ME 446 Laboratory #2
PD, PID and Feedforward Joint Control
Lab sessions will be held the weeks of
Group A: March 1st and March 15th
Group B: March 8th and March 22nd

Objectives

- Learn how to implement velocity and integration calculations.
- Implement and tune PD joint control for the three CRS robot joints.
- Add integral control to the PD joint control.
- Implement and tune PD plus Feedforward Control.
- Implement and tune PD plus Gravity Compensation Control and Compare to PD plus Feedforward Control.
- Using joints 1, 2 and 3 design a task space trajectory to make the CRS robot’s end effect trace a figure eight or something else fun.
Part 1: How to Calculate and Implement Velocity needed in the PID Controller.

1.1 How to find velocity.

The CRS robot has sensors (optical encoders) to measure the angle at each of its joints, but it does not have a velocity sensor. Taking advantage of the fact that our lab() function is called exactly every 1ms, current and previous angle measurements can be used to find and an approximation of the velocity of the system. Global variables can be used to save previous sampled thetas and previous velocity calculations. Velocity can then be calculated using the simple equation:

$$\dot{\theta} = \frac{\theta_{\text{current}} - \theta_{\text{previous}}}{T}.$$  We will call this “raw” velocity. It is a good approximation but can be quite noisy. Normally some filtering is performed on this raw velocity calculation to get rid of some of the noise. But too much filtering causes phase delay which can hinder your control implementation. In my experience 2nd or 3rd order filters is the limit for velocity filtering. More complicated filters such as a Butterworth filter can be used but many times a simple averaging filter will suffice. There are also Laplace domain derivative approximations like $100s/(s+100)$ that can be discretized using the Tustin rule (same as the Trapezoidal integration rule) to achieve a discrete velocity approximation. This requires some digital control knowledge. (See Appendix if you wish) So unless you have taken a class in digital control I do not recommend you use this method for this lab.

We will initially use averaging to filter the raw velocity calculation and only try the other mentioned ideas if we find the filter velocity to still be too noisy. To get an idea on how to implement this averaging filter, study the next two partial listings of C code. Explain to your TA, using terms like finite and infinite data (check out FIR and IIR links listed in the appendix below), what the difference is between these two methods. Which do you think filters the velocity better? Remember that the Lab() function is called every 1ms. Start out by implementing your velocity equations for $\theta_{1M}$ and manually move the waist joint. Plot $\theta_{1M}$, $\theta_{1M}$’s calculated “raw” angular velocity and $\theta_{1M}$’s calculated filtered angular velocity, real-time in Simulink. Look at Lab 1 to remember how to plot real-time in Simulink. Once you have $\theta_{1M}$’s velocity working, implement the same equations to find $\theta_{2M}$ and $\theta_{3M}$’s velocity. **Demo these working to your TA.**
**First Method of Filtering Velocity**

```c
float Theta1_old = 0;
float Omega1_raw = 0;
float Omega1_old1 = 0;
float Omega1_old2 = 0;
float Omega1 = 0;

// This function is called every 1 ms
void lab(float theta1motor,float theta2motor,float theta3motor,float *tau1,float *tau2,float *tau3) {
    Omega1_raw = (theta1motor - Theta1_old)/0.001;
    Omega1 = (Omega1_raw + Omega1_old1 + Omega1_old2)/3.0;
    Theta1_old = theta1motor;

    // order matters here. Why??
    Omega1_old2 = Omega1_old1;
    Omega1_old1 = Omega1_raw;
}
```

**Second Method of Filtering Velocity**

```c
float Theta1_old = 0;
float Omega1_old1 = 0;
float Omega1_old2 = 0;
float Omega1 = 0;
float Omega1_raw = 0;

// This function is called every 1 ms
void lab(float theta1motor,float theta2motor,float theta3motor,float *tau1,float *tau2,float *tau3) {
    Omega1_raw = (theta1motor - Theta1_old)/0.001;
    Omega1 = (Omega1_raw + Omega1_old1 + Omega1_old2)/3.0;
    Theta1_old = theta1motor;

    // order matters here. Why??
    Omega1_old2 = Omega1_old1;
    Omega1_old1 = Omega1;
}
```
**Part 2: PD and PID Control of the CRS Robot.**

**Part 2.1 Implementation of PD Control for the three joints of the CRS Robot.**

Now in Code Composer Studio starting from where you left off in Lab 1, implement PD control laws for the three joints of the CRS robot. First using Windows 10’s File Explorer, copy your Lab.c file to Lab2.c file. Then in Code Composer Studio’s Project Explorer, right click on your CRSrobot project and choose “Add Files” and then “Link to Files” to link Lab2.c to your CCS project. You will also have to right click on Lab.c and “Exclude from Build.”

Implement

\[
\tau_1 = K_{P1} \cdot (\theta_{1d} - \theta_1) - K_{D1} \cdot \dot{\theta}_1
\]

\[
\tau_2 = K_{P2} \cdot (\theta_{2d} - \theta_2) - K_{D2} \cdot \dot{\theta}_2
\]

\[
\tau_3 = K_{P3} \cdot (\theta_{3d} - \theta_3) - K_{D3} \cdot \dot{\theta}_3
\]

where \(\theta_{1d}, \theta_{2d}\) and \(\theta_{3d}\) are step inputs from 0 to \(\pi/6\) radians from \(t = 0\)s to \(t = 2\)s, followed by a step back to 0 from \(t = 2\) to \(t = 4\). Then continue to repeat this step input.

Saturate these torque commands at +/- 5.

Start out with \(K_{P1}, K_{P2}\) and \(K_{P3}\) equal to 5 and \(K_{D1}, K_{D2}\) and \(K_{D3}\) equal to 0.2. Using Simulink to plot your joint step responses, tune these six gains. To plot in Simulink, run the given Simulink file, simulink5ms_plotAndGains.slx and remember to connect the serial port to the cable labeled “Simulink.” Tune your three \(K_p\) gains and your three \(K_d\) gains inside Code Composer Studio’s Watch Expression while the robot is running. Your goal for each joint is to achieve a rise time less than 300 ms, percent overshoot less than 1% and minimal steady state error. Produce plots of your step responses along with the torque applied to the system. Also produce plots of tracking errors,

\[
e_1 = \theta_{1d} - \theta_1, \quad e_2 = \theta_{2d} - \theta_2, \quad e_3 = \theta_{3d} - \theta_3
\]

**NOTE: IMPORTANT!!** To get started with this PD implementation, I highly recommend you start out with just \(\theta_{1d}\), the waist joint. Implement the PD control law for just \(\theta_{1d}\). To calculate the \(\theta_{1d}\) step input, described above, take advantage of the 32bit (long) integer “mycount” that is being incremented by 1 for you each time your “lab” function is called, which is every 1ms. Use an if statement to check if

\[
((\text{mycount}\%2000)==0).
\]

If that is true, you know that 2 seconds have elapsed, so step (change) \(\theta_{1d}\) to \(\pi/6\) radians if \(\theta_{1d}\) is currently at 0 radians or step to 0 radians if \(\theta_{1d}\) is currently at \(\pi/6\) radians. Start “float Kp1” at 5 and “float Kd1” at 0.2 and implement the PD controller for the waist joint:

\[
*\tau_1 = K_{P1} \cdot (\theta_{1d} - \theta_1) - K_{D1} \cdot \dot{\theta}_1
\]

Once the PD control is working for \(\theta_{1d}\) implement the PD control for \(\theta_{2d}\) and \(\theta_{3d}\).
2.2 How to implement an integral approximation.

There are a number of different integration rules that can be used to estimate an integral. For this exercise we will use the trapezoidal approximation to implement the integral portion of the PID controller. In the PID controller the error signal is integrated. Looking at the below figure you should be able to see how the trapezoidal rule at each new time step $T$ sums up the new trapezoid sliver of area under the curve giving the equation $I_K = I_{K-1} + \frac{e_K + e_{K-1}}{2} \times T$ to approximate the integral. The integral normally does not need filtering because its output is naturally smoother than then its input data.

\[
I_K = I_{K-1} + \frac{e_K + e_{K-1}}{2} \times T
\]

Part 2.3 Add Integral Control.

Now add an integral term to your three PD control loops. Integration is a summing operator and must be monitored otherwise it could sum up to very large values. This problem is called integral windup.

You found above that the PD control, once tuned, did a very good job in controlling the robot’s links. Looking close at your step response plots though, you probably did see some steady state error. We are going to add integral term to our controller to attempt to improve this steady state error.

Not only does integration have the problem of integral windup, it also has issues of creating large overshoots in position control systems. So the way we are going to implement integral control in this lab is to only turn on the integral term when the angular velocity of the joint is small. How small is a tuning parameter. The joints are moving slow when they are getting close to steady state error equal to zero. So by
waiting for joint angular velocity to be small to start the integral control, the integral only operates when close to small error.

So add to your PD controllers the integral term $K_I I_K$, where $I_K$ is the integral of the tracking error and calculated using the trapezoidal integration rule. Add this integral term to your PD control only if the joint angular velocity is less than some threshold. **And very important**, if the joint angular velocity is greater than the threshold, make sure to zero the integral $I_K$ and any previous $I_{K-1}$.

To check for integral windup a very simple way is to just monitor the torque command to the joint’s motor. If the absolute value of the torque command to the motor is greater than the maximum of 5, then do not integrate further and leave the integral the value it had the previous sample. Tune your PID controller’s $K_P$, $K_D$, $K_I$ and integral switch point to achieve the desired response. Create plots of your step responses and tracking error and determine if integral control improved the system response.

**Part 3: PID Plus Feedforward Control.**

**Part 3.1 Implementation of Feedforward PD Controller.**

1. Write a Matlab script file that finds two sets of cubic polynomial coefficients in order to generate a cubic polynomial trajectory that:
   a. Starts at zero radians and goes to 0.5 radians in one second.
   b. Holds the joint at 0.5 radians for an additional second.
   c. Moves back from 0.5 radians to 0 radians again in one second.
   d. Holds the joint at 0 radians for another second.
   e. Repeat trajectory starting at step a.

   We will use this same trajectory for all three joints of the CRS robot. This trajectory should therefore satisfy

   \[
   \theta^d(0) = 0 \quad \hat{\theta}^d(0) = 0 \\
   \theta^d(1) = 0.5 \quad \hat{\theta}^d(1) = 0 \\
   \theta^d(2) = 0.5 \quad \hat{\theta}^d(2) = 0 \\
   \theta^d(3) = 0 \quad \hat{\theta}^d(3) = 0 \\
   \theta^d(4) = 0 \quad \hat{\theta}^d(4) = 0
   \]

   Section 5.5.1 in “Robot Modeling and Control” and section 8.2.1 of “Robot Dynamics and Control” second edition will help you find these coefficients. Make sure, when finding these coefficients, to keep eight digits of precision as numerical issues can occur when implementing this (or other) trajectories. When the trajectory is holding the joints at either 0.5 radians or 0 radians,
your code should not be calculating the cubic trajectory equations, just holding at the constant value.

2. Using the coefficients you just found, write a Matlab script that takes a parameter \( t \) seconds. Given \( t \), this function should return \( \theta^d(t), \dot{\theta}^d(t), \ddot{\theta}^d(t) \). With this script, generate plots of the desired theta and its first and second derivatives. Note that you will be writing a similar function in C to generate the desired trajectory.

3. Implement on the CRS robot a PID plus feedforward control to have the joints follow the desired cubic polynomial trajectory. See Section 6.4 of “Robot Modeling and Control” or Section 10.4 for “Robot Dynamics and Control” second edition. Write a C function, that given \( t \), returns that point along the polynomial trajectory just like the script you created above. Then using that trajectory implement the below control law. Ignoring friction, each joint’s control law will have the form

\[
\tau = l\ddot{\theta}^d + K_P (\theta^d - \theta) + K_I \int (\theta^d - \theta) + K_D (\dot{\theta}^d - \dot{\theta})
\]

Use \( l_1 = 0.0167, l_2 = 0.03 \) and \( l_3 = 0.0128 \). This is identical to the controller you implemented in 2.3 above except that the desired trajectory has been added. Also note that the sign in front of \( K_D \) is now positive. Explain why it makes sense that with step input trajectories as used in sections 2 and 3 the KD term is appropriate to be \(-K_D \dot{\theta}\) but now that we are generating polynomial trajectories the derivative term is \(+K_D (\dot{\theta}^d - \dot{\theta})\). Explain this to your TA. Also remember to implement the integral in the same fashion as before only applying integral control when the error is small and zeroing the integral when error is large. Because we are using polynomial trajectories the error should remain small as the joint moves along the trajectory. For this reason integral control will be active more of the time and may need some retuning along with the “small” error threshold for the integral. The \( K_P \) and \( K_D \) gains may also have to be re-tuned to meet specifications.

---


Part 4.1 Implementation of Gravity Compensation Feedforward PD Controller.

For this exercise we will see what we can do if a known mass is added (or picked up) at the end effector of the robot arm. If the mass of the object is known, we can add an additional feedforward term to cancel the weight of the added mass. We will not use integral control here so we can better see the effects of this gravity compensation. Remembering that using the Jacobian, joint torques equate to
\[
\begin{bmatrix}
\tau_1 \\
\tau_2 \\
\tau_3 \\
\end{bmatrix}
= J^T \begin{bmatrix}
F_x \\
F_y \\
F_z \\
\end{bmatrix}.
\]

We can add a feedforward term to our PD controller

\[
\tau = PD + J_e^T \begin{bmatrix}
0 \\
0 \\
mg
\end{bmatrix}
\]

In more detail using the PD plus feedforward controller

\[
\begin{bmatrix}
\tau_1 \\
\tau_2 \\
\tau_3 \\
\end{bmatrix}
= J_e^T \begin{bmatrix}
0 \\
0 \\
mg
\end{bmatrix} + \begin{bmatrix}
I_1 & 0 & 0 & 0 & \dot{\theta}_1^d \\
0 & I_2 & 0 & 0 & \dot{\theta}_2^d \\
0 & 0 & I_3 & 0 & \dot{\theta}_3^d \\
\end{bmatrix}
+ \begin{bmatrix}
K_{P1} & 0 & 0 & 0 & (\theta^d_{M1} - \theta_{M1}) \\
0 & K_{P2} & 0 & 0 & (\theta^d_{M2} - \theta_{M2}) \\
0 & 0 & K_{P3} & 0 & (\theta^d_{M3} - \theta_{M3}) \\
\end{bmatrix}
\]

Steps for this exercise:

1. First we need a controller to use to compare using gravity compensation and not using gravity compensation. We will use your PID plus feedforward control designed in Part 3 but not include the integral terms making it a PD plus feedforward. (We don’t want the integral control rejecting the disturbance that occurs when adding this mass.) So go back to your Part 3 controller and zero your three Ki gains. Take a little time and tune this PD controller for small error response. Show your TA.

2. Now with this PD controller tuned, ask your TA to add a 4*39 Kg mass to the end-effector of your robot. Add the gravity compensation to your PD plus feedforward controller but also create a global float variable “usegravity” that you can use to turn on and off the gravity compensation. Have this variable multiply the gravity compensation terms in your controller. Then in the CCS expressions view, you can set this variable to 1.0 if you want to apply the gravity compensation and to 0.0 if you want to see what happens without gravity compensation.

3. Run your controller with the cubic trajectory and note the differences between using the gravity compensation and not using the compensation. Produce a plot of cubic responses with some having gravity compensation enabled and some with it disabled.

Part 5: Follow a X,Y,Z trajectory.

5.1 Follow a trajectory of your choice.
For this exercise come up with new trajectories for each joint that make the CRS robot arm repeatedly follow a fun trajectory. One idea would be to code your inverse kinematic equations that given an x,y,z point in space returns the three motor angles. With this code you could have the robot follow a straight line or a figure eight. As a safety check first code your trajectory equations in a Matlab M-file and with plots make sure the trajectory never takes the robot arm outside of its angle limits. Then code the same trajectory in C and see how well the robot can follow your trajectory.

**Part 6: Try your PD tuned controller on the HW Simulation**

6.1 Run the HW CRS Robot Simulation with your first PD controller.

For this exercise you will try your tuned PD controller on your HW simulation and see how well it compares to the real robot. One of my goals this semester, as I have time, is to see if I can adjust the parameters of the simulated CRS robot to make it better replicate the actual CRS robot. The goal being to better understand the real parameters of the CRS robot.

Your HW simulation uses, what we have been calling in lab, “DH thetas” instead of “motor thetas.” In addition, the simulation uses what we will call “DH torques” instead of the “motor torques.” The “DH torques” are torques applied at each joint, where the actual “motor torques”, motor three specifically, are located away from the joint using a chain linkage.

So to simulate the same controller you implemented in Part 2 in your simulation you will have to perform a conversion from “DH thetas” to “motor thetas” and then a conversion from “motor torques” to “DH torques.”

Remember from Lab 1 that you found that

$$\begin{bmatrix}
\theta_{1M} \\
\theta_{2M} \\
\theta_{3M}
\end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} \theta_{1DH} \\
\theta_{2DH} \\
\theta_{3DH} \end{bmatrix} + \begin{bmatrix} 0 \\ \pi/2 \\ 0 \end{bmatrix} \quad (1)$$

So

$$\begin{bmatrix}
\dot{\theta}_{1M} \\
\dot{\theta}_{2M} \\
\dot{\theta}_{3M}
\end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} \dot{\theta}_{1DH} \\
\dot{\theta}_{2DH} \\
\dot{\theta}_{3DH} \end{bmatrix} \quad (2)$$

To find the torque relationship we equate the power in both coordinates

$$\dot{\theta}_{DH}^T \tau_{DH} = \dot{\theta}_{M}^T \tau_{M}$$

Substituting $\dot{\theta}_{M}^T$ with (1), the relation between “motor angular velocities” and “DH angular velocities”

$$\dot{\theta}_{DH}^T \tau_{DH} = \dot{\theta}_{DH}^T \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \tau_{M}$$
Giving the relationship between the torques

\[
\tau_{DH} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \tau_M \tag{3}
\]

Given the relationships (1), (2) and (3) modify your robot simulator’s fcn_controller.m function to implement your lab’s PD controller to run a single step response:

1. Remember that the fcn_controller.m gives you “q” which are “DH thetas”, “dq” which are “DH angular velocities” and returns “tau” which are “DH torques.”
2. Create a variable “q_desired_motor” to set the single step response, q_desired_motor = [\pi/6; \pi/6; \pi/6];
3. Create a variable “q_motor” and use equation (1) to convert “q” to “motor thetas.”
4. Create a variable “dq_motor” and use equation (2) to convert “dq” to “motor angular velocities.”
5. Create a variable “tau_motor” and use the Kp and Kd gains you found in Part 2 and implement your PD controller equations that use “motor” variables just like you did in Part 2.
6. Finally find “tau” by using equation (3) to solve for “DH torques.”

Produce a plot of your simulated step response.

Report: (Minimal Requirements)

1. Include the final version of your C code that includes all of your designed controllers for this lab. Of course the controller that is not being used will be commented out. Make clear with comments which section of code is for which section of the lab.
2. Include any Matlab M-files you created
3. Answer the questions found in the lab.
4. Explain how you generated your last “fun” trajectory.
5. Did you notice any performance differences between the different control methods?
6. Include your fcn_controller.m M-file you edited to implement the PD controller for your simulation. Also include a plot of angles, angular velocities and torques of your simulation run.

Appendix