. 3.1, both controllers converge to the steady state value of zero in approximately the same time. One benefit to adaptive control however, is that as the mass changes (i.e. an object is picked up), the adaptation law accounts for the change in mass. With PD Control, there is no way to account for this mass addition and an increase in error is introduced.

We then evaluate each controller by it's ability to follow the desired trajectory provided by the *RRT** path planning algorithm.

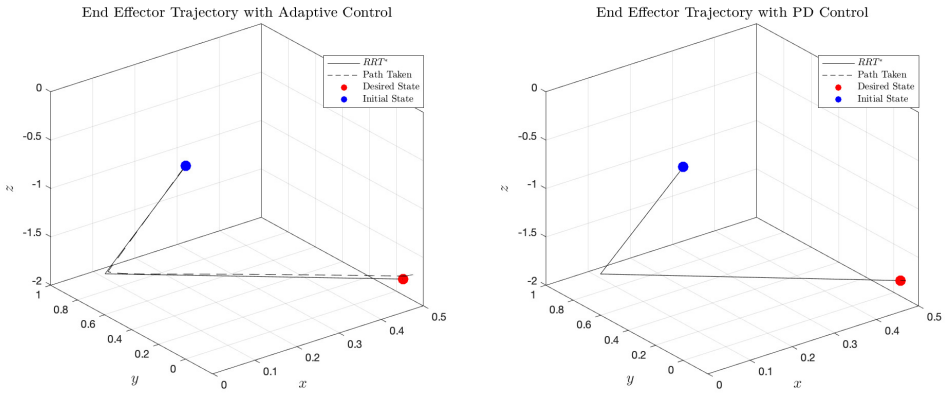

Figure 3.2: Desired end-effector trajectory vs actual

We can see from Fig. 3.2 that in this case, the PD Controller maintains the desired trajectory more tightly than the Adaptive Control model. However, the error between the two control models is of an insignificant degree, thus the adaptive control model is chosen for the simulation.

4 Results and Discussion

Mass Matrix Training

Figure 4.1 shows the training, validation and test loss during the training of the neural network. As shown, the loss seems to converge to a very small error in all three cases, meaning that the neural network should effectively represent the mass of the manipulator if enough of the space was sampled uniformly. One thing to note here was that Julia's RigidBodyDynamics package did not seem to honor the damping fields in the URDF files — therefore, these training results show only the results when no friction was present in the system.

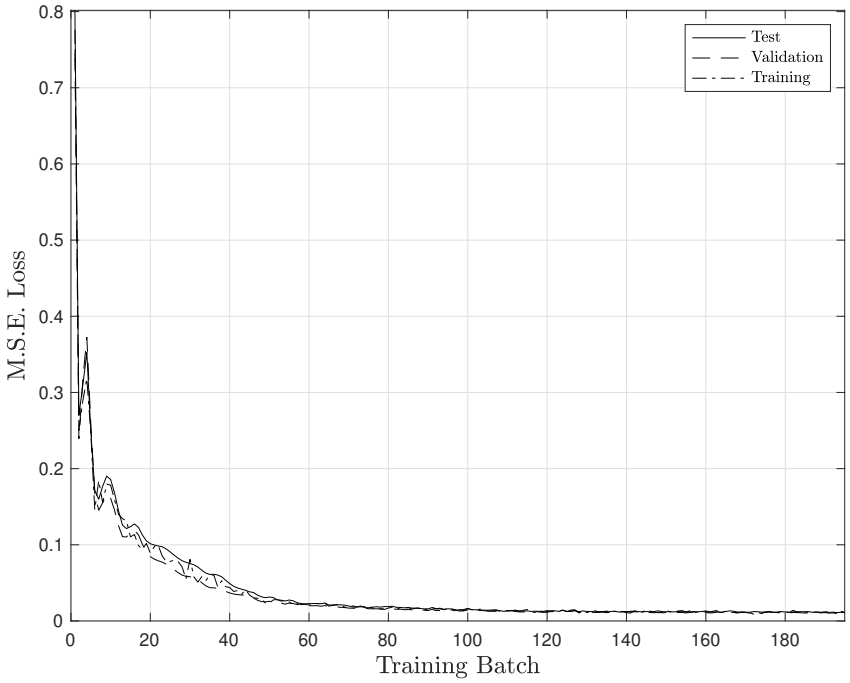

Figure 4.1: Mass Neural Network Cross Validation Loss Metrics

Kalman Filter

Figure 4.2 shows the results of the Kalman filter when estimating the true position of the arm while moving through a trajectory. All in all the results show about what is expected — as more data samples are provided to the filter, the estimations improve. One interesting note is the performance difference with and without the additional weight. The margins of error are significantly smaller for the initial and home-bound trajectories than for the goal-bound trajectory when additional weight was present. This is probably due to the failure in the dynamics when the additional weight was present. The dynamics model used by RigidBodyDynamics was unable to accurately estimate the current acceleration when the additional weight was present, and this led to larger errors in the Kalman estimation. Interestingly enough though, the values did still begin converging by the end of the simulation.

Another area to note is the sharp spikes in the estimation error. These are most likely due to changing trajectory way-points. The trajectories were built using linear interpolation between each point, so sharp spikes in the dynamics are probably present when traveling between the way-points provided during by the RRT* algorithm. Even with these spikes, the error quickly settles, and still begins to converge.

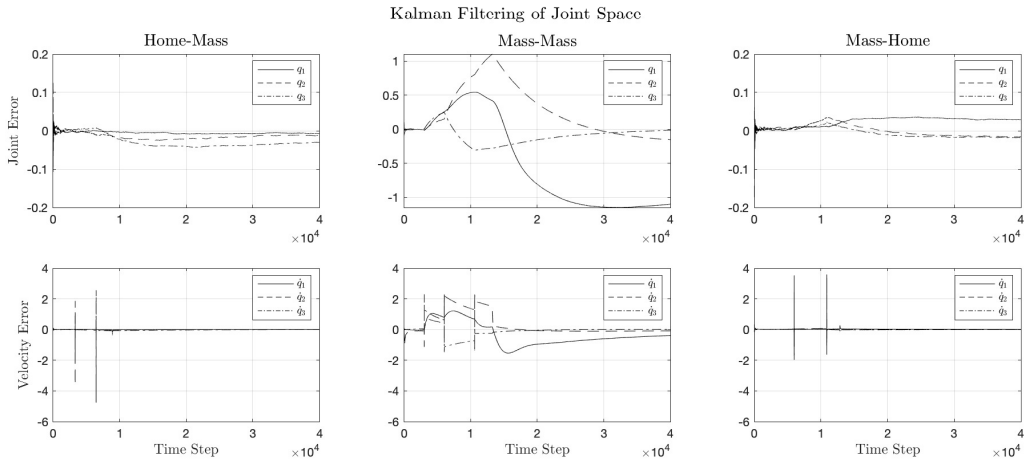

Figure 4.2: Kalman filter prediction error during initial trajectory — joint configurations and velocities.

Control Model

In order to implement the adaptive control in the Julia programming language, we needed a way to account for the adaptation law. This implementation was not initially very straightforward, as the “simulate!()" function places certain restrictions on the inputs to the control function that we choose. To alleviate this issue, we made the adaptive controller a mutable structure which contains the adaptation parameter as a member. This implementation allows us to continually update the parameter, without having to troubleshoot issues associated with other methods.

With this control fully implemented, we then move to the full simulation of the control in which we track the state error and end-effector trajectory.

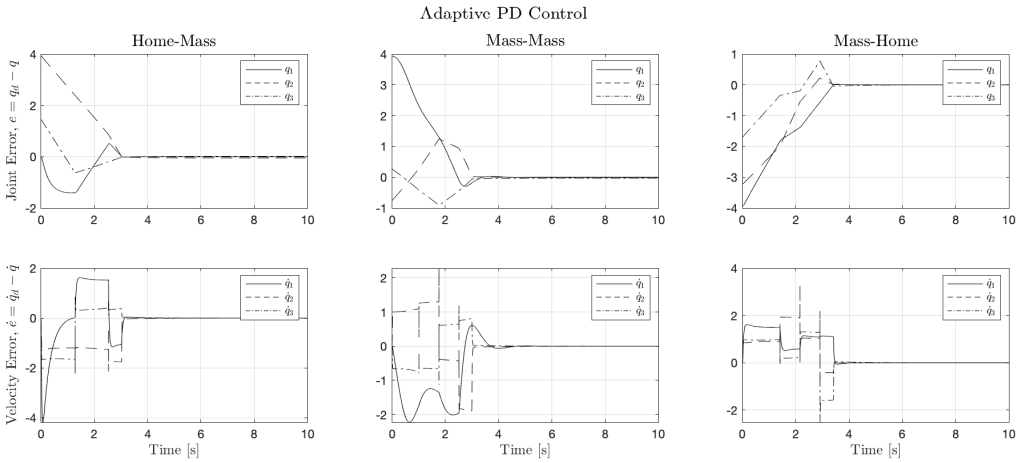

Figure 4.3: Adaptive controller error in full simulation

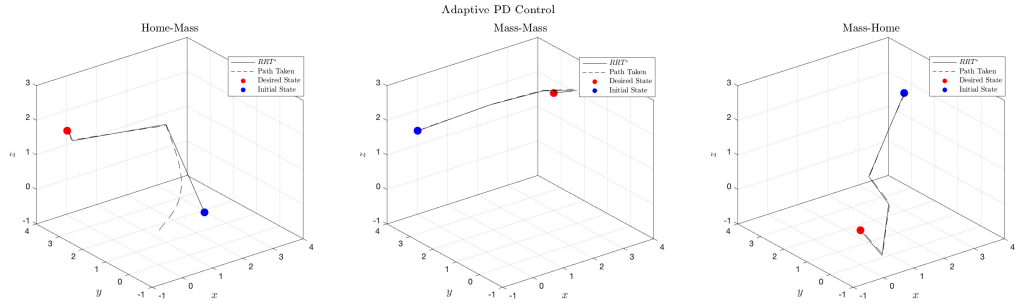

Figure 4.4: End-effector trajectory with adaptive control

We can see from Fig. 4.4 that our initial error results in a false start position, however this error quickly decreases to zero as expected, and the trajectory is followed very tightly for the remainder of the simulation. We then look at a strict PD model, to observe the effects that an increase in mass has on the control.

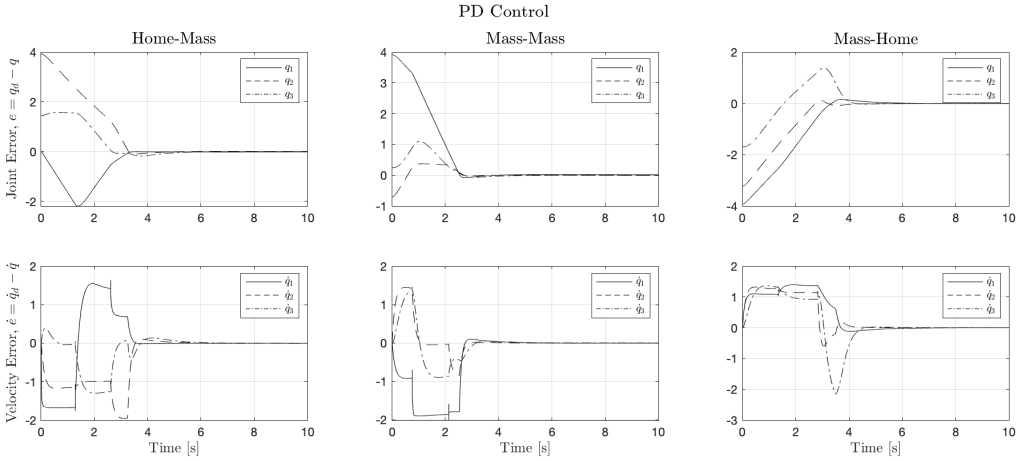

Figure 4.5: PD controller error in full simulation

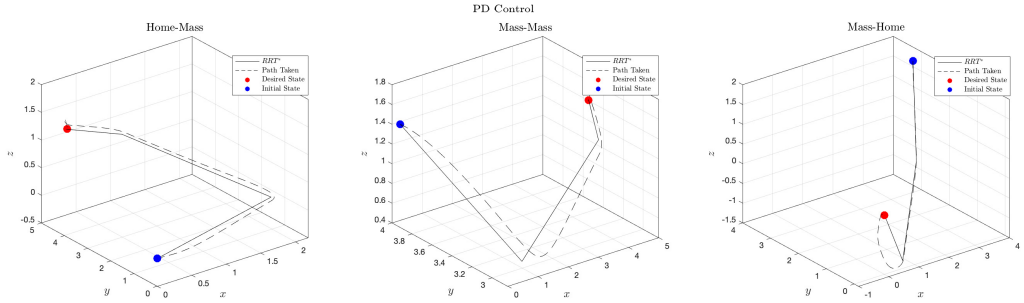

Figure 4.6: End-effector trajectory with PD control

In contrast to the adaptive model, the PD controller places the end-effector in the desired start position. However, we can easily see from Fig. 4.6 that the PD control is not able to follow the desired trajectory as closely, and once the mass is picked up this issue worsens.

5 Conclusion

Mass Matrix Modeling

While we could not simulate the robot in a frictional environment, the results from our learned mass matrix were quite promising. It should be noted that while there was no friction involved, the system was still simulated to acquire the acceleration values to test against. Even though small errors were present in the findings, the controller using the estimated mass matrix still performed quite competitively compared to those with accurate knowledge of the mass matrix. It will be interesting to see if this approach is possible on a real robot system with various levels of friction.

Kalman Filter

The Kalman filter in the experiments worked quite well. Even with fairly high variance in the sampled noise, the filter seemed to converge decently, with better results when the additional mass was not present. Given longer trajectories, it would be expected that these values could have dropped even lower, which is a good sign that the Kalman filter is effectively reducing the noise from the given samples.

Control Implementation

The biggest challenge associated with the control model was the derivation of the system regressor function. Thanks to models found in [1], this step was alleviated and also allowed us to compare the system response to provided examples. While this type of control model is favorable when parameters are unknown, application of this model in areas such as the automotive industry is not always necessary. It is common practice to use robotic manipulators in spot welding, and material handling applications where the properties such as the mass are known ahead of time. In cases like this, an adaptive model is not necessary and other models may be used. However in applications such as aircraft controls, there are several properties that are both unknown, and subject to external conditions. In these applications, and adaptive model would excel.

Challenges and Obstacles

Many of the challenges and obstacles faced during this project related to the limitations within the Julia packages themselves for simulation. The most difficult problem was the lack of control during the simulation of the dynamics themselves. Since RigidBodyDynamics uses the ODESolver package, the dynamic steps are not controllable, and adaptive stepping is used when solving these ODEs. On top of that, there is no easy and convenient method to pass state to and from the simulate function, meaning all state required, both as input and output, needed to be passed in as a functor before the function was called. This lead to many bugs and debugging headaches.

Another limitation seemed to be in the way RigidBodyDynamics handled URDF file values. In particular, it did not seem to honor any kind of static or dynamic friction models, meaning simulating with friction was not possible.

The visualization support was also very limited. While MeshCat and MeshCatMechanisms boast several nice features for animations, we found these aspects of the packages to be unusable or difficult to figure out. We ended up settling on some of the lower level animation functions, as their high level behavior and in-browser animation controls did not work for us.

One last issue was with the machine learning. Flux, the de-facto package for machine learning in Julia appeared to have several broken aspects to it. One in particular made it to where you could not train a model on a GPU if the input and output dimensions of the neural network did not match. Since we were using a three DOF system, our mass matrix had input size 3 and output size 6, and there was a broadcasting Cuda error thrown when trying to run the model on the GPU. It was still possible to run on the CPU, but trained much slower.

Lessons Learned

The first lesson learned while implementing this project was the difficulty of working with high DOF systems. While it should be obvious that increasing DOF will increase the complexity of nearly any non-trivial system, the

degree by which it scales in robotics is brutal. Thankfully we were able to limit the scope of our project to a three DOF system, which made implementing the code to our project much simpler.

Secondly, we got first hand experience in implementing several different types of controllers, including tracking controllers. It was very rewarding to see our theory correctly applied to have the robot successfully navigate the obstacles in our world and follow the planned trajectories in simulation. Our project combined many different facets of the material covered throughout the semester, so to see them all coordinating together was quite exciting.

Works Cited

[1] Lewis, Frank L, Dawson, Darren M, and Abdallah, Chaouki T. *Robot manipulator control: theory and practice*. CRC Press, 2003.