Control and Robotics Design

White-Paper **Updated - April 14, 2021**

ELEC 391 Team MotoGuzzi - Jeff Eom, Mark Gnocato, Yanyu (Grace) Zhang, ECE, University of BC

Abstract

A 3-DOF planar joint robot arm is developed to discard 3 faulty products on a stopped conveyor. In this report, control and robotic parts of the project is focused. Section 1. Motors and Controllers describes the detailed usage of each motors and their controllers. Section 2. Robotics describes the robotics on direct, inverse kinematics and the path planning. Section 3. Controllers describes the detailed information of the controllers that is implemented in the system. Lastly, section 4. Co-simulation describes how it is simulated using Simulink and Simulation X simulation program.

Nomenclature

A1 Length of 1st arm (m)
A2 Length of 2nd arm (m)
Q1 Angle of 1st arm (Degree)
Q2 Angle of 2nd arm (Degree)
Q3 Angle of gripper (Degree)
Q4 Angle to close and open gripper (Degree)

1. Motors and Controllers

The robot arm consists of four motors: one motor mounted at the base of the arm, two motors mounted at two joints of the arm and lastly one motor mounted at the gripper. Each of the motors are controlled individually according to the angle that is calculated from the inverse kinematics of the path plan.

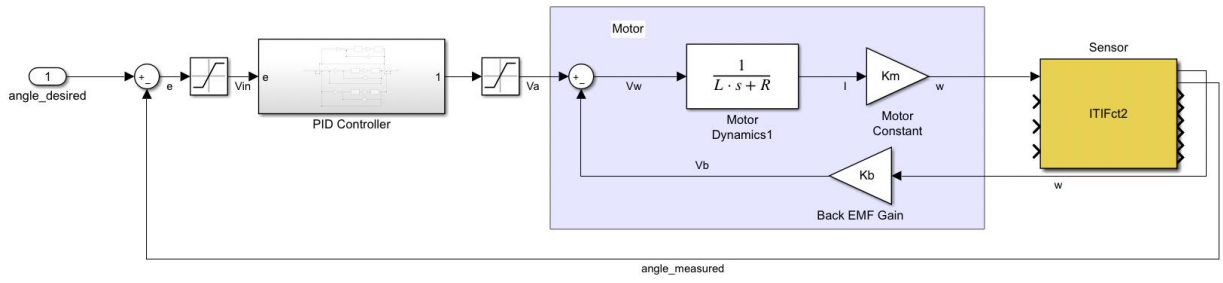

Figure 1. Linear model

The linear model of the control system is shown in Figure 1, the controller has an input of desired angle and uses PID controller then feeds the correct amount of voltage to the motor. The angle measured from the sensor is then used to calculate error which goes back to the input of the controller to minimize its error by subtracting with the desire value. Each motor uses the above controller, so the full system uses four of them in total shown in Figure 2.

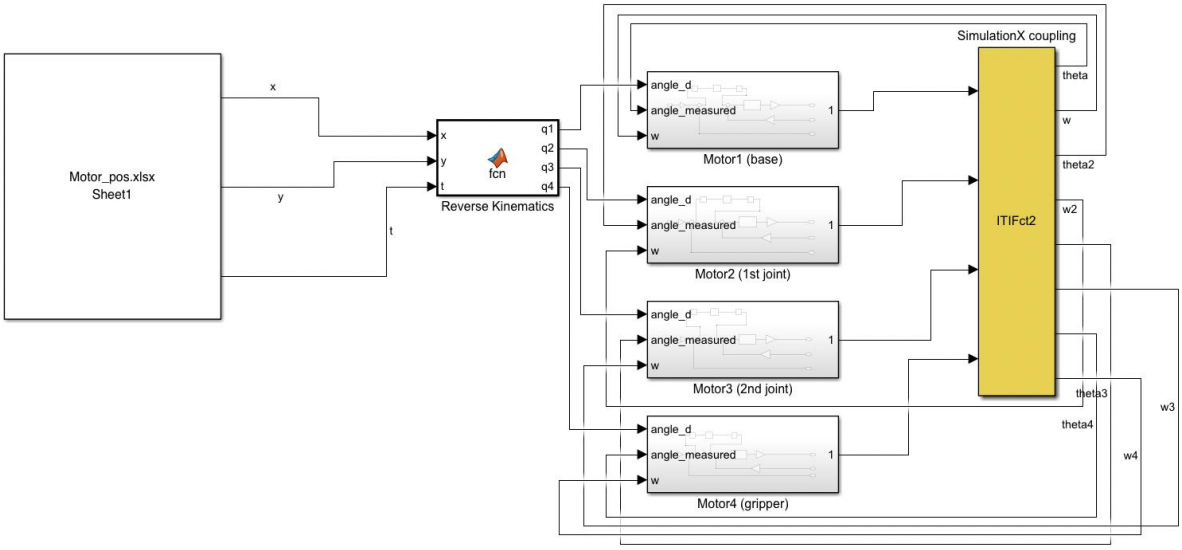

Figure 2. Linear model for all motors

2. Robotics

The path planning of the arm is designed to use excel data sheet to reference the location as shown in Figure 2. Excel sheet contains the position data and the time. Position data contains where faulty products are located and where to dispose the faulty product. The time data is used so the arm will know when and what to do at a certain time.

7.5000	−10	40
7.6000	−10	40
7.7000	−10	40
7.8000	−10	40
7.9000	−10	40
8	−10	40
8.1000	−10	30
8.2000	−10	30
8.3000	−10	30
8.4000	−10	30
8.5000	−10	30

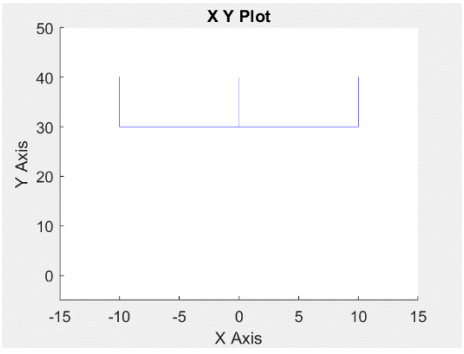

Figure 3. Excel sheet of the path and the path of the arm during the process

In Figure 3. Excel sheet of the path and the path of the arm during the process, excel data sheet shows the position data during 7.5 seconds to 8.5 seconds. The first column represents the elapsed time, second column is the desired x-position in cm, and the last column is the desired y-position in cm. At 7.5 seconds, the arm is moving from where marshmallow is at position (-10,40) to the garbage chute that is at (-10,30).

The visual of the path is shown with blue lines in Figure 3, the image on the right. The detailed path plan is as follows in Figure 4.

0-1 sec to stand still at pos (0, 20)

1-3 sec to move to pos (-10, 30)
3-5 sec to move to first marshmallow (-10, 40)
5-9 sec to close the grip
7-9 sec to move to garbage chute (-10, 30)
9-11 sec to open the grip

11-13 to move to pos (0, 30)
13-15 sec to move to second marshmallow (0, 40)
15-19 sec to close the grip
17-19 sec to move to garbage chute (0,30)
19-21 sec to open the grip

21-23 sec to move to pos (10, 30)
23-25 sec to move to last marshmallow (10, 40)
25-29 sec to close the grip
27-29 sec to move to garbage chute (10,30)
29-31 sec to open the grip

31-33 sec to come back to the original pos (0, 20)

Figure 4. Path in detail

The info above written in the excel sheet is read from the user defined MATLAB function that uses the inverse kinematics to calculate how much the motor must turn. The inverse kinematics uses trigonometric functions to measure their desired angles. The equations used for the inverse kinematics are shown in Figure 5. and the code that does the inverse kinematics is in Figure 6.

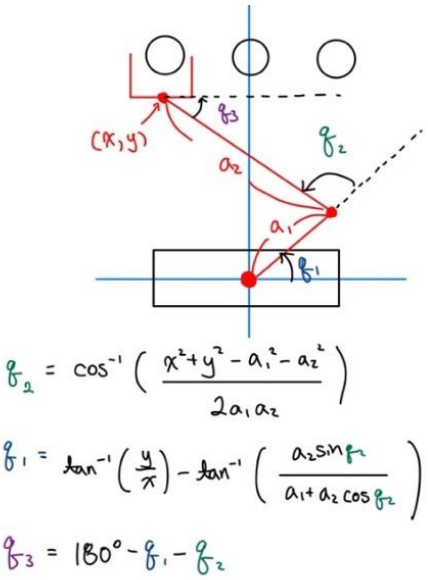

Figure 5. Inverse kinematics calculation [1]

```
function [q1, q2, q3, q4]= fcn(x,y,t)
    a1 = 20;
    a2 = 30;

    q2 = acosd((x^2+y^2-a1^2-a2^2)/(2*a1*a2));
    q1 = atand(y/x)-atand(a2*sind(q2)/(a1+a2*cosd(q2)));
    q4 = 0;
    if q1<0
        q1 = 180+q1;
    end
    q3 = 180-q1-q2;
    if q3<0
        q3 = 90;
    end

    if x==-10 && y==40 && t>=6 && t<=8
        % close gripper
        q4 = 180;
    end
    if x==-10 && y==30 && t>=10 && t<=12
        % open gripper
        q4 = 0;
    end

    if x==0 && y==40 && t>=16 && t<=18
        % close gripper
        q4 = 180;
    end
    if x==0 && y==30 && t>=20 && t<=22
        % open gripper
        q4 = 0;
    end

    if x==10 && y==40 && t>=26 && t<=28
        % close gripper
        q4 = 180;
    end
    if x==10 && y==30 && t>=30 && t<=32
        % open gripper
        q4 = 0;
    end
```

Figure 6. Inverse kinematics code

3. Controllers

All robot arm joints have the controllers that was discussed in section 1. Motors and Controllers. Each controller uses PID controller to reach the correct voltage in a right amount of time with 0 overshoot to not have any interfere with other marshmallows while traveling. In the project, PID controller was written in 3 different ways. In depth PID tuning for each joint is shown in Appendix B, along with the digital filter weighted sum calculation.

3.1 PID controller written in C

PID controller written in C is shown below in Figure 7. This code creates the PIDController struct and functions on the bottom calculates and initiates the PIDController.

```c
struct PIDController
{
    float Kp;
    float Ki;
    float Kd;

    float tau;
    float t;

    float integrator;
    float differentiator;
    float prevError;
    float prevmeasure;

    float output;
};

void PIDController_Init(PIDController *pid)
{
    (*pid).integrator = 0.0f;
    (*pid).prevError = 0.0f;
    (*pid).differentiator = 0.0f;
    (*pid).prevmeasure = 0.0f;
    (*pid).out = 0.0f;
}

float PIDControl(PIDController *pid, float desireAngle, float measuredAngle)
{
    float error = desireAngle - measuredAngle;
    float gain = (*pid).Kp * error;

    (*pid).integrator = (*pid).integrator + 0.5f * (*pid).Ki * (error + (*pid).prevError);

    (*pid).differentiator = -(2.0f * (*pid).Kd * (measuredAngle - (*pid).prevmeasuredAngle)
                            + (2.0f * (*pid).tau - (*pid).t) * (*pid).differentiator)
                            / (2.0f * (*pid).tau + (*pid).t);

    (*pid).out = gain + (*pid).integrator + (*pid).differentiator;

    (*pid).prevError = error;
    (*pid).prevmeasuredAngle = measuredAngle;

    return (*pid).out;
}
```

Figure 7. PID controller written in C. [2]

3.2 PID controller in MATLAB

PID controller in MATLAB is shown in Figure 8. The controller that is built is tested to a random transfer function and the result plot is just the same as the PID block in the Simulink shown in Figure 8.

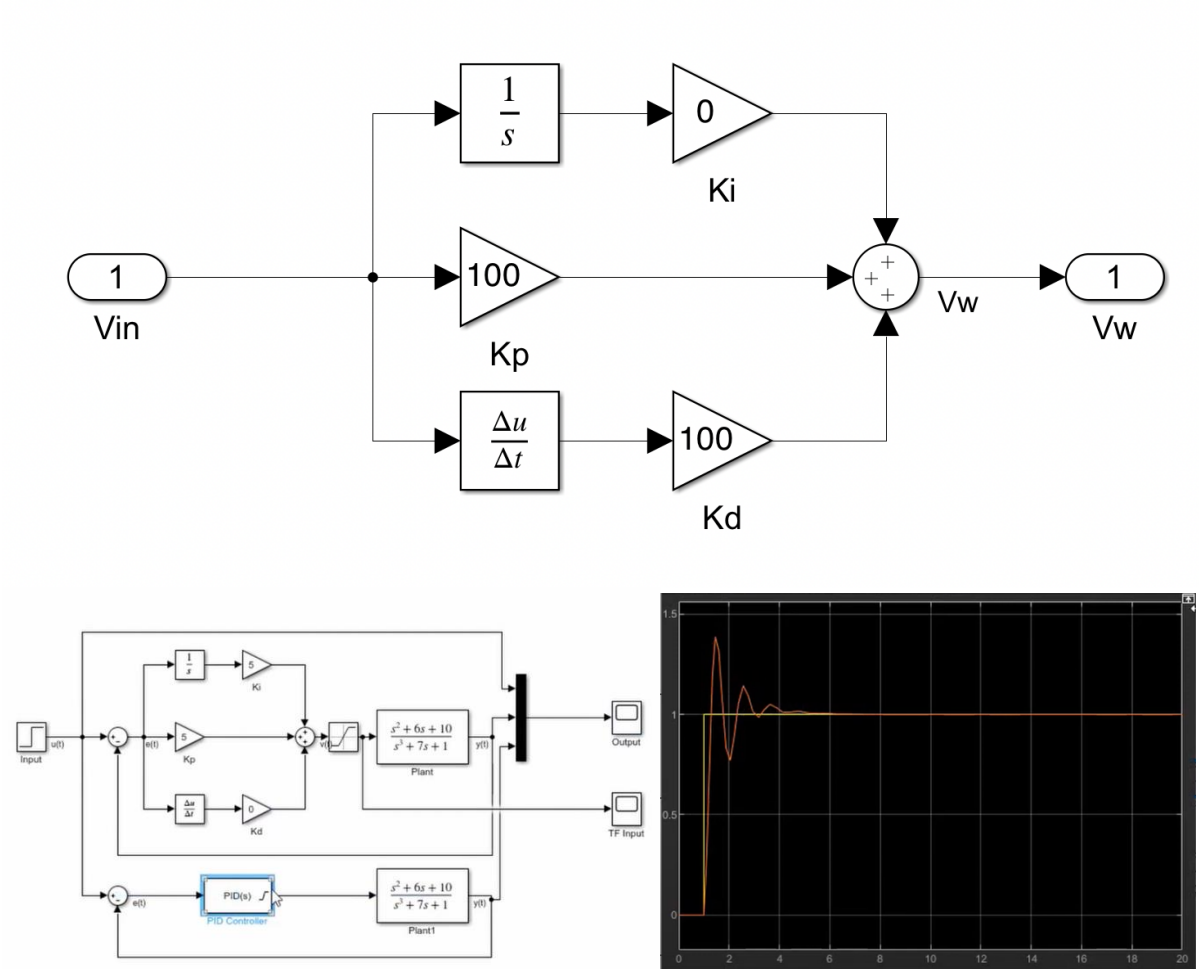

Figure 8. User-defined PID controller and its test [3]

3.3 PID controller in Arduino IDE

In order to estimate the ISR clock rate for the PID controller, Arduino code was written. The estimated ISR clock rate for the controller is measured to be 180 cycles each loop after the setup. The Arduino code is shown in Figure 9.

```
#define sensorPin 14
#define motorPin 7
#define desireAnglePin 8

float errorP, errorI, errorD;
float errorB;
float output;
// PID params
float Kp, Ki, Kd;
float pidVal;

void setup() {
  // baud rate = 9600;
  Serial.begin(9600);
}
void loop() {
  // get error between desire angle and the actual
  // cycle = subtract (18) + analogRead(5)*2 + assignment operator (4) = 32
  errorP = analogRead(desireAnglePin) - analogRead(sensorPin);
  // add up the error I control
  // cycle = add (18) + assignment operator (4) = 22
  errorI += errorP;
  // minus the error D control
  // cycle = subtract (18) + assignment operator (4) = 22
  errorD = errorB-errorP;
  // write previous error for future reference.
  // cycle = assignment operator (4) = 4
  errorB = errorP;
  // cycle = add (18)*2 + multiply(18)*3 + assignment operator (4) = 94
  pidVal=errorP*Kp+errorI*Ki+errorD*Kd;
  // write the voltage to the output pin.
  // cycle = analogWrite(6) = 6
  analogWrite(motorPin,pidVal);
}
```

Figure 9. PID controller written in Arduino IDE [3]

4. Co-simulation

Using the TCP/IP shown in Figure 2, Simulink is connected with SimulationX so that the sensors from the SimulationX deliver value of the motors' angles and their angular speeds to apply the PID control in the Simulink. During the co-simulation, re-tune of the controllers was done to produce near to 0% overshoot as shown in Figure 10.

The tuning technique that was used during the re-tune process of the co-simulation was to increase the P value first by doubling its number until the arm moves in a uniform oscillation. Then increase the D value by doubling until it reaches to almost 0% overshoot and increase, I value at the end to fine-tune until the overshoot is at 0% during each motion.

In SimulationX, all the designs from the Solidworks are transferred so that it simulates like in a real environment. The design of the arm is shown in Figure 11 and it shows that the grip successfully grabs one of the marshmallows. The entire diagram view is shown in Figure 13.

During the design process, the gripper of the final product was meant to be a better and efficient looking gripper than what is shown in Figure 11. The initially designed gripper is shown in Figure 12. Our team failed to transfer the product from the Solidworks to the SimulationX

correctly in time, so a simpler version of the gripper was used instead. If there were more time given for the project, this is the one of the things that must have been prioritized first to be improved in the project.

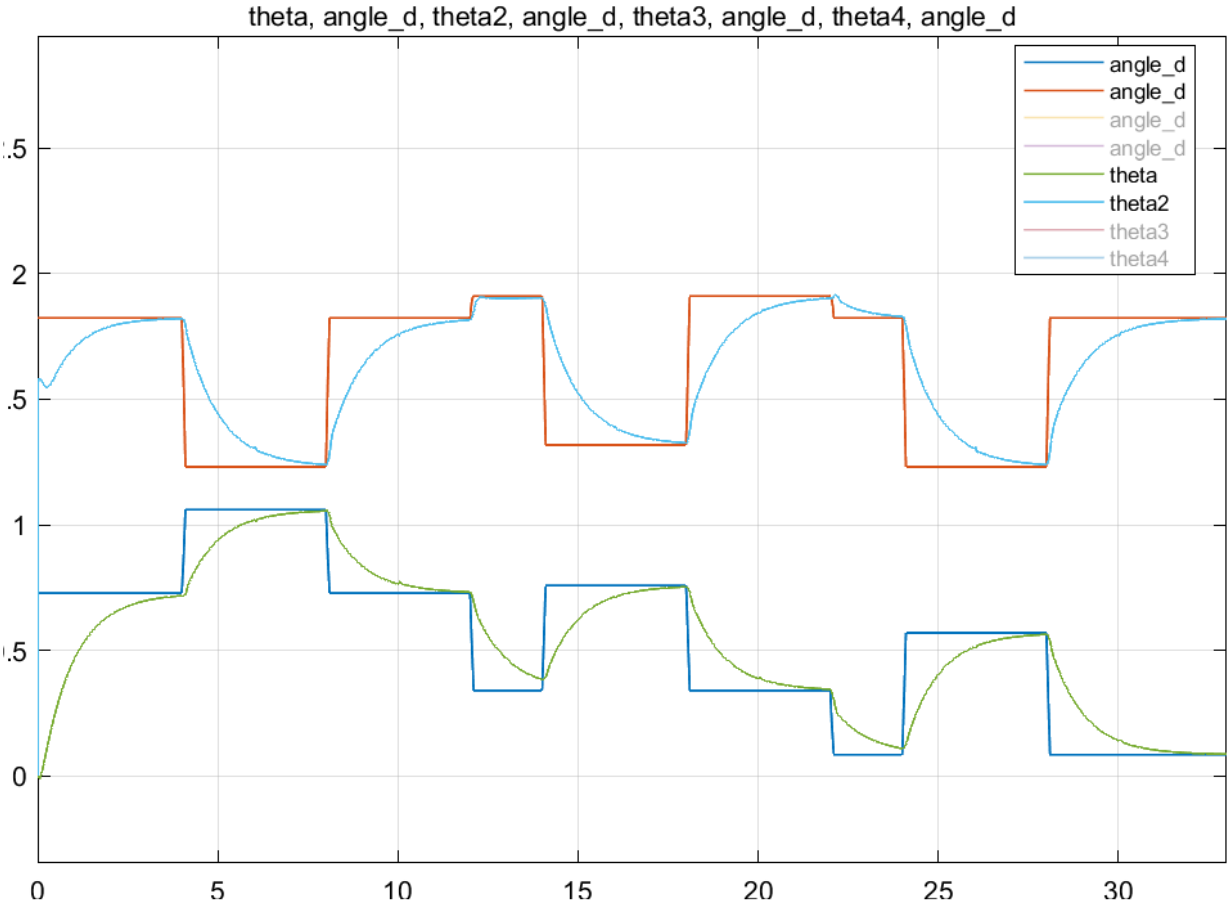

Figure 10. SimulationX data of Q1 and Q2

Appendix A

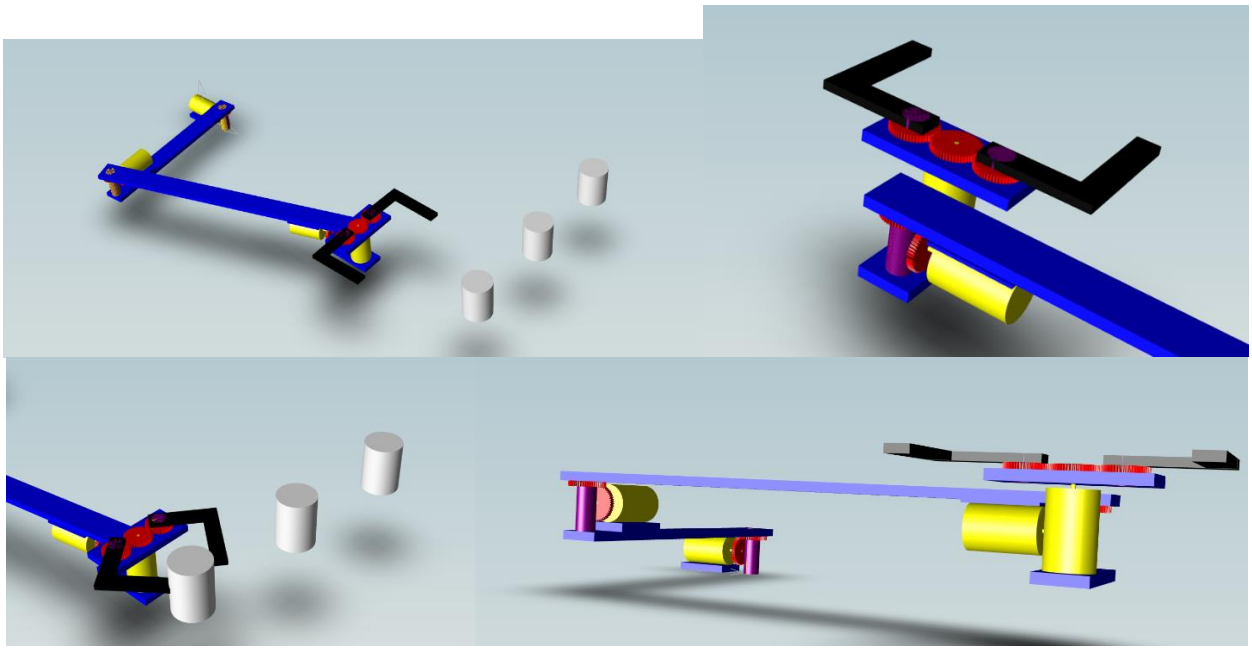

Figure 11. Robot arm in SimulationX

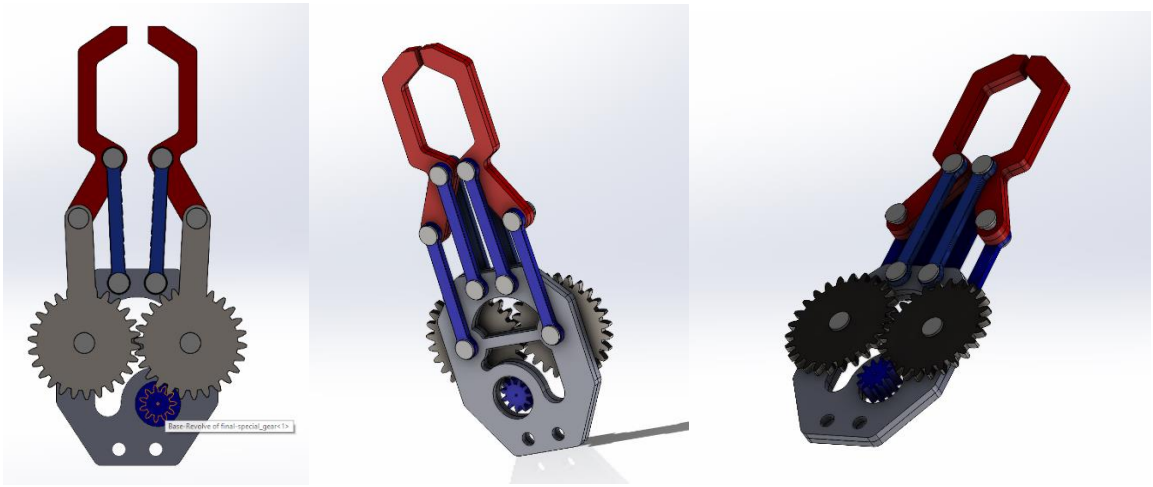

Figure 12. Gripper that was initially designed

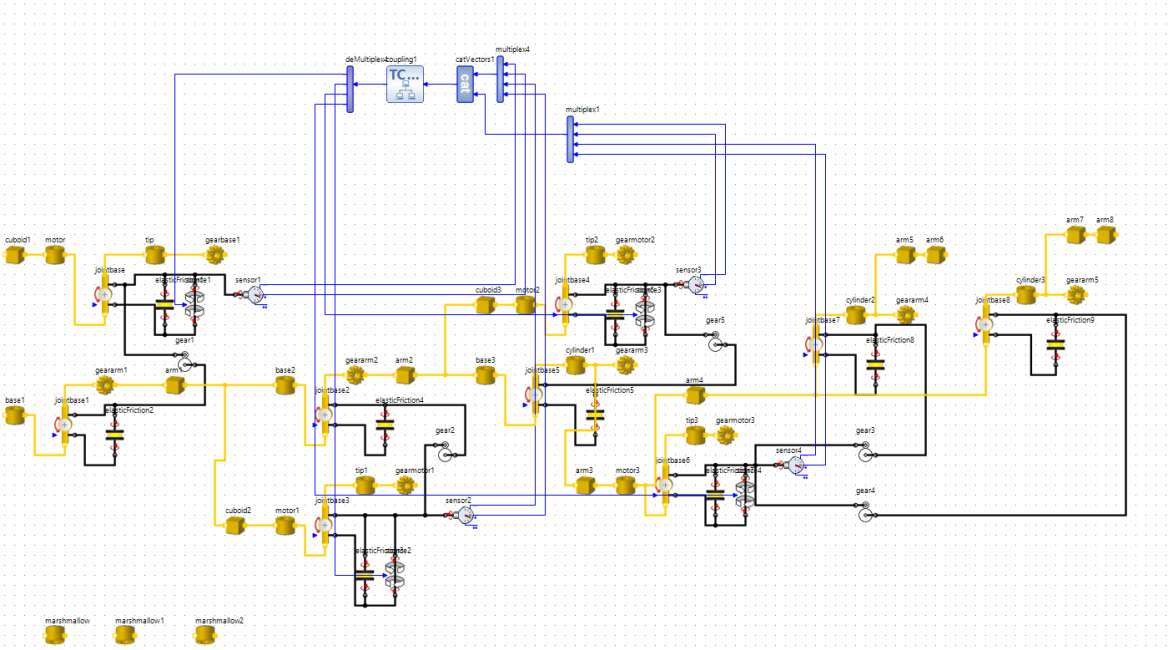

Figure 13. Full diagram built in SimulationX

Appendix B – PID Tuning Strategy

By following the 10-Step Process, different PID tuning strategies can be applied to different joints.

- The arm segment at the base of the robot will likely have a larger range of motion and therefore this motor should have a fast response.
- The wrist is responsible for centering the gripper and it must be accurate when approaching the marshmallow otherwise the gripper will not properly pick the object up. This requires a small steady state error.

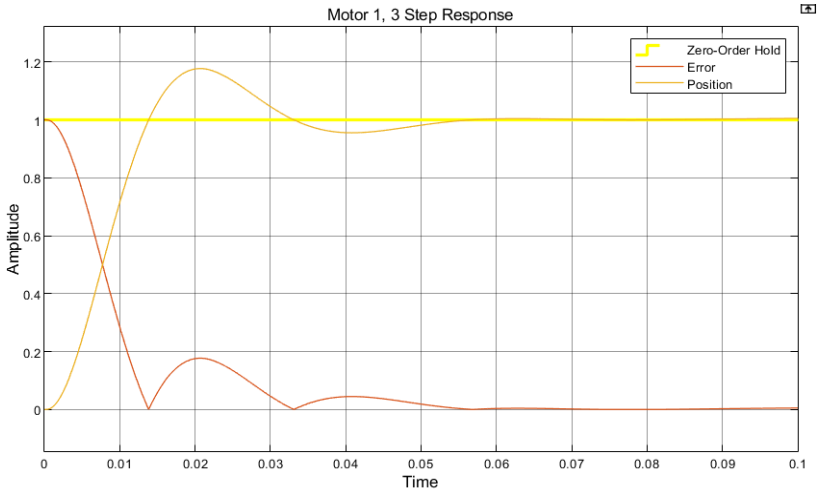

Figure B1 - Tuned Response for Motors 1 and 3

```
S = struct with fields:
        RiseTime: 9.1491e-003
    SettlingTime: 49.4723e-003
     SettlingMin: 927.0715e-003
     SettlingMax: 1.1787e+000
       Overshoot: 17.8705e+000
      Undershoot: 0.0000e+000
            Peak: 1.1787e+000
        PeakTime: 20.7526e-003
K =
   225.8887e+000
derivative_pole =
     3.2378e+003
integral_zero_value =
     6.2710e+000
integral_zero_angle =
    25.0000e+000
Ki =
     2.7435e+000
Kp =
   792.1653e-003
Kd =
    69.5206e-003
open_loop_tf =

  From input to output "y1":
                    458.4 s^3 + 5219 s^2 + 1.813e04 s + 364
  ---------------------------------------------------------------------------
  2.91e-10 s^7 + 5.707e-06 s^6 + 0.01629 s^5 + 2.798 s^4 + 39.43 s^3 + 12.09 s^2

Continuous-time transfer function.
```

Figure B2 - Tuning Parameters for Motor 1, 3

- The second arm segment will be moving constantly and within proximity of the assembly line parts and therefore overshoot should be minimized.
- The gripper should not crush the object and therefore should also minimize overshoot.

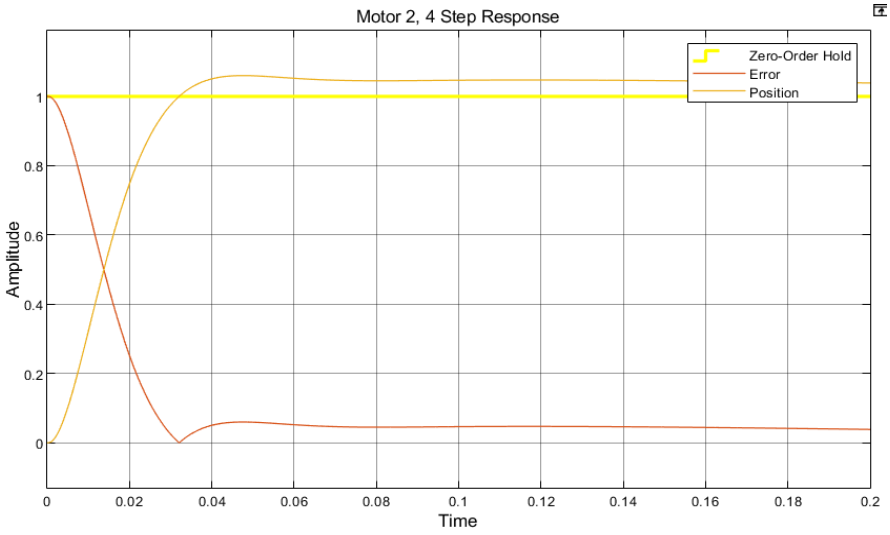

Figure B1 - Tuned Response for Motors 2 and 4

```
S = struct with fields:
        RiseTime: 20.6286e-003
    SettlingTime: 321.7451e-003
     SettlingMin: 908.9089e-003
     SettlingMax: 1.0608e+000
       Overshoot: 6.0810e+000
      Undershoot: 0.0000e+000
            Peak: 1.0608e+000
        PeakTime: 47.4682e-003
K =
   131.7684e+000
derivative_pole =
     4.8568e+003
integral_zero_value =
     8.7794e+000
integral_zero_angle =
    15.0000e+000
Ki =
     2.0912e+000
Kp =
   459.7219e-003
Kd =
    27.0363e-003
open_loop_tf =

  From input to output "y1":
                    267.4 s^3 + 4540 s^2 + 2.07e04 s + 416.2
  ---------------------------------------------------------------------------
  2.91e-10 s^7 + 6.178e-06 s^6 + 0.024 s^5 + 4.19 s^4 + 59.14 s^3 + 18.13 s^2

Continuous-time transfer function.
```

Figure B2 - Tuning Parameters for Motor 2, 4

MATLAB plots used during the 10-Step PID Tuning Process:

```
Number of poles in RHP of open-loop system: 0
Number of net encirclements around the -1 point:   0
=> Number of poles in RHP of closed-loop system:   0
and no closed-loop poles on Im-axis
=> Closed-loop-system is asymptotically stable
```

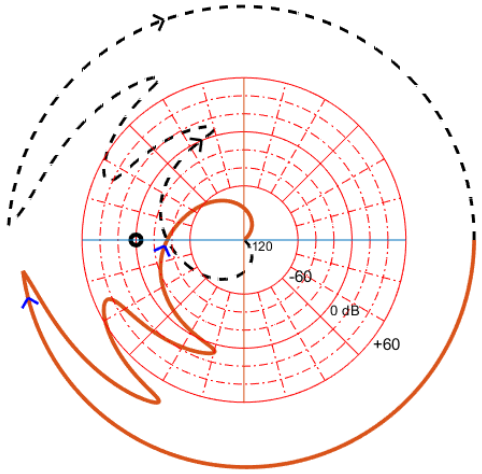

Figure N5 - Nyquist Plot

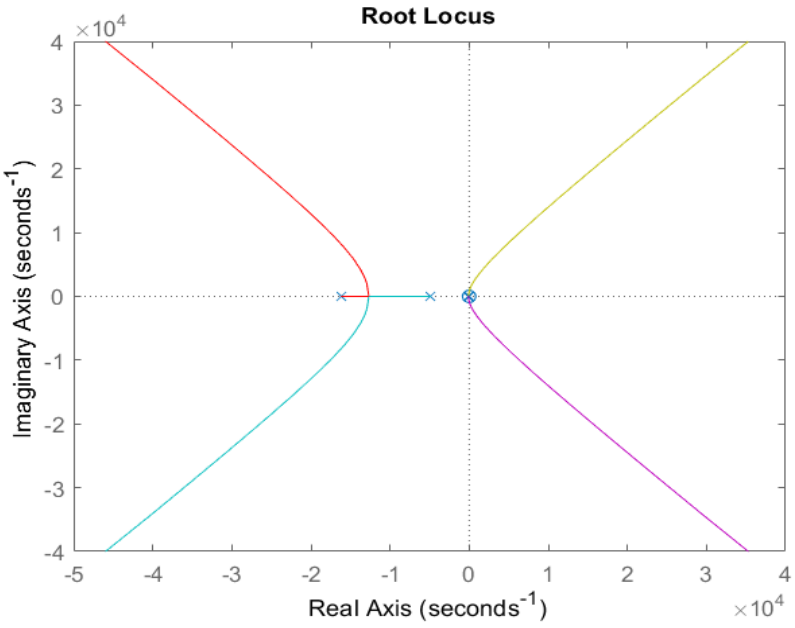

Figure B6 - Root Locus – Large Scale

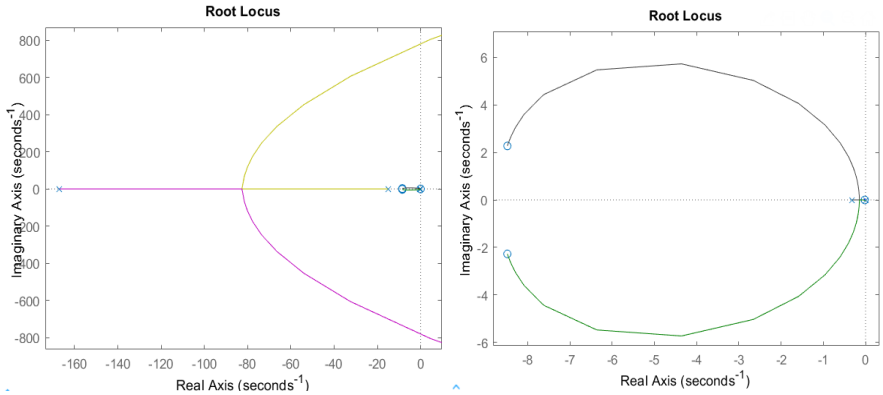

Figure B7 - Root Locus - Small Scale

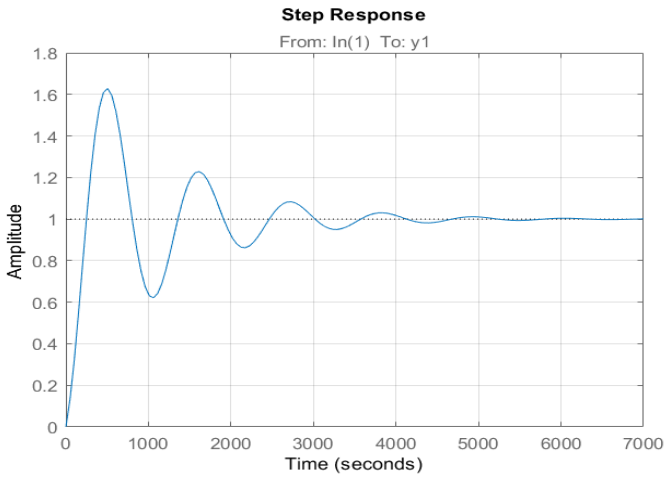

Figure B8 - PID Response Before Tuning

Digital Filter Weighted Exponentials

We assume a frequency of 10kHz and find the weighted exponential arrays. The area under each curve is equal to one.

```
CF = 10e3;   % default 1kHz
P1 = 3.2378e+003
P2 = 4.8568e+003
n_hat1 = floor(4*CF/P1) % ~ 12
n_hat2 = floor(4*CF/P2) % ~ 8
tmax1 = eval(solve(0.02 == exp(-P1*x), x))
tmax2 = eval(solve(0.02 == exp(-P2*x), x))
x1 = linspace(tmax1/n_hat1, tmax1, n_hat1)
y1 = 3.9060e3*exp(-P1*x1)
x2 = linspace(tmax1/n_hat2, tmax2, n_hat2)
y2 = 7.7534e3*exp(-P2*x2)
figure
plot(x1,y1)
hold on;
grid on;
bar(x1,y1, 0.025, 'black');
xlim([tmax1/n_hat1-tmax1/(n_hat1*2) tmax1+tmax1/(n_hat1*2)])
title("Weighted Sum Coefficients for Time Samples")
xlabel("Time (s)")
ylabel("exp(-pt)")
hold off;
figure
plot(x2,y2)
hold on;
grid on;
bar(x2,y2, 0.025, 'black');
xlim([tmax2/n_hat2-tmax2/(n_hat2*2) tmax2+tmax2/(n_hat2*2)])
title("Weighted Sum Coefficients for Time Samples")
xlabel("Time (s)")
ylabel("exp(-pt)")
arr1 = [];
for n = 1:n_hat1
    arr1 = [arr1, y1(n)*tmax1/n_hat1];
end
curve_area1 = sum(arr1)
weighted_coeffs1 = arr1
arr2 = [];
for n = 1:n_hat2
    arr2 = [arr2, y2(n)*tmax2/n_hat2];
end
curve_area2 = sum(arr2)
weighted_coeffs2 = arr2
```

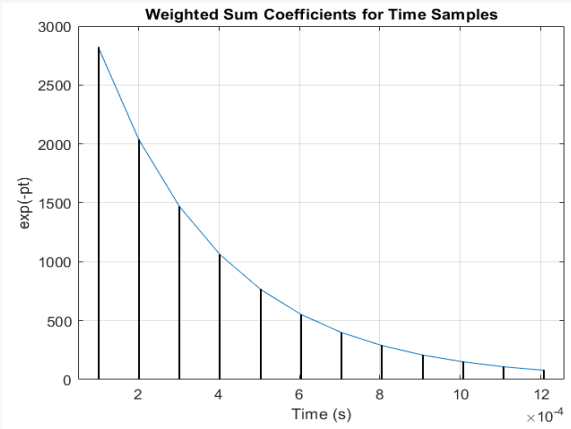

Figure B9 - Weighted Exponential Curve for PID Tune 1

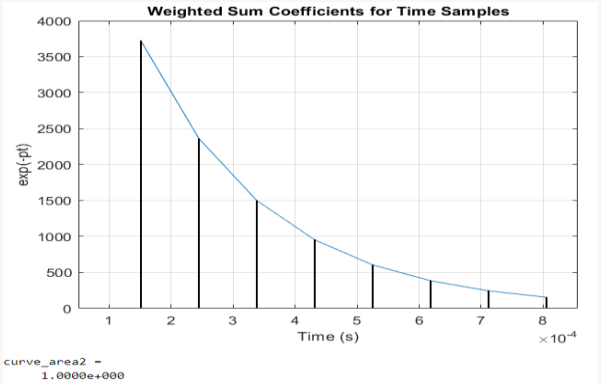

Figure B10 - Weighted Exponential Curve for PID Tune 2

References

[1] Inverse Kinematics for a 2-Joint Robot Arm Using Geometry. (2018, July 31). Robot Academy. https://robotacademy.net.au/lesson/inverse-kinematics-for-a-2-joint-robot-arm-using-geometry/

[2] Embedded PID Temperature Control, Part 3: Implementation and Visualization - Projects. (2016, April 4). All About Circuits. https://www.allaboutcircuits.com/projects/embedded-pid-temperature-control-part-3-implementation-and-visualization/

[3] Phil's Lab. (2020, May 22). PID Controller Implementation in Software. YouTube. https://www.youtube.com/watch?v=zOByx3Izf5U