**Robot Controller System Report**

Last Updated – April, 06, 2021

Group Big Gain, ELEC 391, University of BC, Vancouver, BC, Canada

## Abstract

An implementation of a PID controller is written in MATLAB and used in the Simulink model to control the SCARA robot. C-code is also written for the microcontroller (Arduino) and the ISR clock rate is estimated for determining the sampling time for the simulink model to intercept error signals. Both of these implementations utilize a filtered derivative that takes a weighted average of multiple error samples on each computation. The 10-step PID tuning guide provided on Canvas is used to establish an initial guess for the tuning parameters and the values are adjusted to minimise settling time and overshoot. The subsequent parts of the report determine the path plan and implementation to reach all marshmallows, inverse kinematics calculations to determine the angles required to reach desired coordinates, and direct kinematics to observe the error in position of the motors.

## Implementing the PID Controller (MATLAB Function)

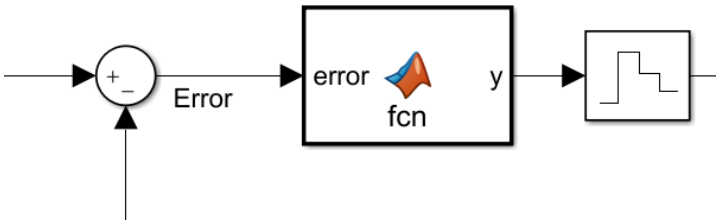



*Figure 1.1: Software implementation of PID Controller in MATLAB*

To set up the implementation of the PID Controller, we first determined the control frequency to be 10 kHz, which gives the PID controller a run time of 0.0001 seconds, which is a bit higher than the ISR time of the Arduino, as we will see later. This is to allow some extra time to make sure we are giving the Arduino code enough time to run and not interrupting it prematurely. This gives a control frequency of $\frac{1}{0.0001} = 10\ kHz$.

To implement a filtered derivative, we decide to average 8 samples on each iteration of the controller because this gives a very accurate response and overlaps exactly with the built-in PID block with a step input. The filter pole for the filtered derivative is chosen to be 5000 to average 8 samples because

$$\#\ of\ averaged\ samples = \frac{4\times\ Control\ Frequency}{Filter\ Pole}$$

A high filter pole will signify less delay in the response and it is also averaging 8 samples which is sufficient for high accuracy. With these decisions made beforehand, we can proceed with the code, a more detailed explanation of the filtered derivative will be given after displaying the code below:

```
function y = fcn(error)

persistent previous_error;                  %previous error = current error for
next iteration, initially set to 0;
persistent Ki_output;                        %need as persistent variable for
Ki_output = Ki_output + (...)
persistent ept;                             %constant vector of exponentials
(p*e^(-p*n*Ts)) where n = 1.....#of samples we want to average
persistent unweighted_differential_outputs;  %vector used for dot product with ept,
shifted at each cycle.

if isempty(previous_error)
       previous_error = 0;
end

%Trial values for testing against Built in PID block
Kp = 0.016;
Ki = 0.0000119;
Kd = 3.11914;

Ts = 0.0001; %Determined using the ISR clock rate of Arduino.

%--------P controller part-----------------
Kp_output = Kp*error;

%---------I controller part----------------

%Initializing Ki_output
if isempty(Ki_output)
       Ki_output = 0;
end

Ki_output = Ki_output + Ki*[(error + previous_error)/2]*Ts; %cumulative areas of
trapeziums

%----------D controller part---------------
controlfreq = 10000; %Hz
filter_pole = 5000;  %big pole because we don't care much about averaging but we do
care a lot about delay. (signal is not noisy)
n = 4*controlfreq/filter_pole;  %We are averaging 8 samples in this example
```

```
%Now, we initialize the ept vector (will remain constant valued throughout the
program)
if isempty(ept)
       ept = zeros(1,n);
       for i = 1:n
       ept(i) = filter_pole*1.132*exp(-filter_pole*i*Ts);
       end
       %We want Ts*sum(ept) = 1 but this is not the case normally, so we have to
scale the ept such that we get a sum
       %of 1. To account for discretization error, we scale it by multiplying it by
1.132 in this example.
end

if isempty(unweighted_differential_outputs)
    unweighted_differential_outputs = zeros(1,n);

unweighted_differential_outputs(1) = (error - previous_error);
filtered_derivative = dot(ept,unweighted_differential_outputs);

Kd_output = Kd*filtered_derivative;

n2 = n+1; %n2 = 9
for count = 1:(n-1) %1:7 in this example
unweighted_differential_outputs(n2-count) = unweighted_differential_outputs(n2 -
count - 1);
%shift values by one index, and in next loop calculate one more new slope at index
1 (starting index)
end

y = Kp_output + Ki_output + Kd_output;

previous_error = error; %this will help us move on to get new values for Ki and Kd

end
```

The values that we want to retain between the times the controller function is called are assigned as persistent, or global variables in MATLAB. The proportional component scales the error and the integral component is a cumulative sum of all the previous errors up to the present one.

## Filtered Derivative

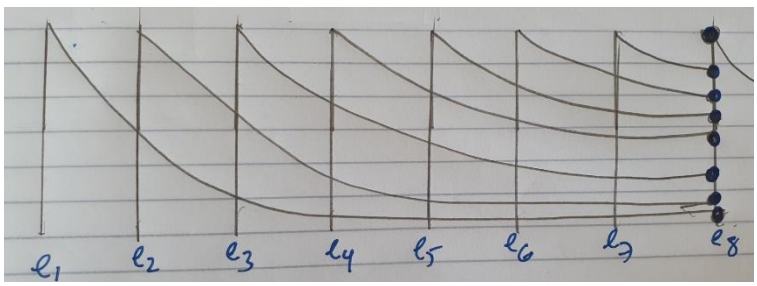


*Figure 1.2: Averaging 8 samples for the derivative component accounts for the exponential decay contributions from the previous errors*

This weighting is accounted for in the *ept (exponential – pole (filter) – sampling time)* vector of 8 numbers in the MATLAB code.

$ept\ vector\ =\ p\ *\ e^{-p*n*Ts}$ where n = sample number from 1 to 8, p = filter pole (5000) and Ts = 0.0001s

Ideally the $\int_0^\infty pe^{-p*n*Ts}\ =\ 1$ with unlimited samples 'n' sampled continuously but we are only sampling 8 samples hence there is a discretization error which causes the sum to be 0.883, hence we scale the *ept* vector by 1/0.883 = 1.132 to make the sum of all 8 numbers exactly equal to 1.

We accumulate eight errors (error – previous error) in the *unweighted_differential_outputs* vector and do a dot product with the ept vector to output a **filtered** derivative that can then be used in the output. It is important to shift the errors in *unweighted_differential_outputs* to the right by 1 each time the PID controller function is called in order to introduce the **new** error to the first index so as to make sure we are doing the dot product with a new *nweighted_differential_outputs* vector each time the PID function is called. This is what the for loop at the end of the MATLAB code shown above does.

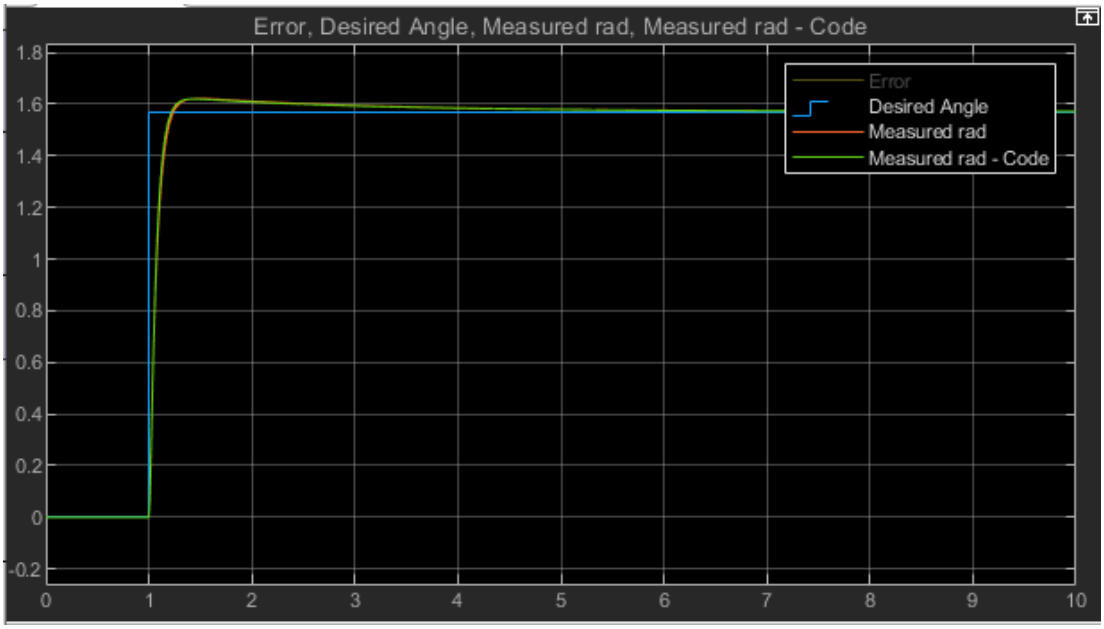


*Figure 1.3: MATLAB PID function can exactly track the response generated by the built-in PID block in Simulink with arbitrary tuning values*

**Arduino PID Controller (C-Code)**

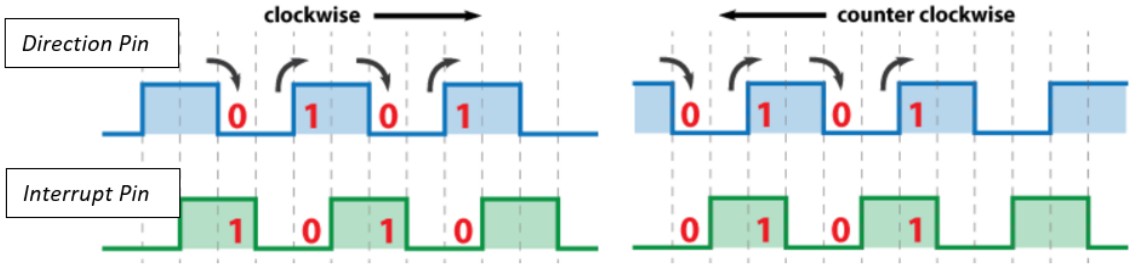



*Figure 1.4: Comparing the state of two pins to determine direction of angle*

For the microcontroller, we need to use the optical encoder to track the real-time change in angle of the motor – this was represented by the feedback in the Simulink model. The pulse() Interrupt service routine in the Arduino Code interrupts the main code every time the *interrupt_pin* rises and we check another pin that indicates when light passes through the slits of the encoder (whether it is high or low) – this is called the *direction_pin.* Whenever the Direction pin is high and the interrupt is high, we increment a counter which can then be converted to a positive angle (clockwise rotation) otherwise, if the direction pin is low when the interrupt pin is high (when the interrupt activates) counter is decremented and anticlockwise rotation. The counter can then be multiplied with the resolution of the arduino which basically indicates how many radians are traversed by each pulse i.e. 2pi/(Counts_per_turn). The encoders we are using have a CPR of 128.

```
#include <math.h>
#define interruptPin 2
#define directionPin 3
#define setValPin 5
#define PIDout 6
#define CPR 128 //(counts per revolution - review this number)

double prevError = 0; //GLOBAL variable, at setup it's set to 0 and it will
remember its previous value at every next function call.
double prevTime = 0;
double pulse_count = 0;
double sensorValue = 0; //angle in degrees
double resolution = (2*pi)/CPR;  //how many radians are traversed in one pulse
double Kp = 5; //pre determined, tuned values fed into microcontroller
double Ki = 5;
double Kd = 5;
```

```
double setpoint, input, output;
int filter_pole = 5000;
n = 8; //# of samples we want to average to get a filtered derivative
double ept[n];  //ept vector used for D in PID later to get a filtered derivative.
double unweighted_differential_outputs[n] = { 0 };  //initially assign all indices
to 0

void setup() {
  // put your setup code here, to run once:
  pinMode(setValPin, INPUT);  //desired angle
  pinMode(PIDout, OUTPUT);//output from the PID controller
  pinMode(directionPin, INPUT);
  pinMode(interruptPin, INPUT); //Jump to the ISR whenever this pin goes grom low
to high and increment/decrement counter based on ISR condition
  attachInterrupt(digitalPinToInterrupt(interruptPin), pulse, RISING); //Whenever
the pin goes from low to high, we check the status of directionPin to determine CW
or CCW rotation

  //ept vector: vector of exponential decays as we average n number of samples for
getting a *filtered* derivative -  will be used in the loop later.
  for (int i = 1; i <= n; i++) {
     ept(i) = filter_pole*1.132*exp(-filter_pole*i*Ts);
  }

  Serial.begin(9600);

}

void loop() {
  // put your main code here, to run repeatedly:
  setpoint = analogRead(setValPin);     //desired angle
  sensorValue = resolution*pulse_count;   //actual angle -> converted from current
pulse count to degrees based on optical encoder resolution
  double error = CalcErr(sensorValue, setpoint);  //Desired - actual
  output = RunPid(error, Kp, Kd, Ki);
  analogWrite(PIDout, output);          //output pin overwrites itself with PID
modifications to the error
}

//Interrupt: whenever the interruptPin rises i.e. goes high, ISR starts and we look
at the value of directionPin: if high, anticlockwise. Otherwise clockwise
void pulse(){
    int directionstate = digitalRead(directionPin);
    if (directionstate == 1){
      pulse_count++;
    }
    else{
```

```
      pulse_count--;
    }
} //pulse_count can be converted to an angle using the resolution which gives an
indication of sensorValue (actual angle)

double CalcErr(double setpoint, double sensorValue) {
  double error;
  return error = setpoint - sensorValue;
}

int RunPid(int error, int Kp, int Kd, int Ki) {
  double currTime = millis();     //time since the program started running in ms
  double delT = currTime - prevTime;  //initially prevTime set to 0 - in MATLAB we
set dT to a bit more than the ISR runtime to be conservative but here we are
calculating the time difference between consecutive error points for each
calculation.
  double delErr = error - prevError;  //initially prevError set to 0
  double filtered_derivative;

  double errorI  = errorI + ((error + prevError)/2)*delT;

  unweighted_differential_outputs[0] = (error - prevError);  //first index is the
differential, as the code runs again and again, we will shift the first index over
and the first index will get a new differential - so we keep shifting and updating
this vector
  filtered_derivative = dot(ept,unweighted_differential_outputs); //dot product
with the ept vector takes care of the averaging and gets us a filtered derivative.

//This for loop shifts each index one position to the left (index 0 value goes to
1, 1 goes 2, 2 goes to 3....)
  for count = 0:(n-2)
      unweighted_differential_outputs(n-count) = unweighted_differential_outputs(n
- count - 1);%shift values by one index, and in next loop calculate one more new
slope at index 1 (starting index)
  end

  Kd_output = Kd*filtered_derivative;

  prevError = error;  //for the next iteration, previous error and previous time
become the current values.
  prevTime  = currTime;
  double output = Kp*error + Ki*errorI + Kd_output;
  return output;
}
```

**ISR Frequency**

| Operator | Instances | Cycle Count |
|---|---|---|

| | | |
|---|---|---|
| = | 25 | 100 |
| pinMode | 4 | 24 |
| attachInterrupt | 1 | 6 |
| For Loop | 2 | 10 |
| Multiplication | 7 | 126 |
| Division | 2 | 40 |
| Addition/subtraction | 10 | 180 |
| Setup() | 1 | 12 |
| If/else | 2 | 21 |
| Return | 2 | 6 |
| Analogread | 1 | 5 |
| Digitalread | 1 | 8 |
| Digitalwrite | 1 | 6 |
| **Total: 554 Cycles** | | |

The total run time of the Arduino code can now be calculated – we know that the microcontroller can run $16 \times 10^6$ cycles in a second with an operating frequency of 16 MHz, hence it takes 1/16MHz to run one cycle. Therefore:

$$Run\ time\ =\ Cycle\ Count \times \frac{1}{16\ MHz}\ =\ \frac{554}{16M}\ =\ 0.000032\ seconds$$

With a sampling time Ts in Simulink of 0.0001 seconds, which is roughly three times more than this value, we are being conservative enough to make sure that we are giving the Arduino code more than enough time to run on each iteration.

**Tuning Strategy**

To demonstrate how we obtained the initial tuning values (without the co simulation) just based off the transfer functions, we will consider the DCX14L motor. Initially, we can follow the 10-step PID tuning process

1) Obtaining the poles of the system transfer function

```
OPAMP = tf([A1] , [B1 B2]);
INTEGRATOR_TF = tf([1],  [1 0]);
SystemTF = MotorTF*OPAMP*INTEGRATOR_TF;
[z,p,k] = zpkdata(SystemTF);    %obtain zeros and poles from TF

system_poles = [-17065.6719111424;-209.653442554962;-23.6500000000000; 0]; %from zpk data
```

2) Obtaining the zeros, K-value and filter pole to append to the System TF to begin tuning

Zeros are designed midway between origin and the pole closest to the jw-axis. The zeros are complex conjugates of each other and are made to be 45 degrees above and below the x-axis.

The Ku is obtained from the current Gain Margin of the System obtained using the margin() function and K is chosen to be 10% of Ku. The filter pole for this calculation is chosen as 50 but

```
%Designing zeros to have half the magnitude of nearest pole to jw axis and starting angle 45
%tune zeros closer to jw axis for more stability and also reduce K to stabilise.

zeros_mag = abs(system_poles(3)/2);
zeros_angle = 45;
zeros_angle_rad = deg2rad(zeros_angle);
[Gm,Pm,Wcg,Wcp] = margin(SystemTF); %Gm is obtained in pure value and not dB.
Ku = Gm; %not in dB

%z's, P (N) and K
z1 = zeros_mag*exp(1i*zeros_angle_rad); %Note that this is not (pi - angle) because we want POSITIVE zero values
z2 = zeros_mag*exp(1i*-zeros_angle_rad); %Similarly not pi + angle
K = 0.1*Ku;
```

will be increased to 5000 later to account for averaging 8 samples.

3) Analyze the root locus, GM, and PM of the initial Controller using the zeros and pole

```
%New root locus and GM/PM plots to see improvement
s = tf('s');
PIDTF = K*((s+z1)*(s+z2))/(s*(s+filter_pole)) %Kp + Ki/s + Kd*(filter_pole*s)/(s+filter_pole);
PID_Filtered_TF = PIDTF*SystemTF;
```

$$K\frac{(s+Z_1)(s+Z_2)}{s(s+P)} \times SystemTF$$

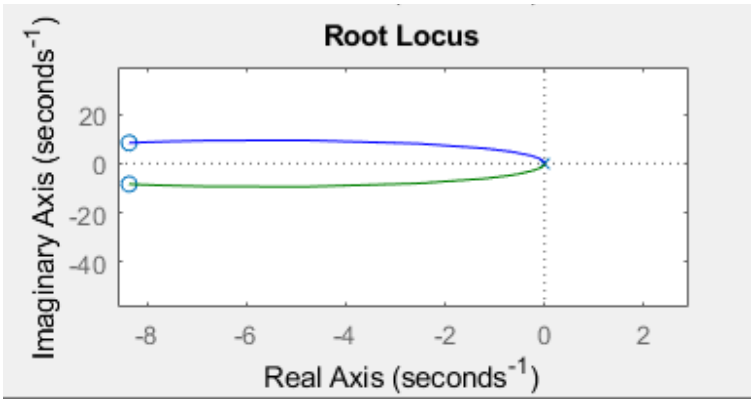


*Figure 1.5: Controller Root Locus - zeros capture the pole*

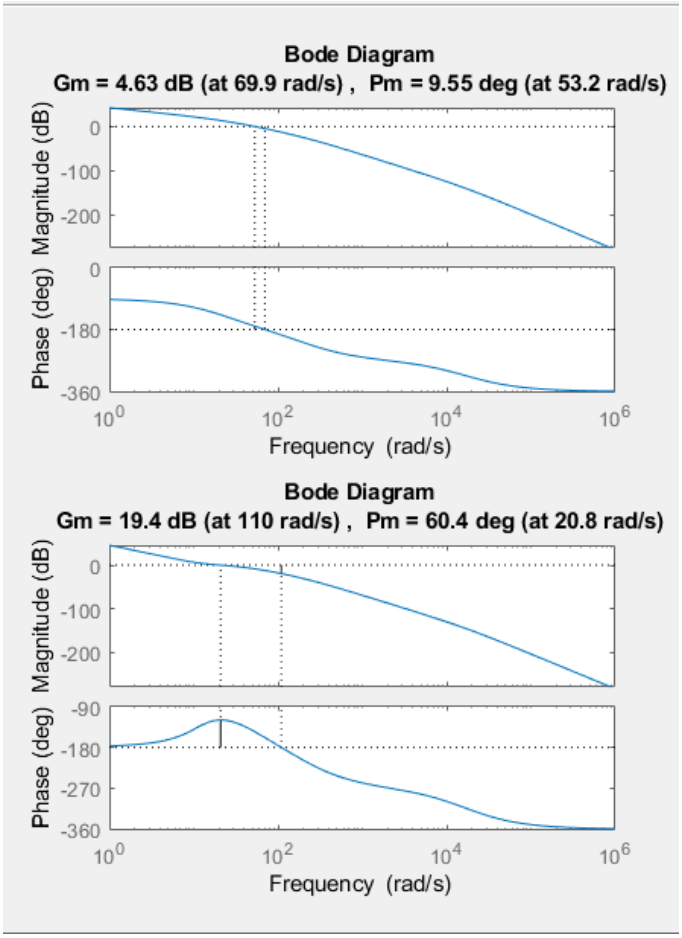



*Figure 1.6: The GM and PM have both increased after the addition of the Zeros, pole, and K gain*

Since GM x-over frequency > PM x-over frequency and both of them are positive, we know that the system is more relatively stable than before and can now obtain the tuning values using the formulas mentioned on the 10-step tuning guide:

$$K_I = \frac{KZ_1Z_2}{P}$$

$$K_P = \frac{K(Z_1+Z_2) - K_I}{P}$$

$$K_D = \frac{K - K_P}{P}$$

```
%PID gains
Ki = ((zl*z2)*K)/filter_pole;
Kp =  (K*(zl + z2) - Ki)/filter_pole;
Kd = (K - Kp)/filter_pole;
```

With these calculations we got the following values for tuning parameters for the DCX22L motor:

**Kp = 0.0474; Ki = 0.4764; Kd = 0.0025**

4) Evaluate and adjust parameters:

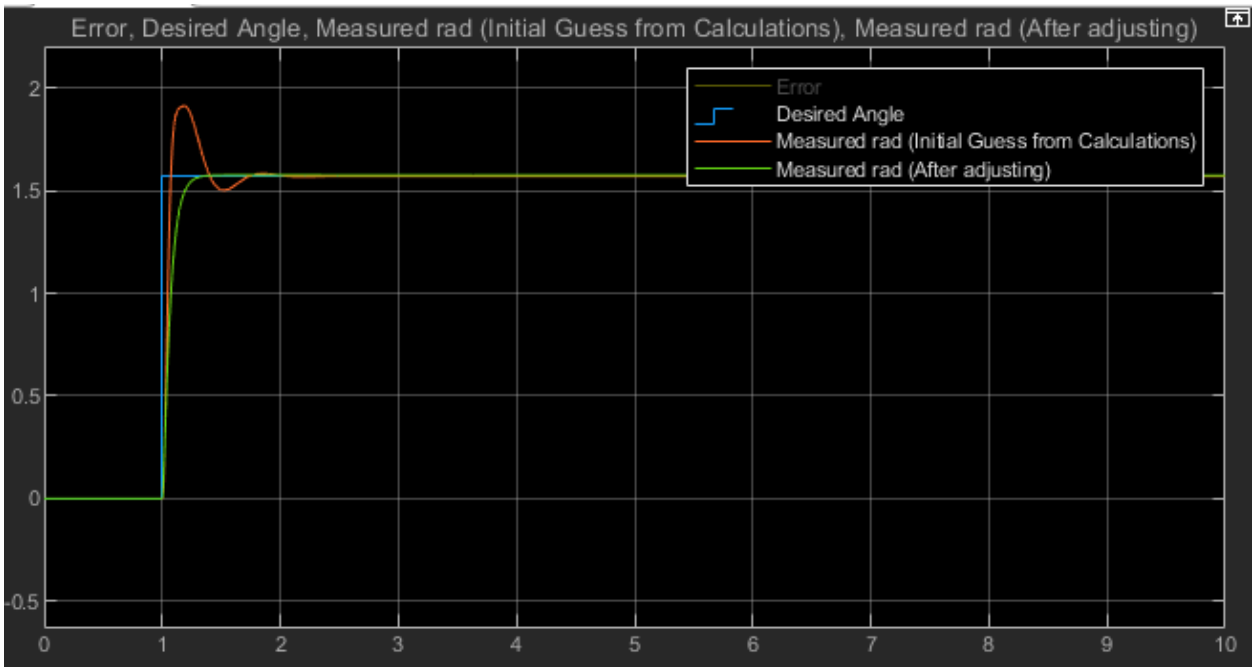



*Figure 1.7: Initial guess for tuning values gives little overshoot and a good starting point*

These tuning values produced an overshoot which we were able to get rid of by decreasing Ki and slightly increasing Kp and Kd to get a response with 0 overshoot and a small rise time. Tuning parameters after adjusting:

**Kp = 0.223809; Ki = 0.011836; Kd = 0.012272**

The tuned values for the other motors were in a similar range and were calculated using the 10-step method.

*Table 1.1 : Tuned values before cosimulation*

| Joint | Kp | Ki | Kd |
|---|---|---|---|
| Shoulder | 0.229 | 0.0118 | 0.0123 |
| Elbow | 0.132 | 1.02 | 7.34 |
| Wrist | 0.394 | 0.835 | 0.0474 |

The optical encoder (with 128 CPR) introduces discretization noise and static inertia and damping values become dynamic in the SimX simulation. The added mass of the arms and friction adds complexity to the mechanical system and may introduce non-linearities hence the parameters had to be re-tuned individually for each joint and in some cases the tuning values had to be changed quite a bit from their pre-co simulation values. The tuning was also affected by the fact that a gearhead gain was added in front of the motor torque constants to slow the motor down. This retuning was done with the help of reference table 1.2:

*Table 1.2: Reference table for adjusting tuning values with the cosimulation*

| | RISE TIME | OVERSHOOTS | SETTLING TIME | STEADY STATE ERROR |
|---|---|---|---|---|
| **Kp** | DECREASE | INCREASE | SMALL CHANGE | DECREASE |
| **Ki** | DECREASE | INCREASE | INCREASE | ELIMINATE |
| **Kd** | INCREASE | DECREASE | DECREASE | NO CHANGE |

**Tuning Shoulder Joint**

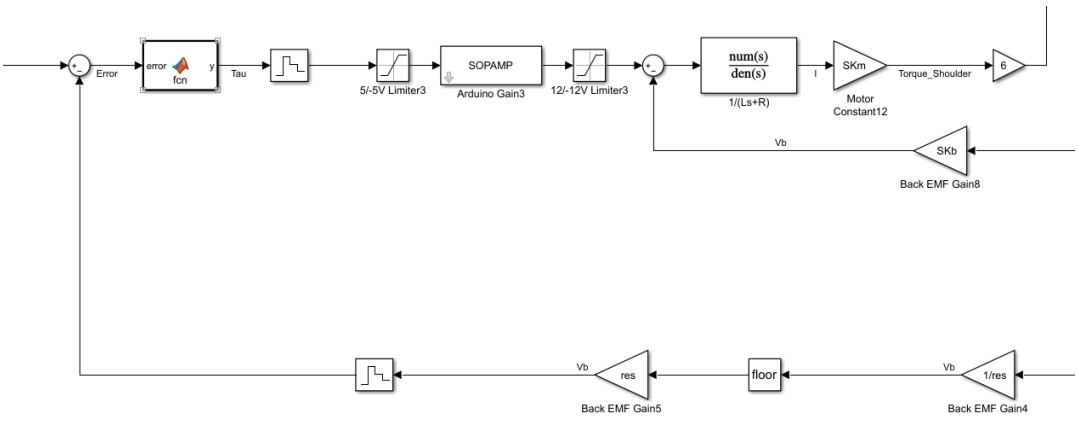


*Figure 1.8: Example of the Co-simulation Simulink model: Shoulder Joint (DCX32L) - added encoder noise and gearhead gain of 6 to reduce motor speed*

The shoulder joint was unstable after co-simulation with SimX with high oscillations so Ki had to be reduced by a factor of 10. To decrease rise time and settling time, Kp and Kd were both increased slightly.

**Tuning Elbow Joint:**

The Ki term was reduced by a factor of 2 to remove oscillations without compromising in steady state error. The settling time for this joint was higher than the shoulder and the oscillations were stronger, so we found that we had to increase Kd to a high number (≈ 50) to control the oscillations. These oscillations could be because of a strong shoulder motor which is exerting a high torque on the shoulder motor - hence we need a high Kd to get it under control.

**Tuning Wrist joint:**

This joint did not have many oscillations hence Ki did not have to be reduced by a high amount. Although the Kd was increased by approximately 2 units to control the settling time and reduce the overshoot a bit more.

*Table 1.3: Re-tuned values after co-simulation and non-linearities*

| Joint | Kp | Ki | Kd |
|---|---|---|---|
| Shoulder | 0.46 | 0.0045 | 0.06744 |
| Elbow | 0.015 | 0.0001 | 50 |
| Wrist | 0.016 | 0.00219 | 3.119 |

The gripper is controlled using a simple clock-switching mechanism with P-control, it will be shown at the end of this section.

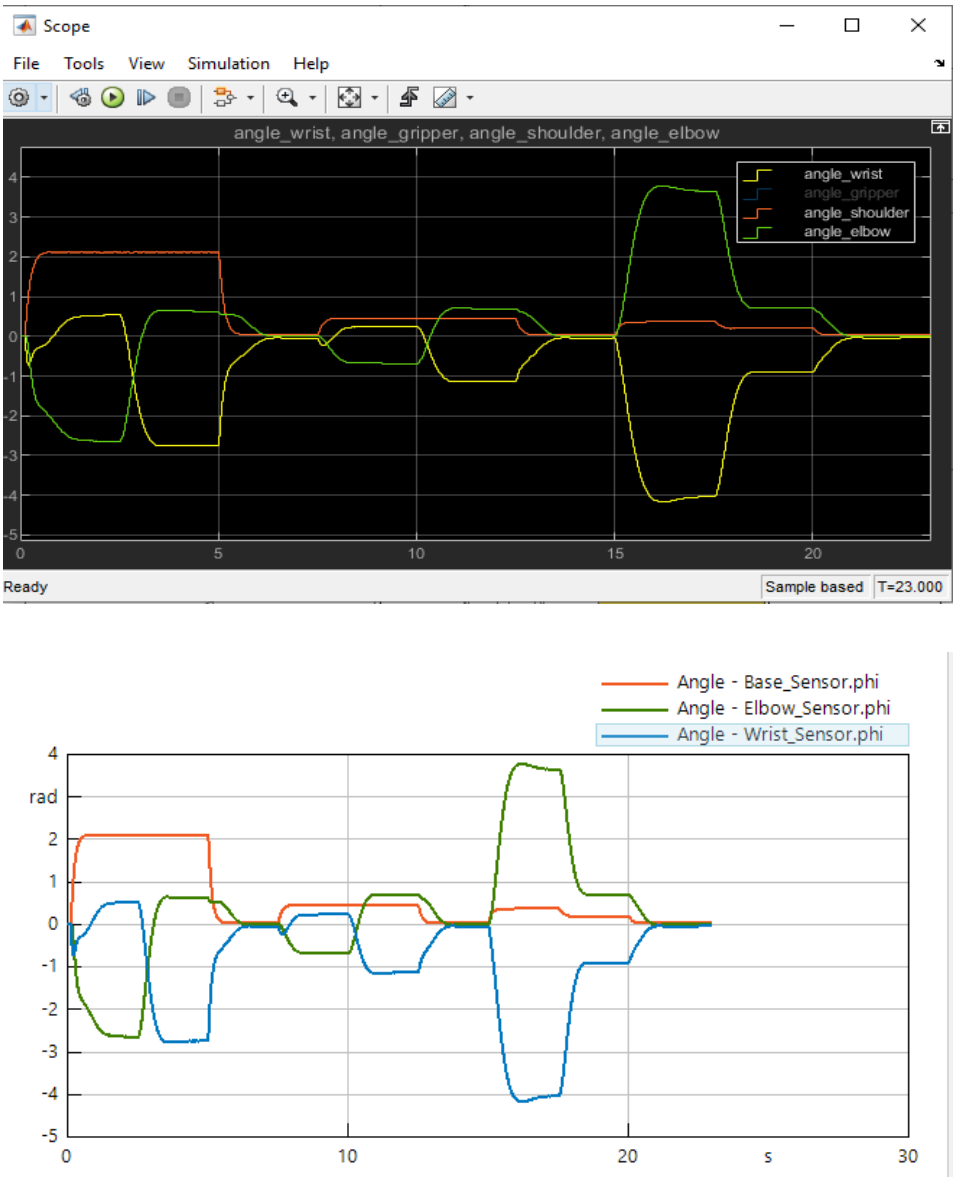


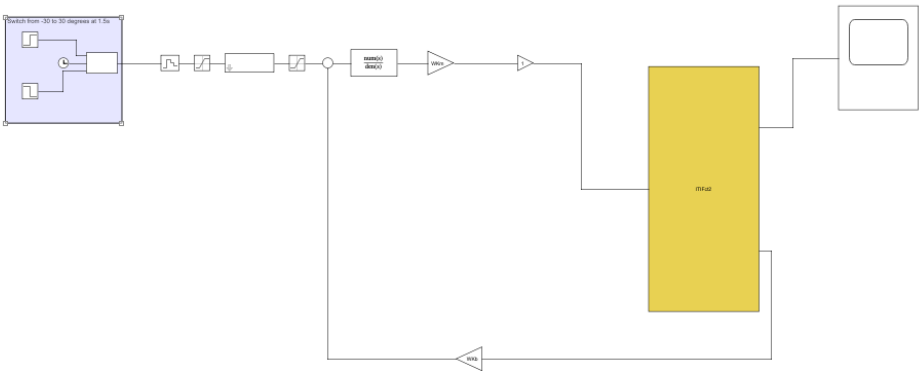

*Figure 1.9: Tuned Response of all three joints in 23 second simulation of grabbing all three marshmallows*

We can see that all the motors retain their position at every stage before moving to the next angle, there is 0 overshoot on any joint as the transition between one angle to the

next is very sharp. The rise time is never more than a second hence all of the control RCG's are met after careful tuning.

**Gripper Motor:**

The gripper motor is based on a simple clock-switching mechanism that toggles between 60 and -60 degrees using simple P control. It doesn't require tuning.



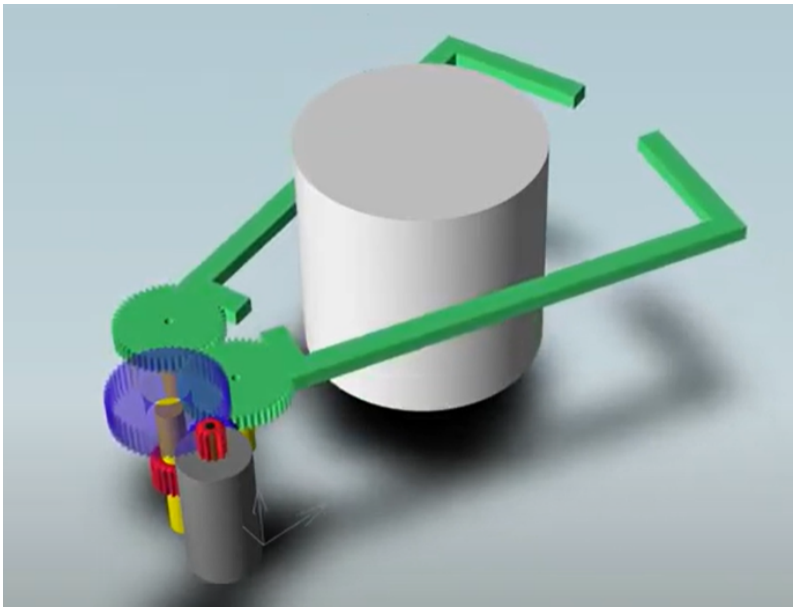


*Figure 2.0: Isolated Gripper Cosimulation*

## Path Planning and Inverse Kinematics

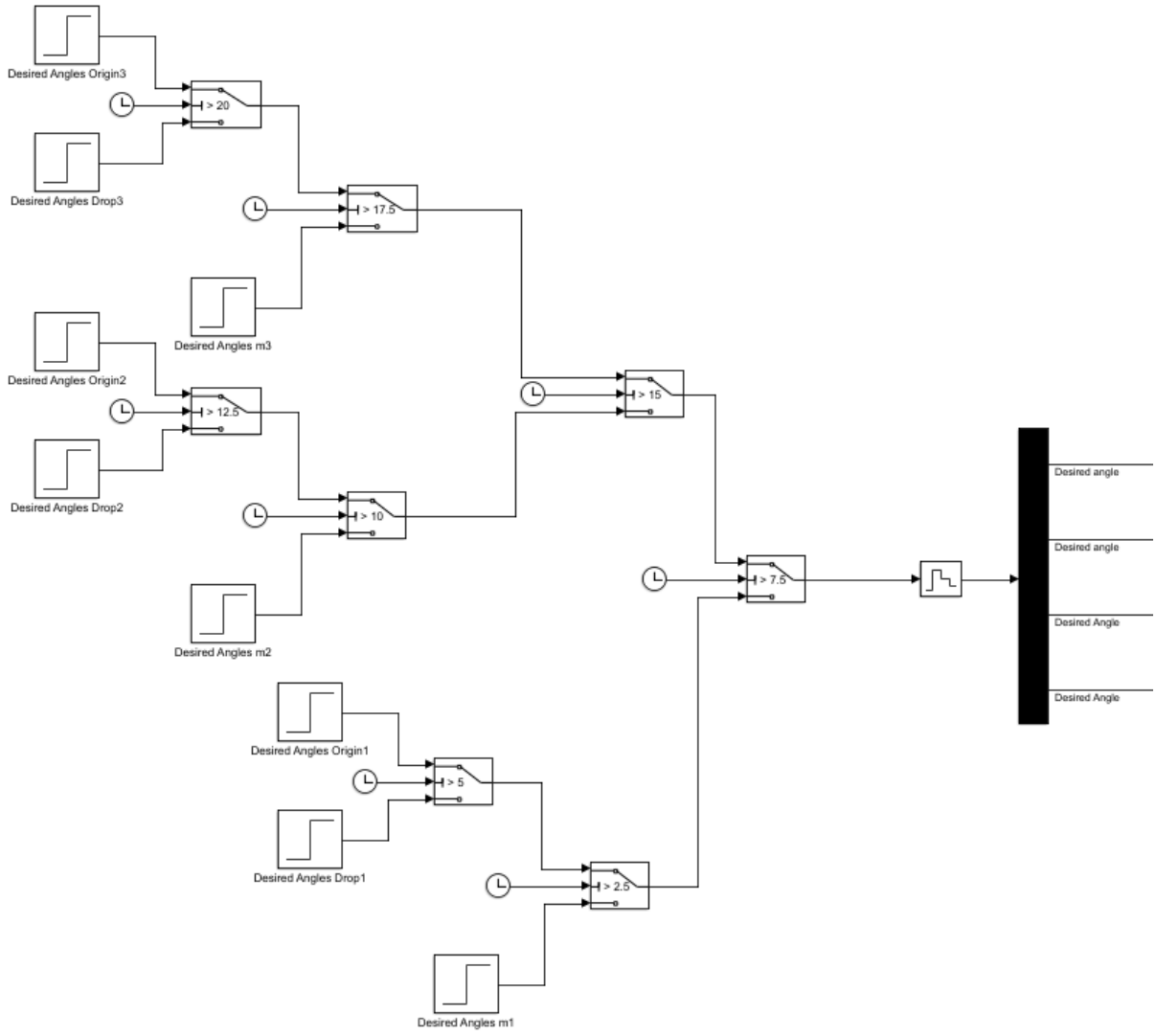
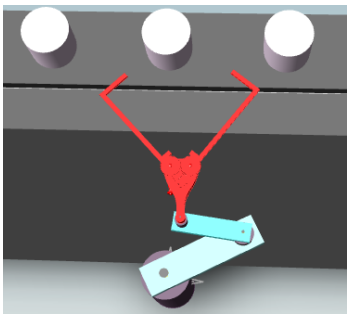


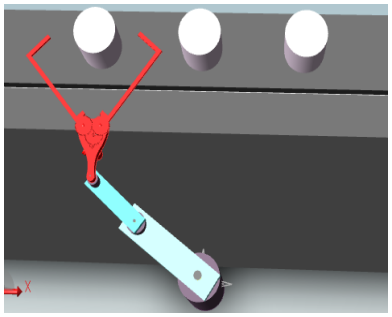

*Figure 2.1: Clock dependant switches to change angles mid-simulation*

We used clock dependent switches to grab each of the three marshmallows and drop them in the chute all in one simulation. The step inputs in the figure take in a vector of angles (gripper (set to

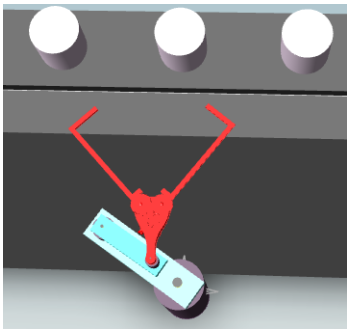
0 - co-simulated separately), wrist, elbow, shoulder) and these angles are determined using inverse kinematics calculations that I have shown in the next section. The clock-switch passes the top input if the condition on it is met, otherwise it passes the bottom input. The clock tracks the simulation time, so for example, since the simulation time at the start is less than 2.5 seconds, the switch at the very bottom of the figure 2.1 passes the third input which is a vector that instructs the marshmallow to pick up the 1st marshmallow. Similarly, other step inputs activate respectively as the simulation time progresses and other switches begin to flip. Finally a demux splits this vector into angles that go to their respective motors. The step input vectors as they relate to the path can be more easily visualized by correlating with the movement of the robotic arm:



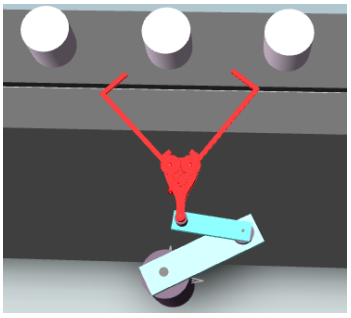
Resting Position (Origin)



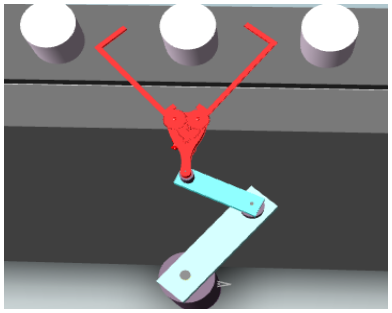
Desired Angles m1



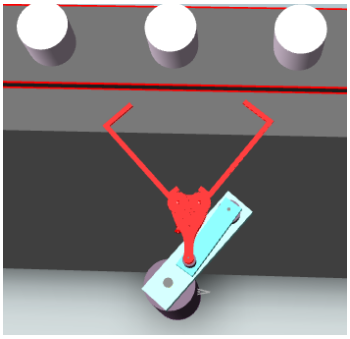
Desired Angles Drop1



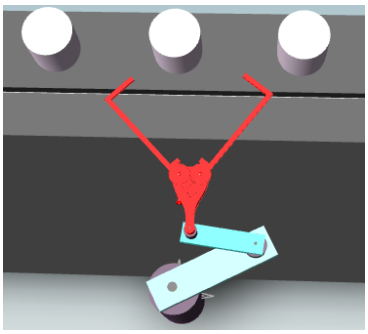
Resting Position (Origin)

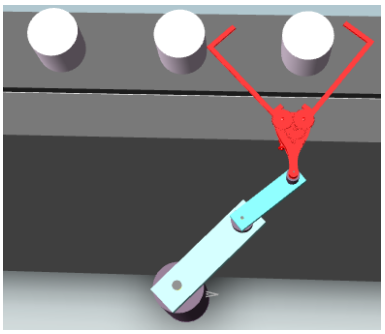


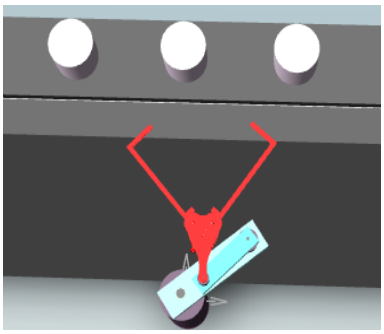
Desired Angles m2



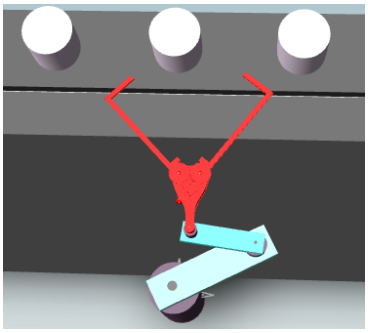
Desired Angles Drop2







Resting Position (Origin)  Desired Angles m3  Desired Angles Drop3



Resting Position (Origin)

*Figure 2.2: Step input vectors for each trajectory visualized*

For each of these labelled positions, we need the corresponding angle vectors to feed into the different step inputs mentioned above. We use inverse kinematics for this purpose in the next section.

**Inverse Kinematics**

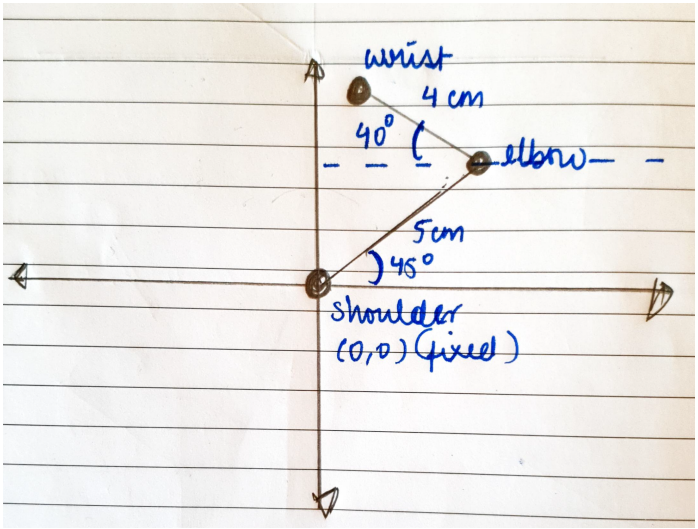


*Figure 2.3 Resting position of the robotic arm*

Before we can find the change in angles needed for the shoulder, elbow and wrist motor (note that the gripper doesn't need inverse kinematics), we need to establish the coordinates of each motor in its resting position. The shoulder arm length is 5 cm and the elbow arm length is 4 cm, they are both oriented at 45 degree and 40 degree angles respectively as shown in the figure 2.3. If we fix the shoulder motor's coordinates as (0,0), the resting coordinates of the elbow motor are $(5cos(45),\ 5sin(45)) = (3.53,\ 3.53)$

and the wrist motor's coordinates are

$$((3.53\ -\ 4\ *\ cos(40)), (3.53\ +\ 4\ *\ sin(40))$$

$$=\ (0.46,\ 6.1)$$

Now that we have the coordinates of the resting position, we can do the inverse kinematics for **grabbing** the first, second, and third marshmallow. These are the most critical steps for accurately grabbing - the marshmallow-dropping angles can be backtracked along the grabbing path and then the arm is reset to its resting position (by feeding zero angles to the motors to get them to their reference point). The desired coordinates for each marshmallow path were estimated using the Solidworks model, and these were then used to do inverse kinematics calculations that follow:

Grabbing 1st Marshmallow (*Desired Angles m1 - Figure 2.2*):

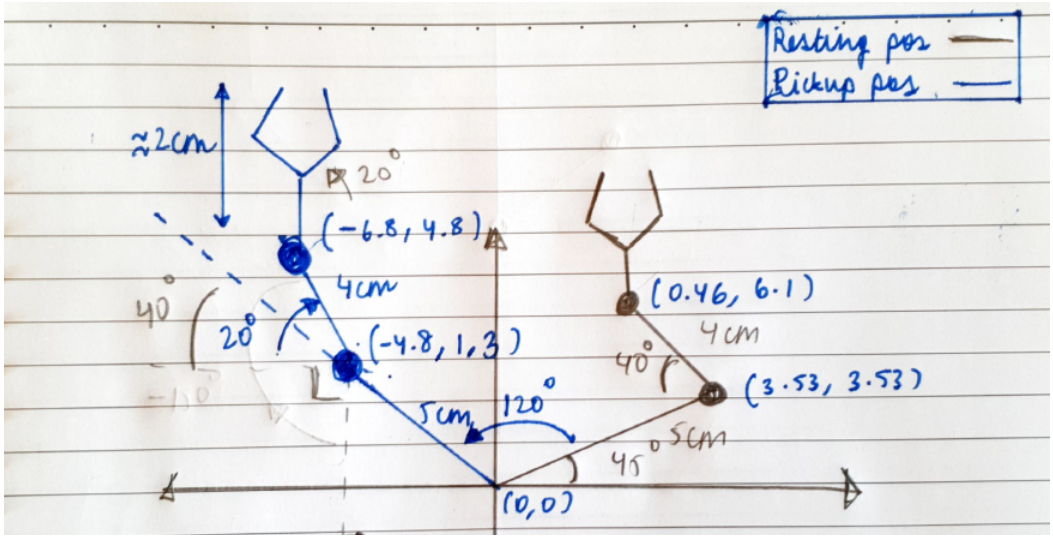


*Figure 2.4: Determination of angles of rotation from the coordinates needed to reach the first marshmallow*

The first (leftmost) marshmallow is located at approximately (-6.8, 7.0) which means it is approximately 10 cm away from the shoulder (base) motor. To get the gripper to this point, we relocate the motors from resting coordinates to the coordinates mentioned in the pen.

For the **shoulder arm**,

we have the distance between the new and old coordinates as

$$d = \sqrt{(-4.8 - 3.53)^2 + (1.3 - 3.53)^2} = 8.652$$

We have a triangle of sides 5 cm, 5 cm and 8.652 cm. We can find the angle opposite to the side with length 8.652 using the cosine rule:

$$8.652^2 = 5^2 + 5^2 - (2) * (5) * (5) * cos(\theta)$$

$$\text{Solving for } \theta \approx 120°$$

Similarly, for the elbow arm, we want the wrist motor to reach a coordinate of (-6.8, 4.8) so after determining in a similar fashion using the cosine rule, we determine that the arm should move clockwise by 20 degrees using the cosine rule with an isosceles triangle with two sides 4 cm, we can get the wrist to reach the required coordinate.

Orienting the wrist motor is very straightforward, especially in the way our SimulationX Sensors are referenced, because the wrist motor just offsets the displacement angle change from the elbow, thereby keeping the gripper pointing straight to the marshmallows at all times, hence we just use the negative angle displacement of $\Delta\Theta$ that the elbow experienced.

The gripper is an isolated co-simulation and is not PID controlled, hence no calculations are needed for the inverse kinematics for that motor.

Grabbing 2nd Marshmallow (*Desired Angles m2*):

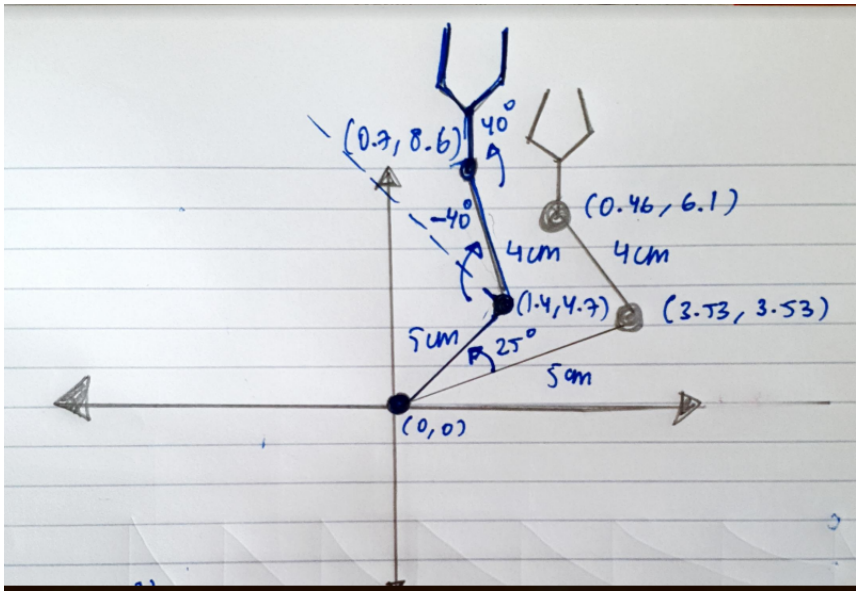


*Figure 2.5: Changes in coordinates and subsequent angle calculations for grabbing 2nd marshmallow*

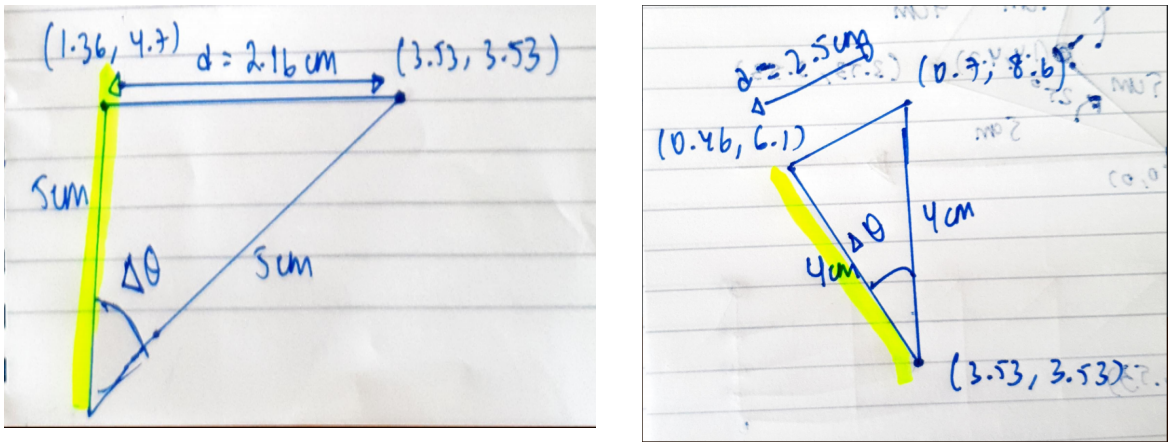



*Figure 2.6: Triangles for applying cosine rule for base (left) and elbow motor (right)*

Highlighted side shows the new arm positions. For the base motor,

$$d = \sqrt{(1.36 - 3.53)^2 + (4.7 - 3.53)^2} = 2.16$$

$$2.16^2 = 5^2 + 5^2 - (2) * (5) * (5) * cos(\theta)$$

$$\Delta\theta \approx 25°$$

For the elbow motor,

$$d = \sqrt{(8.6 - 6.1)^2 + (0.7 - 0.46)^2} = 2.51$$

$$2.51^2 = 4^2 + 4^2 - (2) * (4) * (4) * cos(\theta)$$

$$\Delta\theta \approx 40°$$

The wrist motor just offsets the change in the elbow angle to maintain a straight position for the gripper hence $\Delta\theta_{wrist} \approx -40$.

Grabbing 3rd marshmallow (*Desired Angles m3)*:

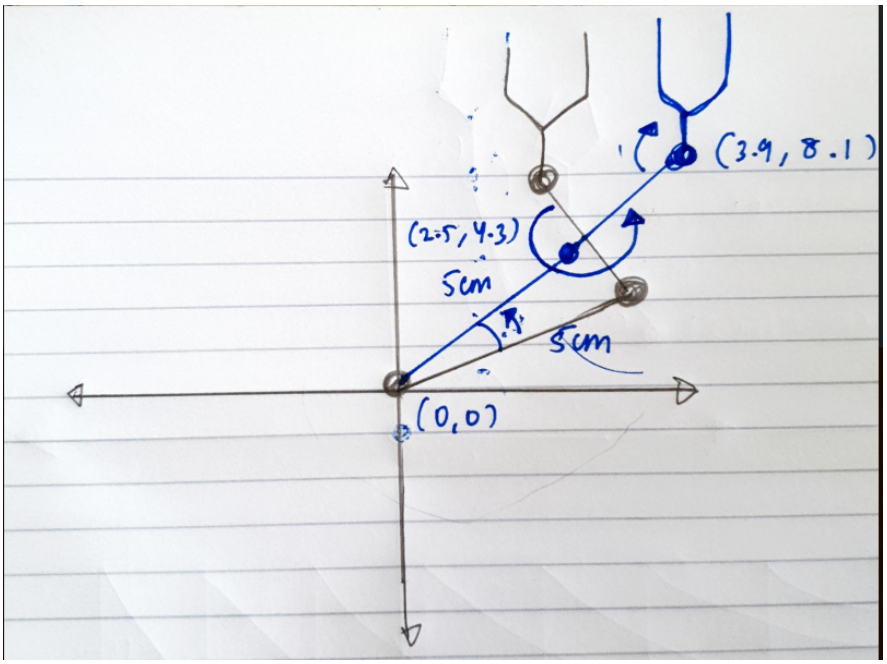



*Figure 2.7: Changes in coordinates and subsequent angle calculations for grabbing 3rd marshmallow*

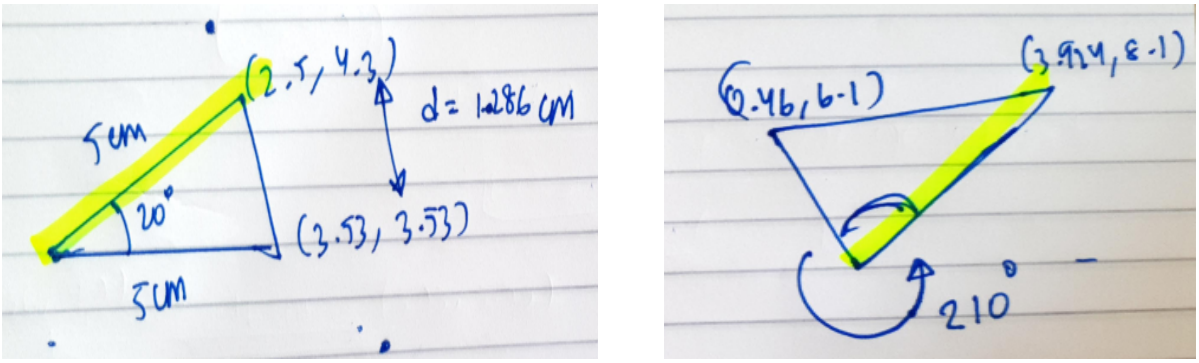



*Figure 2.8: Triangles for applying cosine rule for base (left) and elbow motor (right)*

For the base motor,

$$d = \sqrt{(2.5 - 3.53)^2 + (4.7 - 3.53)^2} = 1.42$$

$$1.42^2 = 5^2 + 5^2 - (2) * (5) * (5) * cos(\theta)$$

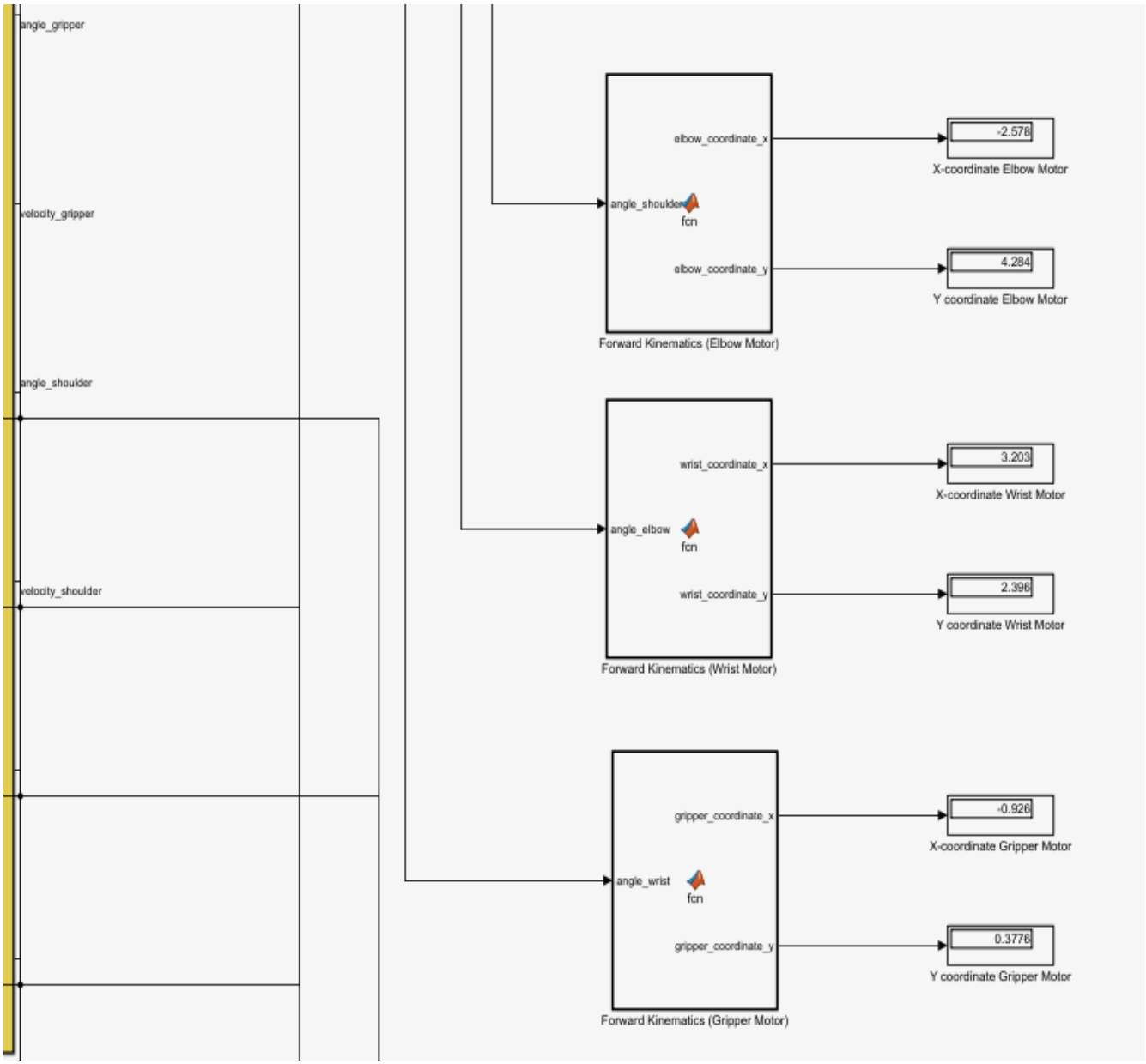
$$\Delta\theta \approx 20°$$

For the elbow motor, we do a similar calculation but this time we rotate the arm clockwise by applying 360 - θ**,** where θ is found from inverse kinematics calculations so as to avoid collision with the 2nd marshmallow. After doing the inverse kinematics to our planned path, using the predetermined desired coordinates from Solidworks, we get the following sets of angles that are fed as vectors to step inputs in the clock-switching model in figure 2.1 in path planning. This is the result of the inverse kinematics calculations:

```
%marshmallow 1
degree_vectors_m1 = [0 150 -150 120]; % gripper wrist elbow shoulder
desired_angle_vectors_m1 = deg2rad(degree_vectors_m1);
degree_vectors_drop1 = [0 -40 40 120]; % gripper wrist elbow shoulder
desired_angle_vectors_drop1 = deg2rad(degree_vectors_drop1);

%marshmallow 2
degree_vectors_m2 = [0 40 -40 25]; % gripper wrist elbow shoulder
desired_angle_vectors_m2 = deg2rad(degree_vectors_m2);
degree_vectors_drop2 = [0 -40 40 25]; % gripper wrist elbow shoulder
desired_angle_vectors_drop2 = deg2rad(degree_vectors_drop2);

%marshmallow 3
degree_vectors_m3 = [0 -210 210 20]; % gripper wrist elbow shoulder
desired_angle_vectors_m3 = deg2rad(degree_vectors_m3);
degree_vectors_drop3 = [0 -40 40 10]; % gripper wrist elbow shoulder
desired_angle_vectors_drop3 = deg2rad(degree_vectors_drop3);


desired_angle_vectors_origin = [0 0 0 0]; %resting position - implemented after every time a marshmallow is dropped in chute
```

*Figure 2.9: Highlighted vectors fed to the step inputs in clock switching figure*

## Forward Kinematics



*Forward kinematics to display the coordinates of all motors in real-time*

Note that the shoulder motor is fixed at (0,0). To implement the code for converting the angle exerted by the arm in joint space into movement of motors in task-space, a matlab function for Forward Kinematics was created (shown here just for the elbow motor as an example):

```
function [elbow_coordinate_x, elbow_coordinate_y] =
fcn(angle_shoulder)
angle_shoulder_deg = rad2deg(angle_shoulder);

% If the input angle is negative, make it positive for analysis by
adding 360
if angle_shoulder_deg < 0
    angle_shoulder_deg = 360 + angle_shoulder_deg;
end

%conditional statements for obtaining the coordinates of the elbow
and wrist motors from the angle traversed
if angle_shoulder_deg < 90
    elbow_coordinate_x = 5*cosd(angle_shoulder_deg);
    elbow_coordinate_y = 5*sind(angle_shoulder_deg);

elseif 90 < angle_shoulder_deg && angle_shoulder_deg < 180 %2nd
quadrant - x coordinates are negative
    elbow_coordinate_x = -5*cosd(180 - angle_shoulder_deg);
    elbow_coordinate_y = 5*sind(180 - angle_shoulder_deg);

elseif 180 < angle_shoulder_deg && angle_shoulder_deg < 270 %3rd
quadrant - x coordinates are negative
    elbow_coordinate_x = -5*cosd(angle_shoulder_deg - 180);
    elbow_coordinate_y = 5*sind(angle_shoulder_deg - 180);

else                        %4th quadrant - x coordinates are negative
    elbow_coordinate_x = 5*cosd(360 - angle_shoulder_deg);
    elbow_coordinate_y = -5*sind(360 - angle_shoulder_deg);
end
```

```
end
```

Similar code was created for the wrist and gripper and the only thing that changes is the arm lengths and factoring in the previous arm - for example getting the x-coordinate for the wrist motor would involve projecting the wrist arm on the x-axis and adding the x-coordinate contribution from the elbow motor because the wrist rotates about the elbow and **not** the origin.

## References


Tuning a PID Controller¶. (n.d.). Retrieved from https://compendium.readthedocs.io/en/latest/robotcode/components/pid/pidcontroller2.html

L. Stocco *PID Tuning 10-Step Process* (n.d.) Retrieved from

PID-10.pdf: ELEC 391 201 2020W Electrical Engineering Design Studio II (ubc.ca)

L. Stocco *Digital Filters (lecture)* (n.d.) Retrieved from

Digital Filters (ubc.ca)

P Corke *Inverse Kinematics for a 2-Joint Robot Arm Using Geometry: Lesson.*(2018, July 30). Retrieved from https://robotacademy.net.au/lesson/inverse-kinematics-for-a-2-joint-robot-arm-using-geometry/


## Appendices

### APPENDIX A

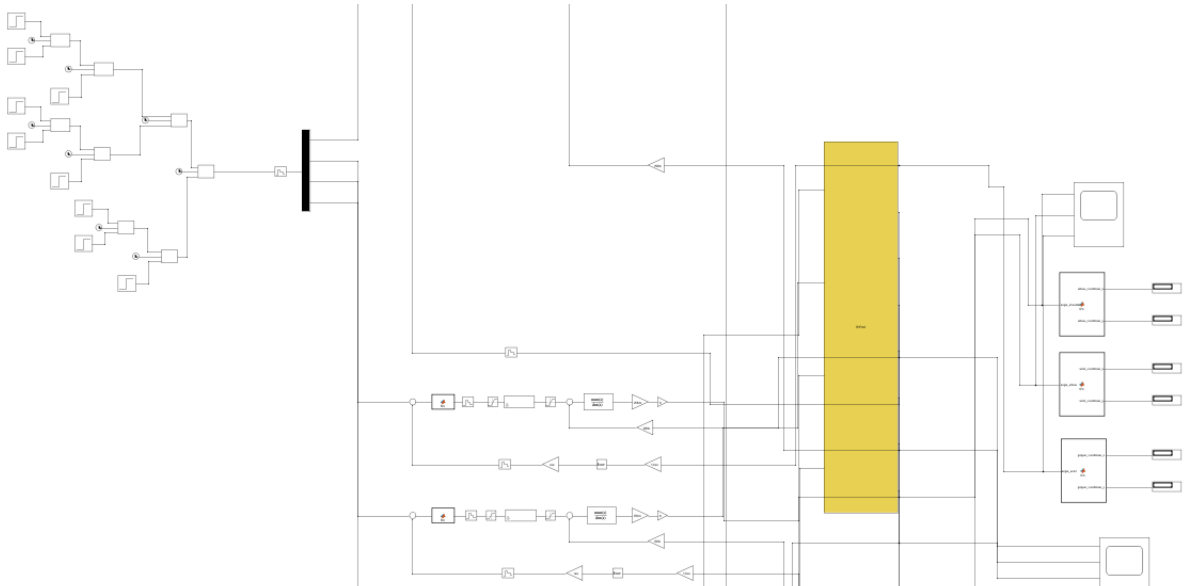


Figure 1.1: Cosimulation with SimulationX (Gripper angle set to 0 for isolated gripper simulation)

**APPENDIX B**

- Forward Kinematics Code for wrist and gripper position

This code is very similar to the one for elbow position except the arm length is changed accordingly from 5 cm to 4 cm and 1 cm respectively.

```
function [wrist_coordinate_x, wrist_coordinate_y] = fcn(angle_elbow)
angle_elbow_deg = rad2deg(angle_elbow);

% If the input angle is negative, make it positive for analysis by
adding 360
if angle_elbow_deg < 0
    angle_elbow_deg = 360 + angle_elbow_deg;
end

%conditional statements for obtaining the coordinates of the elbow
and wrist motors from the angle traversed
if angle_elbow_deg < 90
    wrist_coordinate_x = 4*cosd(angle_elbow_deg);
    wrist_coordinate_y = 4*sind(angle_elbow_deg);

elseif 90 < angle_elbow_deg && angle_elbow_deg < 180 %2nd quadrant -
x coordinates are negative
    wrist_coordinate_x = -4*cosd(180 - angle_elbow_deg);
    wrist_coordinate_y = 4*sind(180 - angle_elbow_deg);

elseif 180 < angle_elbow_deg && angle_elbow_deg < 270 %3rd quadrant -
x coordinates are negative
    wrist_coordinate_x = -4*cosd(angle_elbow_deg - 180);
    wrist_coordinate_y = 4*sind(angle_elbow_deg - 180);

else                                                        %4th
quadrant - x coordinates are negative
    wrist_coordinate_x = 4*cosd(360 - angle_elbow_deg);
    wrist_coordinate_y = -4*sind(360 - angle_elbow_deg);
end
```

```
end
```

*Gripper position*

```
function [gripper_coordinate_x, gripper_coordinate_y] =
fcn(angle_wrist)
angle_wrist_deg = rad2deg(angle_wrist);

% If the input angle is negative, make it positive for analysis by
adding 360
if angle_wrist_deg < 0
    angle_wrist_deg = 360 + angle_wrist_deg;
end

%conditional statements for obtaining the coordinates of the elbow
and wrist motors from the angle traversed
if angle_wrist_deg < 90
    gripper_coordinate_x = 1*cosd(angle_wrist_deg);
    gripper_coordinate_y = 1*sind(angle_wrist_deg);

elseif 90 < angle_wrist_deg && angle_wrist_deg < 180 %2nd quadrant -
x coordinates are negative
    gripper_coordinate_x = -1*cosd(180 - angle_wrist_deg);
    gripper_coordinate_y = 1*sind(180 - angle_wrist_deg);

elseif 180 < angle_wrist_deg && angle_wrist_deg < 270 %3rd quadrant -
x coordinates are negative
    gripper_coordinate_x = -1*cosd(angle_wrist_deg - 180);
    gripper_coordinate_y = 1*sind(angle_wrist_deg - 180);

else                                                    %4th
quadrant - x coordinates are negative
    gripper_coordinate_x = 1*cosd(360 - angle_wrist_deg);
    gripper_coordinate_y = -1*sind(360 - angle_wrist_deg);
end

end
```