Lab 2: PD & PID Joint Control

Caleb Albers & Karun Koppula

May 11, 2018

1 Introduction

The focus of this lab is to derive the equations of motion for the CRS Robot and use them in the implementation of Proportional-Derivative (PD) and Proportional-Integral-Derivative (PID) control. By implementing PD and PID control loops on each joint, the end-effector can smoothly follow arbitrary trajectories with minimal overshoot, little to no steady-state error, and no jittery or abrupt movements.

2 Equations of Motion of Two-Link Planar Arm

2.1 Energy Equations

The description of energy within the two-link planar arm can be simplified by assuming that all mass is at single points, with the center of mass equal to the geometric center of each link. The kinetic energy of the system is shown in Equation 2.12, which is derived geometrically from Figure 1 by summing the kinetic energy from Link 1 with the kinetic energy that Link 2 can provide with respect to the angle given by q1 and q2. The equations are simplified by using the mapping shown in Equation 2.1, which was given in the lab manual.

$$p_1 = m_2 l_{c2}^2 + m_2 l_2^2 + I_2 \tag{2.1}$$
$$p_2 = m_3 l_{c3}^2 + I_3 \tag{2.2}$$
$$p_3 = m_3 l_2 l_{c3} \tag{2.3}$$
$$p_4 = m_2 l_{c2} + m_3 l_2 \tag{2.4}$$
$$p_5 = m_3 l_{c3} \tag{2.5}$$

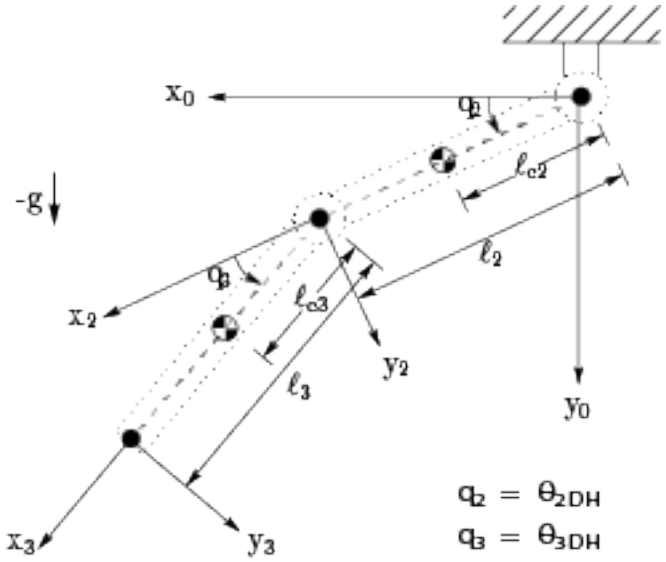

Figure 1: Two-link revolute joint arm
(Figure adapted from Fig 7.8 in Ref. [1])

In order to derive the equations of motion, the Jacobian for the two links must be found. Both joints are rotational, so the Jacobian follows the following convention:

$$J_i = \begin{bmatrix} J_v \\ J_\omega \end{bmatrix} = \begin{bmatrix} z_{i-1} \times (o_n - o_{i-1}) \\ z_{i-1} \end{bmatrix} \tag{2.6}$$

Applying that formula to Joint 2 gives:

$$J_2 = \begin{bmatrix} -l_{c2} \sin\theta_{2DH} \\ l_{c2} \cos\theta_{2DH} \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.7}$$

We can then derive the 3D velocity vector for link 2 (with respect to the center of mass) by taking J_v (the top-most 3xN portion of the link 2 Jacobian) and multiplying it by the rotational velocity vector $\dot{\theta}$.

$$\begin{aligned}\text{Joint 2 Velocity} &= J_{2,velocity}\begin{bmatrix}\dot{\theta}_{2DH}\\ \dot{\theta}_{3DH}\end{bmatrix}\\ &= \begin{bmatrix}\dot{\theta}_{2DH}\big[-l_{c2}\sin(\theta_{2DH})\big]\\ \dot{\theta}_{3DH}\big[l_{c2}\cos(\theta_{2DH})\big]\\ 0\end{bmatrix}\end{aligned} \tag{2.8}$$

Likewise, we can form the Jacobian for link 3. This Jacobian will be a 6x2 matrix, however, as it is dependent on both θ_{2DH} and θ_{3DH}.

$$J_3 = \begin{bmatrix}-l_{c3}\sin(\theta_{2DH}+\theta_{3DH}) - l_2\sin(\theta_{2DH}) & -l_{c3}\sin(\theta_{2DH}+\theta_{3DH})\\ l_{c3}\cos(\theta_{2DH}+\theta_{3DH}) + l_2\cos(\theta_{2DH}) & l_{c3}\cos(\theta_{2DH}+\theta_{3DH})\\ 0 & 0\\ 0 & 0\\ 0 & 0\\ 1 & 0\end{bmatrix} \tag{2.9}$$

We can similarly derive the velocity for Joint 3's center of mass as follows:

$$\begin{aligned}\text{Joint 3 Velocity} &= J_{3,velocity}\begin{bmatrix}\dot{\theta}_{2DH}\\ \dot{\theta}_{3DH}\end{bmatrix}\\ &= \begin{bmatrix}\dot{\theta}_{2DH}\big[-l_{c3}\sin(\theta_{2DH}+\theta_{3DH}) - l_2\sin(\theta_{2DH})\big]\\ \dot{\theta}_{3DH}l_{c3}\cos\theta_{(2DH}+\theta_{3DH}) + l_2\cos(\theta_{2DH})\big]\\ 0\end{bmatrix}\end{aligned} \tag{2.10}$$

With the velocities for the center of mass for Joints 2 and 3 computed, we can now calculate the kinetic energy. This is found by computing both the kinetic energy due to linear velocity ($\frac{1}{2}mv^2$) and rotational velocity ($\frac{1}{2}I\dot{\theta}^2$):

$$\begin{aligned}K &= \frac{1}{2}m_2v_2^2 + \frac{1}{2}I_2\dot{\theta}_{2DH}^2 + \frac{1}{2}m_3v_3^2 + \frac{1}{2}I_3(\dot{\theta}_{2DH}+\dot{\theta}_{3DH})^2\\ &= \frac{1}{2}m_2(l_{c2}\dot{\theta}_2DH)^2 + \frac{1}{2}I_2\dot{\theta}_{2DH}^2\\ &+ \frac{1}{2}m_3(l_3\dot{\theta}_{2DH})^2 + \frac{1}{2}m_3l_{c3}^2(\dot{\theta}_{2DH}+\dot{\theta}_{3DH})^2 + l_2l_3\dot{\theta}_{2DH}\cos(\theta_{3DH})(\dot{\theta}_{2DH}+\dot{\theta}_{3DH})\\ &+ \frac{1}{2}I_3(\dot{\theta}_{2DH}+\dot{\theta}_{3DH})^2\end{aligned} \tag{2.11}$$

Substituting the given values for p_1, p_2, and p_3 yields a condensed form for the kinetic energy:

$$
\begin{aligned}
K = & \frac{1}{2}\dot{\theta}_{2DH}^2 p_1 \\
& + (\frac{1}{2}\dot{\theta}_{2DH}^2 + \dot{\theta}_{2DH}\dot{\theta}_{3DH} + \frac{1}{2}\dot{\theta}_{3DH}^2)p_2 \\
& + (\cos(\theta_{3DH})\dot{\theta}_{2DH}^2 + \cos(\theta_{3DH})\dot{\theta}_{2DH}\dot{\theta_{3DH}})p_3
\end{aligned}
\tag{2.12}
$$

The system naturally has potential energy due to gravity, as shown in Figure 1. This potential energy can act on one or both links, depending on the joint angles. It is described fully in Equation 2.13, which ones again takes into account the simplifications provided by Equation 2.1.

$$
\begin{aligned}
V &= -m_2 g l_{c2} \sin(\theta_{2DH}) - m_3 g(l_2 \sin\theta_{2DH} + l_c 3 \sin\theta_{2DH} + \theta_{3DH}) \\
&= -p_4 g \sin(\theta_{2DH}) - p_5 g \sin(\theta_{2DH} + \theta_{3DH})
\end{aligned}
\tag{2.13}
$$

2.2 Dynamic Equations in terms of D-H Thetas

$$
\begin{aligned}
L &= K - V \\
&= \frac{1}{2}\dot{\theta}_{2DH}^2 p_1 \\
&+ (\frac{1}{2}\dot{\theta}_{2DH}^2 + \dot{\theta}_{2DH}\dot{\theta}_{3DH} + \frac{1}{2}\dot{\theta}_{3DH}^2)p_2 \\
&+ (\cos(\theta_{3DH})\dot{\theta}_{2DH}^2 + \cos(\theta_{3DH})\dot{\theta}_{2DH}\dot{\theta_{3DH}})p_3 \\
&+ p_4 g \sin(\theta_{2DH}) + p_5 g \sin(\theta_{2DH} + \theta_{3DH})
\end{aligned}
\tag{2.14}
$$

$$
\tau_i = \frac{d}{dt}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i}
\tag{2.15}
$$

Solving this equation for τ yields:

$$
\begin{aligned}
\tau = & \begin{bmatrix} \ddot{\theta}_{2DH}\big[p_1 + p_2 + 2p_3\cos(\theta_{3DH})\big] & \ddot{\theta}_{3DH}\big[p_2 + p_3\cos(\theta_{3DH})\big] \\ \ddot{\theta}_{2DH}\big[p_2 + p_3\cos(\theta_{3DH})\big] & \ddot{\theta}_{3DH}\big[p_2\big] \end{bmatrix} \\
& + \begin{bmatrix} \dot{\theta}_{2DH}\big[-p_3\sin(\theta_{3DH})\dot{\theta}_{3DH}\big] & \dot{\theta}_{3DH}\big[-p_3\sin(\theta_{3DH})\dot{\theta}_{3DH} - p_3\sin(\theta_{3DH})\dot{\theta}_{2DH}\big] \\ \dot{\theta}_{2DH}\big[p_3\sin(\theta_{3DH})\dot{\theta}_{2DH}\big] & 0 \end{bmatrix} \\
& + \begin{bmatrix} -p_4 g\cos(\theta_{2DH}) - p_5 g\cos(\theta_{2DH} + \theta_{3DH}) \\ -p_5 g\cos(\theta_{2DH} + \theta_{3DH}) \end{bmatrix}
\end{aligned}
\tag{2.16}
$$

It can be seen that this equation contains three distinct parts. Some aspects are dependent on $\ddot{\theta}$. Others are dependent on $\dot{\theta}$. The remaining components are due to gravity. By combining this fact with Equation 2.17, we can arrive at the final derivations for $D(\theta)$, $C(\theta, \dot{\theta})$, and $g(\theta)$.

$$D(\theta)\ddot{\theta} + C(\theta, \dot{\theta})\dot{\theta} + G(\theta) = \tau \tag{2.17}$$

$$\theta = \begin{bmatrix} \theta_{2DH} \\ \theta_{3DH} \end{bmatrix}, \dot{\theta} = \begin{bmatrix} \dot{\theta}_{2DH} \\ \dot{\theta}_{3DH} \end{bmatrix}, \ddot{\theta} = \begin{bmatrix} \ddot{\theta}_{2DH} \\ \ddot{\theta}_{3DH} \end{bmatrix}, \tau = \begin{bmatrix} \tau_{M2DH} \\ \tau_{M3DH} \end{bmatrix} \tag{2.18}$$

$$D(\theta) = \begin{bmatrix} p_1 + p_2 + 2p_3 \cos(\theta_{3DH}) & p_2 + p_3 \cos(\theta_{3DH}) \\ p_2 + p_3 \cos(\theta_{3DH}) & p_2 \end{bmatrix} \tag{2.19}$$

$$C(\theta, \dot{\theta}) = \begin{bmatrix} -p_3 \sin(\theta_{3DH})\dot{\theta}_{3DH} & -p_3 \sin(\theta_{3DH})\dot{\theta}_{3DH} - p_3 \sin(\theta_{3DH})\dot{\theta}_{2DH} \\ p_3 \sin(\theta_{3DH})\dot{\theta}_{2DH} & 0 \end{bmatrix} \tag{2.20}$$

$$G(\theta) = \begin{bmatrix} -p_4 g \cos(\theta_{2DH}) - p_5 g \cos(\theta_{2DH} + \theta_{3DH}) \\ -p_5 g \cos(\theta_{2DH} + \theta_{3DH}) \end{bmatrix} \tag{2.21}$$

Because we have Equation 2.17 and know that the matrix $D(\theta)$ is invertible, we can rearrange the constituent parts to yield a relationship between the dynamics and the acceleration for the DH frame joints.

$$\begin{bmatrix} \ddot{\theta}_{2DH} \\ \ddot{\theta}_{3DH} \end{bmatrix} = D(\theta)^{-1}\tau - D(\theta)^{-1}C(\theta, \dot{\theta})\dot{\theta} - D(\theta)^{-1}G(\theta) \tag{2.22}$$

$$\begin{aligned} x_1 &= \theta_{2DH} & \dot{x}_1 &= x_2 \\ x_2 &= \dot{\theta}_{2DH} & \dot{x}_2 &= \ddot{\theta}_{2DH} \\ x_3 &= \theta_{3DH} & \dot{x}_3 &= x_4 \\ x_4 &= \dot{\theta}_{3DH} & \dot{x}_4 &= \ddot{\theta}_{3DH} \end{aligned} \tag{2.23}$$

2.3 Dynamic Equations in terms of Motor Thetas

$$\theta_{2DH} = \theta_{2M} - \frac{\pi}{2} \tag{2.24}$$

$$\theta_{3DH} = \theta_{3M} - \theta_{2M} + \frac{\pi}{2} \tag{2.25}$$

$$\tau_{2DH} = \tau_{2M} + \tau_{3M} \tag{2.26}$$

$$\tau_{3DH} = \tau_{3M} \tag{2.27}$$

$$D(\theta)\ddot{\theta} + C(\theta, \dot{\theta})\ddot{\theta} + G(\theta) = \tau \tag{2.28}$$

$$\theta = \begin{bmatrix} \theta_{2M} \\ \theta_{3M} \end{bmatrix}, \dot{\theta} = \begin{bmatrix} \dot{\theta}_{2M} \\ \dot{\theta}_{3M} \end{bmatrix}, \ddot{\theta} = \begin{bmatrix} \ddot{\theta}_{2M} \\ \ddot{\theta}_{3M} \end{bmatrix}, \tau = \begin{bmatrix} \tau_{M2} \\ \tau_{M3} \end{bmatrix} \tag{2.29}$$

$$D(\theta) = \begin{bmatrix} p_1 & -p_3 \sin(\theta_{3M} - \theta_{2M}) \\ -p_3 \sin(\theta_{3M} - \theta_{2M}) & p_2 \end{bmatrix} \tag{2.30}$$

$$C(\theta, \dot{\theta}) = \begin{bmatrix} 0 & -p_3 \cos(\theta_{3M} - \theta_{2M})\dot{\theta}_{3M} \\ p_3 \cos(\theta_{3M} - \theta_{2M})\dot{\theta}_{2M} & 0 \end{bmatrix} \tag{2.31}$$

$$G(\theta) = \begin{bmatrix} -p_4 g \sin(\theta_{2M}) \\ -p_5 g \cos(\theta_{3M}) \end{bmatrix} \tag{2.32}$$

$$\begin{bmatrix} \ddot{\theta}_{2M} \\ \ddot{\theta}_{3M} \end{bmatrix} = D(\theta)^{-1}\tau - D(\theta)^{-1}C(\theta, \dot{\theta})\dot{\theta} - D(\theta)^{-1}G(\theta) \tag{2.33}$$

$$\begin{aligned} x_1 &= \theta_{2M}, & \dot{x}_1 &= x_2, \\ x_2 &= \dot{\theta}_{2M}, & \dot{x}_2 &= \ddot{\theta}_{2M}, \\ x_3 &= \theta_{3M}, & \dot{x}_3 &= x_4, \\ x_4 &= \dot{\theta}_{3M} & \dot{x}_4 &= \ddot{\theta}_{3M} \end{aligned} \tag{2.34}$$

3 Simulation

3.1 Simulink Model

A model of the CRS robot was given in the lab manual, along with five parameters that were experimentally derived by the course staff. We directly implemented this model in Simulink according to Figure 2. The implementation of the function that outputted qddot in the Simulink model is given in Listing 1.

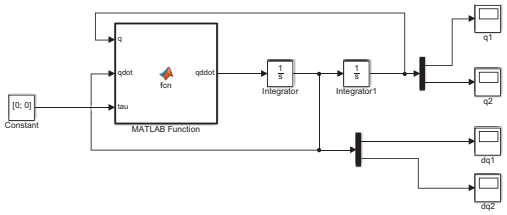

Figure 2: Simulink Model of CRS Robot
(Figure courtesy of lab manual)

```
function qddot = fcn(q,qdot,tau)
g = 9.81;
p = [0.0300, 0.0128, 0.0076, 0.0753, 0.0298];
D = [ p(1), -p(3)*sin(q(2)-q(1)); -p(3)*sin(q(2)-q(1)), p(2)];
C = [0 , -p(3)*cos(q(2)-q(1))*qdot(2); p(3)*cos(q(2)-q(1))*qdot(1) , 0];
G = [-p(4)*g*sin(q(1)); -p(5)*g*cos(q(2))];

qddot = D\(tau-C*qdot-G);
```

Listing 1: Implemented Simulink Function

3.2 Velocity Filtering

A raw velocity given by $\dot{\theta} = \frac{\theta_{current} - \theta_{previous}}{T}$ can serve as a discretized approximation of the true motor velocity, however tends to be noisy. As such, a filter to reduce this noise is needed to have a viable signal. One simple technique to accomplish this noise reduction is via an averaging filter, which takes the mean of several prior values along with the current measured input in order to derive a filtered velocity.

Two implementations for an averaging filter appear in the lab manual; These methods are shown in Listings 2 and 3. It can be seen that both filtering methods are of order 2 and

save prior values. In lines 15 and 16 of Listing 2, the $n-1$ value is saved as $n-2$, and n is saved as $n-1$. This order is important, as otherwise the $n-1$ value would be overwritten prior to being able to save it as $n-2$.

The major difference between the two implementations comes in the form of what *prior values* are saved. In Method 1, the prior raw velocity values are saved. This forms a Finite Impulse Response (FIR) filter, as no *filtered values* are saved. As such, the filter does not have any memory of past filter response.

```
1  float Theta1_old = 0;
2  float Omega1_raw = 0;
3  float Omega1_old1 = 0;
4  float Omega1_old2 = 0;
5  float Omega1 = 0;
6
7  // This function is called every 1 ms
8  void lab(float thetamotor1,float thetamotor2,float thetamotor3,float *tau1,
       float *tau2,float *tau3)
9  {
10     Omega1_raw = (thetamotor1 - Theta1_old)/0.001;
11     Omega1 = (Omega1_raw + Omega1_old1 + Omega1_old2)/3.0;
12
13     Theta1_old = thetamotor1;
14
15     //order matters here. Why??
16     Omega1_old2 = Omega1_old1;
17     Omega1_old1 = Omega1_raw;
18
19 }
```

Listing 2: Velocity Filtering Method 1

On the other hand, Method 2 is an Infinite Impulse Response (IIR) filter, as it saves the prior *filtered values* for use in the current filter's input. This technique relies on memory of prior filter response.

```
1  float Theta1_old = 0;
2  float Omega1_old1 = 0;
3  float Omega1_old2 = 0;
4  float Omega1 = 0;
5
6  // This function is called every 1 ms
7  void lab(float thetamotor1,float thetamotor2,float thetamotor3,float *tau1,
       float *tau2,float *tau3)
8  {
9      Omega1 = (thetamotor1 - Theta1_old)/0.001;
10     Omega1 = (Omega1 + Omega1_old1 + Omega1_old2)/3.0;
11
12     Theta1_old = thetamotor1;
13
14     //order matters here. Why??
```

```
    Omega1_old2 = Omega1_old1;
    Omega1_old1 = Omega1

}
```

Listing 3: Velocity Filtering Method 2

The Infinite Impulse Response method (Listing 3) is what we implemented, as it offers better low-pass filtering ability at a lower filter order (2^{nd} order) than what would be necessary to get similar performance out of the FIR method. Our implementation was done in the form of a function that reads and modifies global variables. This function is shown in Listing 4.

```
void filter_velocity() {
    int i;
    for(i = 0; i < 3; i++) {
        Omega[i] = (theta_motor[i] - Theta_old[i])/0.001;
        Omega[i] = (Omega[i] + Omega_old1[i] + Omega_old2[i])/3.0;
        Theta_old[i] = theta_motor[i];

        Omega_old2[i] = Omega_old1[i];
        Omega_old1[i] = Omega[i];
    }
}
```

Listing 4: Implemented Velocity Filtering Method 2

A Simulink implementation of this filter can also be seen in Figure 3.

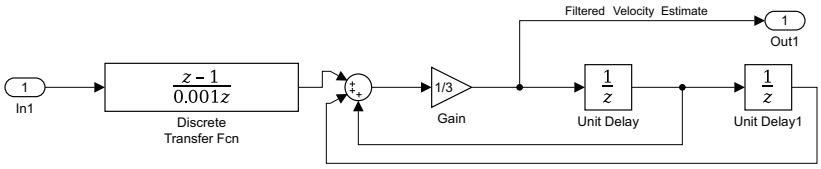

Figure 3: Discrete Velocity Filtering Subsystem

4 Simulation Link 2 and Link 3 PD Control of CRS Robot

By extending our previous Simulink model, we can simulate proportional-derivative control of joints 2 and 3 easily. We started by writing a PD controller that takes a $\theta_{desired}$ and θ_{actual} as inputs, filters the velocity using the aforementioned 2^{nd} order IIR filter, and then outputs a τ control effort. The final PD control subsystem can be seen in Figure 4.

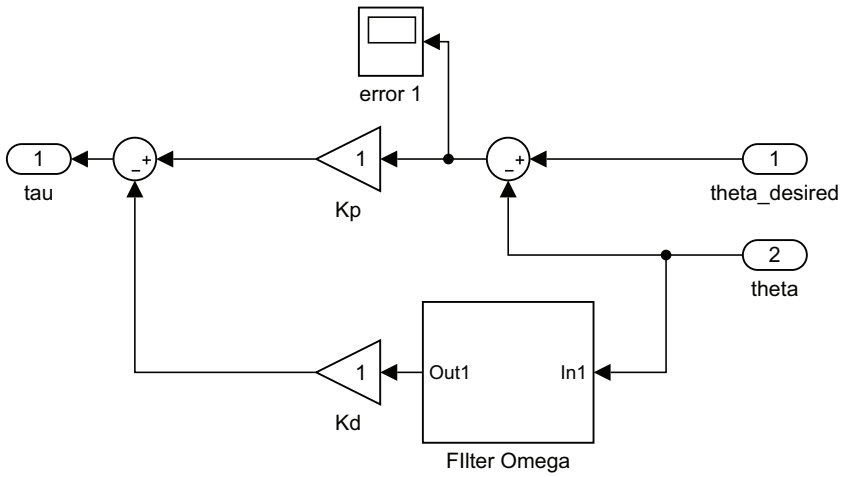

Figure 4: PD Controller Subsystem

Figure 5 depicts the combined system controller that Simulink utilized for simulations. Both θ values were scoped out in order to get a visualization of system response. The step response for θ_1 (Link 2) can be seen in Figure 6, and the corresponding error term, e_1, is shown in Figure 7.

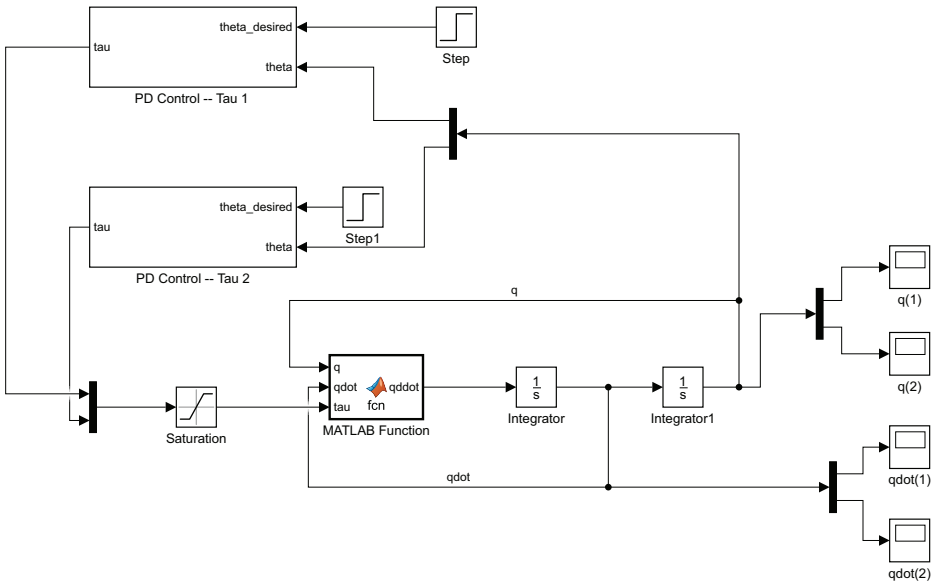

Figure 5: Combined System Controller

Likewise, the step response for θ_2 (Link 3) is shown in Figure 8. The error term e_2 for θ_2 can be seen in Figure 9.

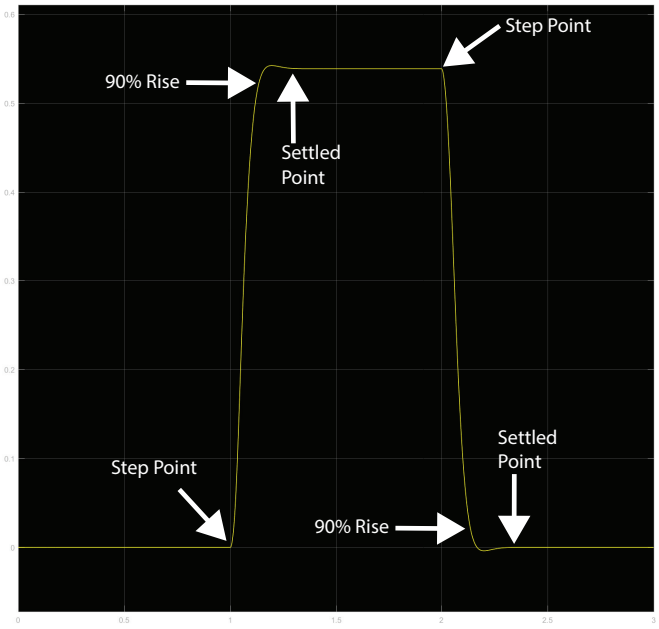

Figure 6: Step Response for q1

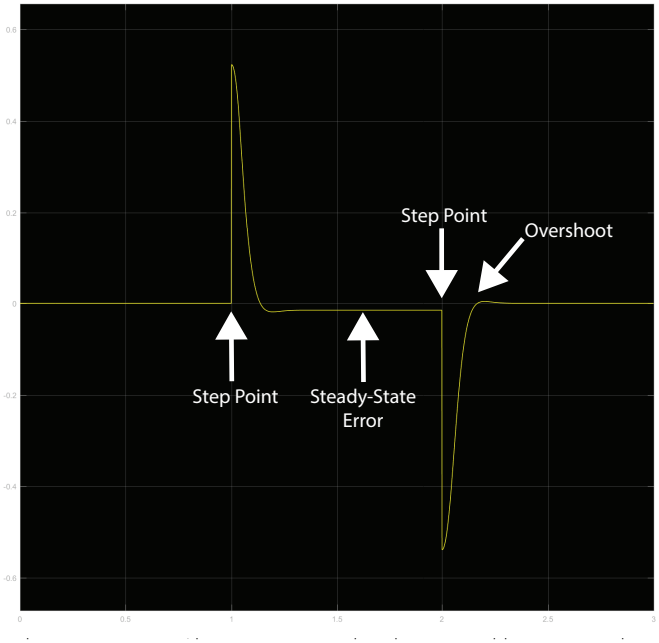

Figure 7: Tracking Error for q1

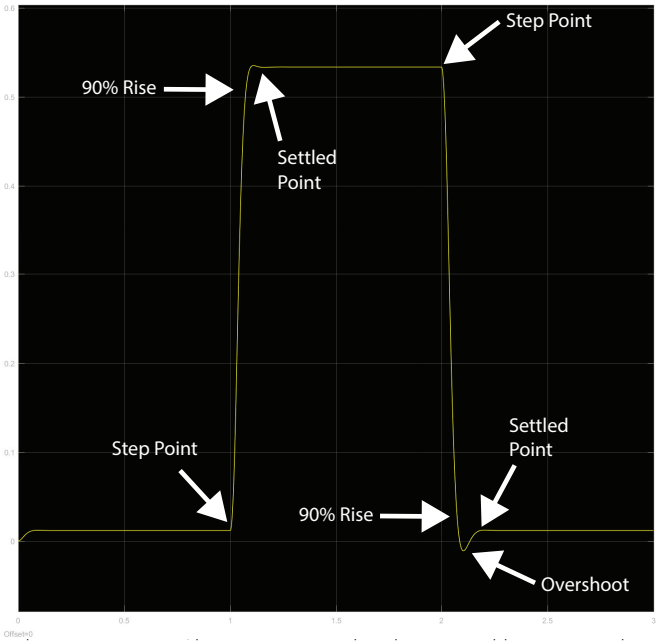

Figure 8: Step Response for q1

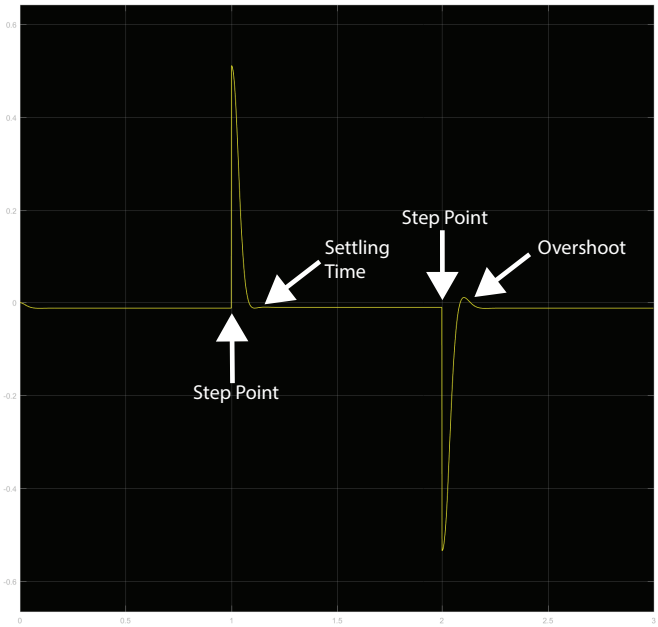

Figure 9: Tracking Error for q2

An annotated representation of the actual output torque values, t_1 and t_2, for Link 2 and Link 3 respectively, can be seen in Figure 10.

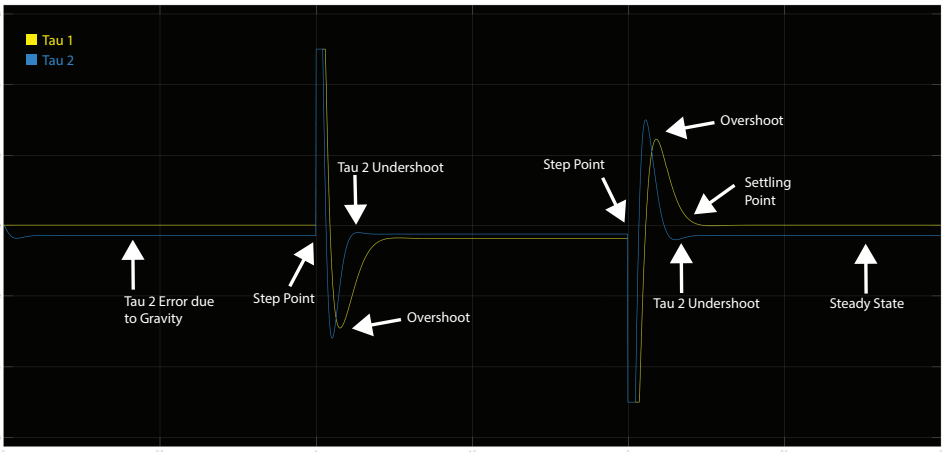

Figure 10: Torque

5 PD and PID Control of CRS Robot

5.1 PD Control Implementation

Implementing the Proportional-Derivative control on the CRS robot started out with transferring our Simulink implementation into C code via a control function we wrote that operates on global variables and is called in the lab() function every 1ms. Once written, torque outputs were measured in Simulink while the robot arm followed the same step trajectories as our previous simulations did. We tuned the PD gains (k_p and k_d, respectively) to achieve under 1% overshoot, minimal steady-state error, and under 300ms rise time. The results of the tuning can be seen in Figure 11.

$$\begin{aligned} k_{p,1} &= 40 \qquad k_{d,1} = 2.5 \\ k_{p,2} &= 15 \qquad k_{d,2} = 1.3 \\ k_{p,3} &= 30 \qquad k_{d,3} = 1.4 \end{aligned} \tag{5.1}$$

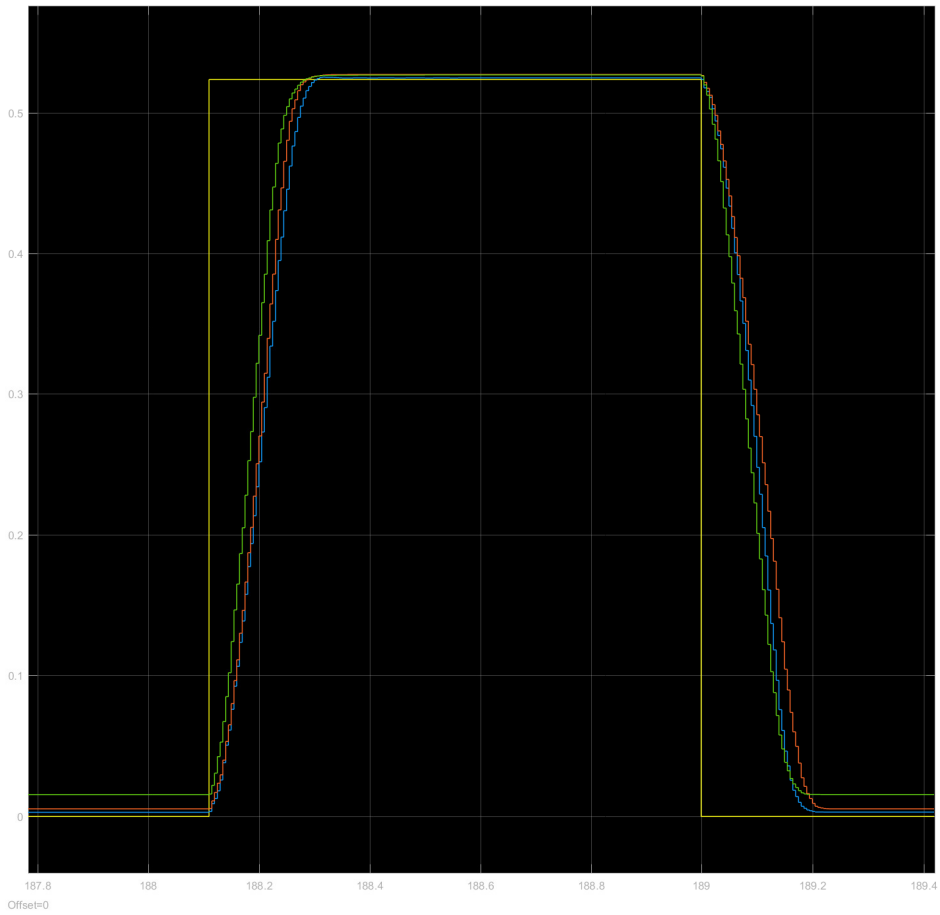

Figure 11: PD Control Response

5.2 Integral Approximation methods

Our digital PID controller requires an estimation of the integral in order to function. We utilized a trapezoidal approximation of the error integration. The formula is shown in Equation 5.2 and functions by summing the prior integral value with a "slice" of the of the integral given by the difference in error values multiplied by the time difference between the error value measurements.

$$I_K = I_{K-1} + \frac{e_K + e_{K-1}}{2} * T \tag{5.2}$$

The trapezoidal method provides a smooth error integral estimate, however integrals are prone to a phenomenon known as windup, where big variations in the input signal causes

large error accumulations. This can cause the PID loop to overshoot excessively. In order to combat this problem, we implemented two methods of windup mitigation.

The first method we used is to only calculate the integral term in the PID loop when the tracking value (θ_d) is within a specific threshold of the θ_{motor} value. During all other times, only PD control is utilized and the integral term is set to zero (thus negating windup). We choose a threshold of 10% using experimental results.

The second method we used was to clamp the integral sum when the output torque saturated at ± 5. This ensured that windup would not cause the control signal to go beyond the viable bounds of the physical system (the limits of torque the motors on the robot can produce).

5.3 Integral Control

With integral control properly implemented and windup mitigated, it was necessary to tune all gains (including re-tuning the k_p and k_d gains). Thankfully, the added integral control was able to bring the steady-state error to zero. In our experience, only minor changes to the derivative gain k_d were necessary, however we saw larger changes to be needed for the k_p proportional gains.

We found best results with very large integral gain values. Since the integral control term only turns on when the PD loop gets "close" to the desired signal, high k_i values allowed for the integral component to very quickly reduce error while not interfering with the rise time greatly. This also explains why only subtle changes to the proportional gains were needed. Our final results can be seen in Figure 12.

$$\begin{array}{lll} k_{p,1} = 45 & k_{d,1} = 2.0 & k_{i,1} = 100 \\ k_{p,2} = 60 & k_{d,2} = 2.0 & k_{i,2} = 275 \\ k_{p,3} = 55 & k_{d,3} = 1.7 & k_{i,3} = 185 \end{array} \tag{5.3}$$

6 PID & Feed-Forward Control

6.1 Feed-Forward PID Controller Implementation

Until now, we have only considered position tracking where a control loop compensates for disturbances. This works for simple cases where movements are small and effectively constant, however in order to follow time-varying trajectories, we must introduce the concept of feedforward control. Feedforward control makes use of a transfer function between

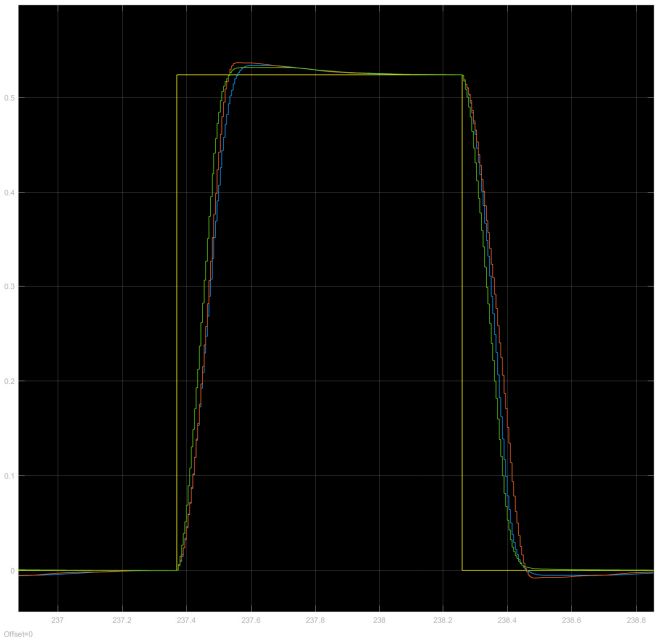

Figure 12: PID Control Response

the reference input and is superimposed on the output of the compensator that we previously utilized. This concept effectively allows a control system to respond to a given input independently of any system response to disturbances. This works well for time-varying trajectories, as it gives the system a chance to respond to the trajectory changes prior to handling any disturbances in the output. Mathematically, the benefits come in the form of enhanced stability and tracking accuracy. In order to add this feedforward concept to our PID loop, Equation 6.1 was used with coefficients J1=0.0167, J2 = 0.03 and J3 = 0.0128, all of which were given in the lab manual.

$$\tau = J\ddot{\theta}^d + K_p(\theta^d - \theta) + K_I \int (\theta^d - \theta) + K_D(\dot{\theta}^d - \dot{\theta}) \tag{6.1}$$

It can be seen that this control law flips the sign of the derivative gain K_D. This is due to the addition of desired derivative tracking. The feed-forward control law's full derivative term is $K_D(\dot{\theta}^d - \dot{\theta})$. If no desired velocity is given, this reduces to $K_D(0 - \dot{\theta})$, which is equivalent to $-K_D\dot{\theta}$, as shown in our previous PID implementations.

The individual joint trajectory responses can be seen in Figures 13, 14, and 15. The Feedforward PID control law as a system is shown in Figure 16.

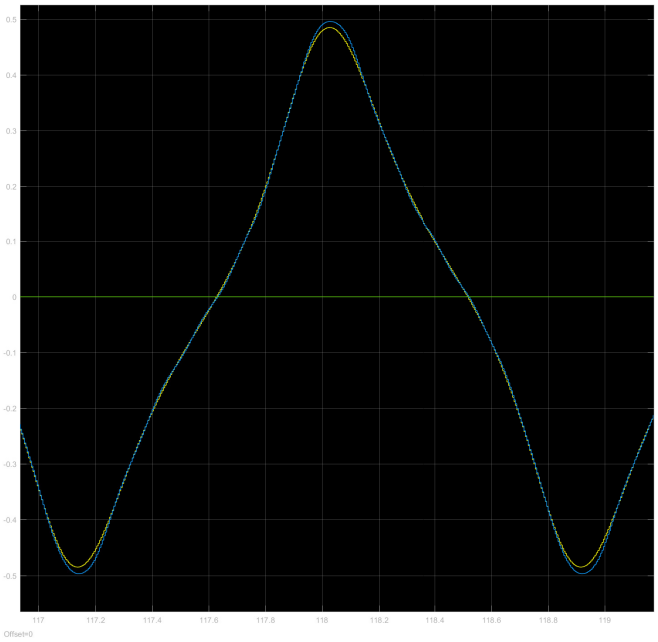

Figure 13: q1 Trajectory Response

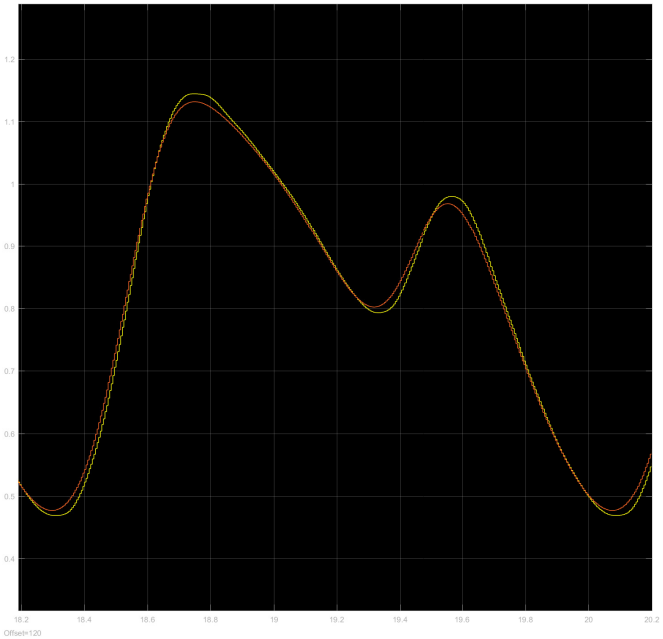

Figure 14: q2 Trajectory Response

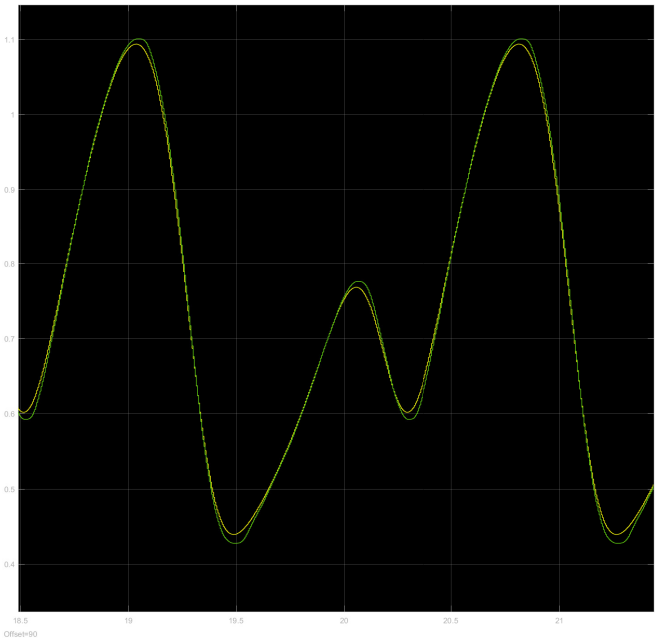

Figure 15: q3 Trajectory Response

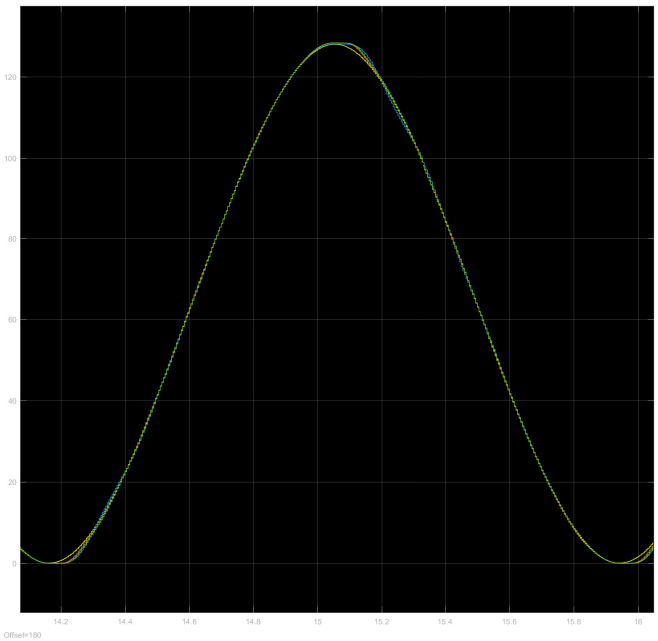

Figure 16: PID with Feedforward Response

7 Trajectory Following with Repetition

In order to generate a smooth trajectory for testing, we followed Section 5.5.2 in Reference [1], which gave out instructions on finding the coefficients for cubic polynomials. The cubic functions that appear in Equation 5.21 in Reference [1] are reproduced below in Equation 7.1.

$$q_0 = a_0 + a_1 t_0 + a_2 t_0^2 + a_3 t_0^3 \tag{7.1}$$
$$v_0 = a_1 + 2a_2 t_0 + 3a_3 t_0^2 \tag{7.2}$$
$$q_f = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 \tag{7.3}$$
$$v_f = a_1 + 2a_2 t_f + 3a_3 t_f^2 \tag{7.4}$$

Two sets of coefficients were needed. The first polynomial described the up-facing travel, and the second described the down-facing travel. Solving for both sets of coefficients yields the results shown in Equation 7.5, which were used to form the final trajectory paths described in Equation 7.6.

$$\begin{aligned} a_{up,0} &= 0 & a_{down,0} &= -2 \\ a_{up,1} &= 0 & a_{down,1} &= 6 \\ a_{up,2} &= 1.5 & a_{down,2} &= -4.5 \\ a_{up,3} &= -1 & a_{down,3} &= 1 \end{aligned} \tag{7.5}$$

$$\begin{aligned} \theta_d &= 1.5t^2 - t^3 & \theta_d &= -2 + 6t - 4.5t^2 + t^3 \\ \dot{\theta_d} &= 3t - 3t^2 & \dot{\theta_d} &= 6 - 9t + 3t^2 \\ \ddot{\theta_d} &= 3 - 6t & \ddot{\theta_d} &= -9 + 6t \end{aligned} \tag{7.6}$$

We utilized Matlab to help solve for these coefficients. The code we utilized is available in Listings 5 and 6 below.

```
function [ theta_d, theta_dot_d, theta_ddot_d ] = position_d( t )
%TRAJ_GEN Summary of this function goes here
%   Detailed explanation goes here
    if t > 2
            theta_d = 0;
            theta_dot_d = 0;
            theta_ddot_d = 0;
```

```
    elseif t < 1 %% upswing
            theta_d = 1.5*t^2 -t^3;
            theta_dot_d = 3*t - 3*t^2;
            theta_ddot_d = 3 - 6*t;
    elseif t >= 1 %%downswing
            theta_d = -2 + 6*t - 4.5*t^2 + t^3;
            theta_dot_d = 6 - 9*t +3*t^2;
            theta_ddot_d = -9 +6*t;
    end
end
```

Listing 5: Desired Theta Values for Time t

```
%%
clear;
clc;

t = linspace(0,2);

%%
theta_d = zeros(size(t));
theta_dot_d = zeros(size(t));
theta_ddot_d = zeros(size(t));
%%
for i = 1:size(t,2)
    [a, b, c] = position_d(t(i));
    theta_d(i) = a;
    theta_dot_d(i) = b;
    theta_ddot_d(i) = c;
end

plot(t,theta_ddot_d);
```

Listing 6: Plotting Trajectory θ's

For our arbitrary trajectory, we utilized Bernoulli's Lemniscate (Equation 7.7 with $a = 5$) to create a figure-8 in the x-y plane. We then had the z-trajectory follow a sinusoid as shown in Equation 7.8. A simulation of the 3D trajectory can be seen in Figure 17.

$$x = \frac{a\sqrt{2}\cos \pi t}{\sin^2 \pi t + 1} + 14, \qquad y = \frac{a\sqrt{2}\cos \pi t \sin \pi t}{\sin^2 \pi t + 1} \tag{7.7}$$

$$z = 3\sin t + 10 \tag{7.8}$$

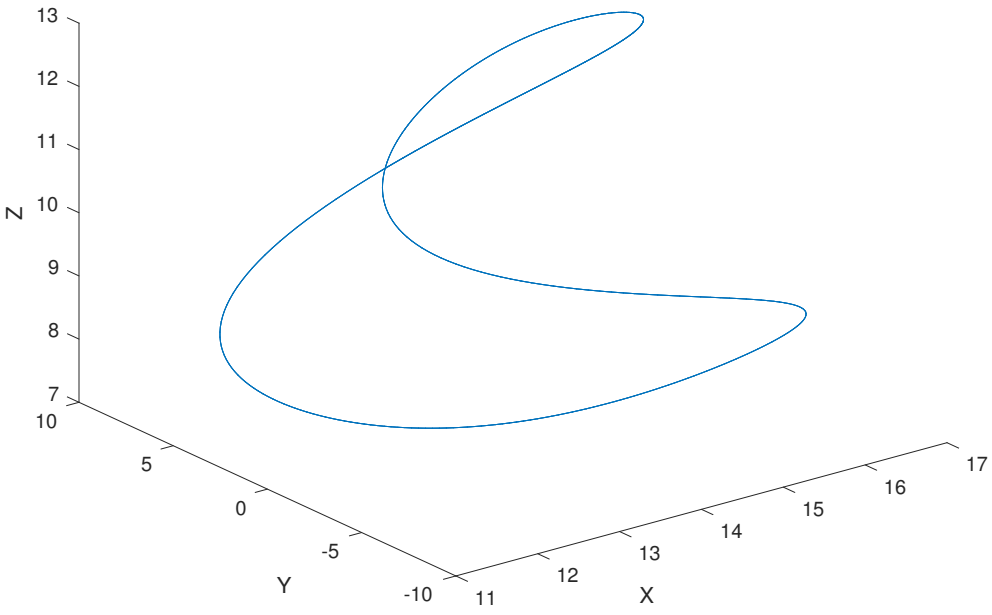

Figure 17: 3D Trajectory

8 Conclusion

8.1 Differences in Control Methods

This lab saw the implementation of three different control methods: PD Control, PID Control, and PID with Feedforward Control. The latter two techniques implemented integral terms in the control loops, so they were able to bring the steady-state error to zero for all cases, unlike the rudimentary Proportional-Derivative method. Both PD and PID control allowed for position tracking, however PID with Feedforward granted the ability to track both position *and* velocity simultaneously. This added velocity tracking gave way to better trajectory following, as it ensured that the end-effector experienced smooth, non-abrupt movements.

References

[1] M.W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. Wiley, 2005. ISBN: 9780471649908. URL: https://books.google.com/books?id=wGapQAAACAAJ.

A lab.c code

```
// code goes here
#include <tistdtypes.h>
#include <coecsl.h>
#include "user_includes.h"
#include "math.h"



// These two offsets are only used in the main file user_CRSRobot.c  You just need to create them here and find the correct offset and then these offset will adjust the encoder readings
//float offset_Enc2_rad = -0.37;
//float offset_Enc3_rad = 0.27;

float offset_Enc2_rad = -0.427257;
float offset_Enc3_rad = 0.230558;


// Your global variables.

long mycount = 0;

#pragma DATA_SECTION(whattoprint, ".my_vars")
float whattoprint = 0.0;

#pragma DATA_SECTION(theta1array, ".my_arrs")
float theta1array[100];

#pragma DATA_SECTION(theta2array, ".my_arrs")
float theta2array[100];

#pragma DATA_SECTION(theta3array, ".my_arrs")
float theta3array[100];


long arrayindex = 0;

float printtheta1motor = 0;
float printtheta2motor = 0;
float printtheta3motor = 0;

float x_pos = 0;
float y_pos = 0;
float z_pos = 0;

// inverse kinematics
float theta_1 = 0;
float theta_2 = 0;
float theta_3 = 0;
```

```
49
50 float motor_theta_1 = 0;
51 float motor_theta_2 = 0;
52 float motor_theta_3 = 0;
53
54 // Assign these float to the values you would like to plot in Simulink
55 float Simulink_PlotVar1 = 0;
56 float Simulink_PlotVar2 = 0;
57 float Simulink_PlotVar3 = 0;
58 float Simulink_PlotVar4 = 0;
59
60 //Controller Parameters
61 //Proportional
62 float kp[3] = {110, 130, 55};
63
64 //Derivative
65 float kd[3] = {2,2,1.7};
66
67 //Intergral
68 float ki[3] = {550,600,185};
69
70 // velocity filtering
71 float Theta_old[3] = {0,0,0};
72 float Omega_old1[3] = {0,0,0};
73 float Omega_old2[3] = {0,0,0};
74 float Omega[3] = {0,0,0};
75
76 //Integral Estimation
77 float Ik[3] = {0,0,0};
78 float Ik_old[3] = {0,0,0};
79 float e_old[3] = {0,0,0};
80
81 float integral_threshold = 0.1;
82
83
84 // current positions
85 float theta_motor[3] = {0,0,0};
86
87 //current tau
88 float t[3] = {0,0,0};
89
90 //feedfoward control
91 float theta_d[3] = {0,0,0};
92 float theta_dot_d[3] = {0,0,0};
93 float theta_ddot_d[3] = {0,0,0};
94
95 float J[3] = {0.0167, 0.03, 0.0128};
96
97
98 //This function calculates the motor positions given a desired x,y,and z
       position
99 void inverse_kinematics(float x, float y, float z){
```

```
100
101     //The DH angles are calculated using a geometric analysis of the possible
        configurations of the robot
102     theta_1 = atan2(y,x);
          //The base DH angle is the inverse tangent between the x and y
        coordinates
103     theta_3 = 3.14159 - acos(-(pow(x,2)+pow(y,2)+pow(z-10,2)-200)/200);
          //The elbow DH angle is calculated using the law of cosines
104     theta_2 = -(atan2(z-10,sqrt(pow(x,2)+pow(y,2))) + theta_3/2);
          //The shoulder DH angle is calculated using Pythagoras' Theorem and the
        half angle formula
105
106     //The DH angles must be converted to motor theta angles using appropriate
        transformation
107     theta_d[0] = theta_1;
108     theta_d[1] = theta_2 + PI/2;
109     theta_d[2] = theta_3 + theta_2;
110 }
111
112 void filter_velocity() {
113     int i;
114     for(i = 0; i < 3; i++) {
115         Omega[i] = (theta_motor[i] - Theta_old[i])/0.001;
116         Omega[i] = (Omega[i] + Omega_old1[i] + Omega_old2[i])/3.0;
117         Theta_old[i] = theta_motor[i];
118
119         Omega_old2[i] = Omega_old1[i];
120         Omega_old1[i] = Omega[i];
121     }
122 }
123
124 void estimate_integral() {
125     int i;
126     for(i=0;i<3;i++){
127         float e_k = theta_d[i] - theta_motor[i];
128         Ik[i] = Ik_old[i] + (e_old[i] + e_k)/2*0.001;
129         Ik_old[i] = Ik[i];
130         e_old[i] = e_k;
131     }
132 }
133
134
135
136 void pid_control() {
137     int i = 0;
138     for (i = 0; i < 3; i++){
139         t[i] = kp[i]*(theta_d[i]-theta_motor[i]) - kd[i]*Omega[i];
140         if (fabs(theta_d[i] - theta_motor[i]) < integral_threshold){
141             estimate_integral();
142             t[i] = t[i] + ki[i]*Ik[i];
143         } else {
144             Ik[i] = 0;
```

```
145             Ik_old[i] = 0;
146         }
147 
148 
149         if (t[i] >= 5) {
150             t[i] = 5;
151             Ik[i] = Ik_old[i];
152         }
153         else if (t[i] <= -5) {
154             t[i] = -5;
155             Ik[i] = Ik_old[i];
156         }
157     }
158 }
159 
160 void trajectory(float time){
161     //lemniscate
162     float x_d = 5*sqrt(2)*cos(PI*time)*sin(PI*time)/(sin(PI*time)*sin(PI*time)
        + 1) + 14;
163     float y_d = 5*sqrt(2)*cos(PI*time)/(sin(PI*time)*sin(PI*time) + 1);
164     float z_d = 3*sin(PI*time)+ 10;
165 
166 
167     inverse_kinematics(x_d,y_d,z_d);
168 }
169 
170 //void position_d(float t) {
171 //    float i = 0;
172 //    for (i = 0; i < 3; i++){
173 //        if((t >= 0) && (t < 1)) {
174 //            theta_d[i] = 1.5*pow(t,2) -pow(t,3);
175 //            theta_dot_d[i] = 3*t - 3*pow(t,2);
176 //            theta_ddot_d[i] = 3 - 6*t;
177 //        }
178 //        else if ((t >= 1) && (t <=2 )) {
179 //            theta_d[i] = -2 + 6*t - 4.5*pow(t,2) + pow(t,3);
180 //            theta_dot_d[i] = 6 - 9*t +3*pow(t,2);
181 //            theta_ddot_d[i] = -9 +6*t;
182 //        }
183 //        else {
184 //            theta_d[i] = 0;
185 //            theta_dot_d[i] = 0;
186 //            theta_ddot_d[i] = 0;
187 //        }
188 //    }
189 //
190 //}
191 
192 void feedforward_control() {
193     int i = 0;
194     for (i = 0; i < 3; i++){
```

```
195         t[i] = kp[i]*(theta_d[i]-theta_motor[i]) + kd[i]*(theta_dot_d[i] -
    Omega[i]) + J[i]*theta_ddot_d[i];
196         if (fabs(theta_d[i] - theta_motor[i]) < 0.05){
197             estimate_integral();
198             t[i] = t[i] + ki[i]*Ik[i];
199         } else {
200             Ik[i] = 0;
201             Ik_old[i] = 0;
202         }
203         if (t[i] >= 5) {
204             t[i] = 5;
205             Ik[i] = Ik_old[i];
206         }
207         else if (t[i] <= -5) {
208             t[i] = -5;
209             Ik[i] = Ik_old[i];
210         }
211     }
212
213 }
214
215
216
217
218
219 //This function calculates the forward kinematics of the manipulator given the
     motor positions
220
221 void forward_kinematics(float motor1, float motor2, float motor3){
222
223     //The forward kinematics function uses the translational vector from the
    full DH matrix calculated in Robotica
224     x_pos = 10*cos(motor1)*(cos(motor3)+sin(motor2));
225     y_pos = 10*sin(motor1)*(cos(motor3)+sin(motor2));
226     z_pos = 10*(1+cos(motor2)-sin(motor3));
227
228     }
229
230 // temoprary
231 float desired_value = 0;
232
233
234
235 // This function is called every 1 ms
236 void lab(float theta1motor,float theta2motor,float theta3motor,float *tau1,
    float *tau2,float *tau3, int error) {
237
238 //    theta_motor = {theta1motor, theta2motor, theta3motor};
239     theta_motor[0] = theta1motor;
240     theta_motor[1] = theta2motor;
241     theta_motor[2] = theta3motor;
242
```

```
243     filter_velocity();
244
245
246 //     position_d((mycount%2000)/1000.0);
247 //
248 //     feedforward_control();
249
250     float time = (mycount%2000)/1000.0;
251
252     trajectory(time);
253
254     pid_control();
255
256     *tau1 = t[0];
257     *tau2 = t[1];
258     *tau3 = t[2];
259
260
261     Simulink_PlotVar1 = theta_d[2];
262     //Simulink_PlotVar2 = theta_motor[0];
263     //Simulink_PlotVar3 = theta_motor[1];
264     Simulink_PlotVar4 = theta_motor[2];
265
266
267     //Motor torque limitation(Max: 5 Min: -5)
268
269     // save past states
270     if ((mycount%50)==0) {
271
272         theta1array[arrayindex] = theta1motor;
273         theta2array[arrayindex] = theta2motor;
274
275         if (arrayindex >= 100) {
276             arrayindex = 0;
277         } else {
278             arrayindex++;
279         }
280
281     }
282
283     /*
284      * Forward Kinematics
285      *
286      * based on motor angles
287      */
288
289     forward_kinematics(theta1motor, theta2motor, theta3motor);
290
291     /*
292      * Inverse Kinematics
293      *
294      * based on {x,y,z} pos calculated above
```

```
295       */
296
297      inverse_kinematics(x_pos, y_pos, z_pos);
298
299
300      if ((mycount%500)==0) {
301          if (whattoprint > 0.5) {
302              serial_printf(&SerialA, "I love robotics\n\r");
303          } else {
304              printtheta1motor = theta1motor;
305              printtheta2motor = theta2motor;
306              printtheta3motor = theta3motor;
307
308
309              SWI_post(&SWI_printf); //Using a SWI to fix SPI issue from sending
       too many floats.
310          }
311          GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1; // Blink LED on Control Card
312          GpioDataRegs.GPBTOGGLE.bit.GPIO60 = 1; // Blink LED on Emergency Stop
      Box
313      }
314
315
316      mycount++;
317  }
318
319  void printing(void){
320  //  serial_printf(&SerialA, "%.2f %.2f,%.2f   \n\r",printtheta1motor,
      printtheta2motor,printtheta3motor);
321  //  serial_printf(&SerialA, "x: %.2f,   y: %.2f,  z: %.2f   \n\r",x_pos,y_pos,
      z_pos);
322  //  //serial_printf(&SerialA, "Estimated IK solution: theta1: %.2f,   theta2:
      %.2f,  theta3: %.2f   \n\r",theta_1,theta_2,theta_3);
323  //    serial_printf(&SerialA, "Estimated IK solution: motor_theta1: %.2f,
      motor_theta2: %.2f,  motor_theta3: %.2f   \n\r",motor_theta_1,
      motor_theta_2,motor_theta_3);
324
325  //    serial_printf(&SerialA, "theta1motor:  %.2f \n\r",theta1motor);
326  //    serial_printf(&SerialA, "theta_motor[0]:  %.2f \n\r",theta_motor[0]);
327  }
```

Listing 7: lab.c