Virtual Quality Control Robot – Control

White Paper **Updated – April 13, 2021**

Deniz Tabakci, Group Maracas, ECE, University of BC, Vancouver, BC, Canada

Abstract

The Robotic Arm is controlled by 3 tuned PID controllers and 1 P controller. The simulation for the controllers are done using MATLAB functions and the expected ISR rate of the Arduino Leonardo. Motor data from Maxon Motors and electrical circuit Linear Models imported from Multisim are used to create the simulation models in Simulink. Simulink is used together with SimulationX (Referred to as Co-Sim going forth) to simulate the motor and its connections.

In this paper Section 1 describes the process of creating the PID controller in MATLAB and testing by comparing with the Simulink's own PID controller. Section 2 describes the conversion of the MATLAB Function code for the PID into C code for compilation using Arduino Leonardo. ISR Rate calculations are also included in Section 2. Section 3 describes the Strategy used to tune the PID controllers connected to each motor for both the Simulink and the Simulink/SimX Co-Sim Model. Section 4 describes Inverse Kinematics of the Robot and how the angle inputs for the Robotic Arm were calculated and used. Section 5 describes the path planning for picking up and dropping each Marshmallow. The goal of Path Planning is to ensure that the arm moves in an acceptable manner (e.g. does not knock the marshmallows out of the conveyor belt).

Nomenclature

PID Proportional, Integral, Derivative (controller)

1. PID Controller, MATLAB

The code for the PID controller was made up of 4 disinct parts. The initilization (setup), P for proprotional, I for Integral and D for Derivative. The P and the I parts were simple to implement whereas the D part first required the implementation of a digital filter before the Derivative could be implemented.

1.1 Initilization

```
function Output = PID(error)

persistent ept              %holds the exponentials
persistent results          %holds reccorded derivatives
persistent previous_error   %holds input from last execution for derivative calc
persistent integral         %holds the total integral
CF = 10000;                 %ISR frequency
delta_t = 1/CF;             %Will change according to CF, sampling time
size = 8;                   %can change between 5-10
n = 4;                      %Decaying until n*tau, can be 4 or 5
pole = n*CF/size;
P_gain = 0.2616;
D_gain = 0.0432;
I_gain = 13.390;
```

```
%initialize vectors and variables
if isempty(results)
    results = zeros(1,size);
    ept = zeros(1,size);
    sum = 0;
    integral = 0;
    previous_error = 0;

    %calculate scale factor for derivative
    for i = 1:size
        sum = sum + pole*exp(-pole*i*delta_t);
    end
    Scale_Factor = 1/(sum);

    %properly create ept
    for i = 1:size
        ept(i) = Scale_Factor*pole*exp(-pole*i*delta_t);
    end
end
```

Figure 1.1 : The initilization of the MATLAB Function

As can be seen in Figure 1.1, the Persistent vectors and variables are created to hold the data from the previous runs of the function. Along with the persistent variables normal variables are also created in this section. Then these variables are initialized during the first run of the code. The initilization of the code mainly focuses on creating the vectors and variables needed to

create the D controller and its respective Digital Filter such as the ept vector or the Scale Factor. The calculations can be seen in Figure1.2.

1.2 Proportional (P)

The Proportional controller is relatively the easiest controller to implement among the PID. The P controller simply outputs the error (input of the PID) multiplied by a pre-defined P-gain.

$$Proportional = error$$

1.3 Integral (I)

The Integral controller is also relatively easy to implement compared to the D controller. The integral controller simply output the Integral of its inputs multiplied by a pre-defined I-gain. A persistent variable is kept as a cumulative sum of each error added to the error from the previous run, halved and multipled by the time-step :

$$Integral += \frac{error + previous\_error}{2} * T_S$$

1.4 Derivative (D)

The Derivative controller is the most complex controller to implement out of the three mainly because implementing a Derivative controller also requires the implementation of a Digital Filter[1]. The output of the D controller is the output of the filtered derivative multiplied by the pre-defined D-gain.

$$Derivative = \sum_{n=1}^{size} Scale\ Factor * p * e^{-p*n*T_S} * \frac{(error_n - error_{n-1})}{T_S},\ (error_{n-1} = 0 \quad if \quad n = 1)$$

```
%Calculating the D(Derivative) of PID
results(2:size)=results(1:size-1);
results(1)=(error - previous_error)/delta_t;
previous_error = error;
derivative = dot(results,ept);

%Calculating the I(Integral) of PID
integral = integral + (error + previous_error)/2*delta_t;

%Calculating the P(Proportional) of PID
proportional = error*P_gain;

Output = proportional + I_gain*integral + D_gain*derivative;
```

Figure 1.2 : The calculations for each of PID and for the output

Figure 1.3 is the difference between the Simulink PID and the User-Created PID

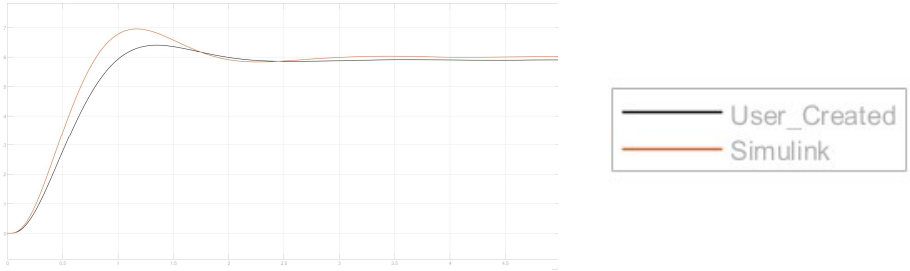

Figure 1.3 : The user created controller juxtaposed with the Simulink PID controller

2. PID Controller, Arduino(C)

The MATLAB code was converted into a PID controller written in C (for Arduino). The clock cycles were then counted[2] and used to determine the ISR rate to be used with the PID controller and the simulation. Note that the initilization of variables and vectors are not included in the clock cycle count as the process of initilization only occurs once. Thus only the loop function is incldued in the clock cycle count. The cycle count was multiplied by the number of PID controllers used in the simulation. Since three motors were controlled by a PID controller the total cycle count was multiplied by three. A seperate clock cycle count was done for the P controller and added to the total count as well. The code can be seen in Figure 2.1.

```
void setup() {
  for(i = 0; i<8; i++){
    results[i] = 0;
  }
  for(i=0;i<8;i++){
      sum+= pole*exp(-pole*i*delta_t);
  }
  Scale_Factor = 1/(sum);
  for(i=0;i<8;i++){
      ept[i] = Scale_Factor*pole*exp(-pole*i*delta_t);
  }
}
```

```
void loop() {                                                          //10
  error = analogRead(0);                                               //5+4
  for(i = 0; i<7; i++){                                                //5
    results[i]=results[i+1];                                           //(4+18)*7
  }
  results[0]=(error - previous_error)/delta_t;                         //4+18+20
  previous_error = error;                                              //4
  for(i = 0; i<8; i++){                                                //5
    derivative+=results[i]*ept[i];                                     //8*(4+2*18)
  }
  integral = integral + (error + previous_error)/2*delta_t;            //4+3*18+20
  proportional = error*P_gain;                                         //4+18
  analogWrite(0,proportional + I_gain*integral + D_gain*derivative);   //6+4*18
  delay(#)                                                             //3
}
```

Figure 2.1 : The C Code written for Arduino. The Clock cycle counts are commented on the right side

$$Total\ Cycle\ Count = 3 * PID\ Count + P\ Count = 730 * 3 + 50 = 2240$$

$$ISR_{actual} = \frac{Arduino\ Clock\ Frequency}{Total\ Cycle\ Count} = \frac{16000000}{2240} \approx 7150\ Hz$$

To be on the safer side the ISR Rate is halved to give the loop roughly double the amount of time it actually needs.

$$ISR_{used} = \frac{7150}{2} = 3575\ Hz, \qquad T_S = \frac{1}{3575} = 0.280ms$$

Note that there was no code used for the calculation of the inverse or forward kinematics so no runtime was allocated for the process. The process for these calculations are detailed in Section 4. Robot Kinematics.

3. Tuning

The tuning process of the controllers are very similar at each joint. Initially each motor is tuned seperately using the motor data and linear models for the PCB circuits imported from Multisim. Since all motors are the same this process was not repeated. Once the ideal gain values were calculated for each motor, these gains were used as a starting point for tuning the joints of the arm. The priority for tuning the joints was to decrease overshoot as much as possible followed by the decrease of rise time and settle time. The steady state error was also kept in check, however, this error remained negligible thoroughout the tuning process. The tuned controller were tested with variety of angle transitions to ensure the controller was versatile.

3.1 Tuning the Motors

First a Simulink model was created using the linear models for each of the motors as can be seen in Figure 3.1.

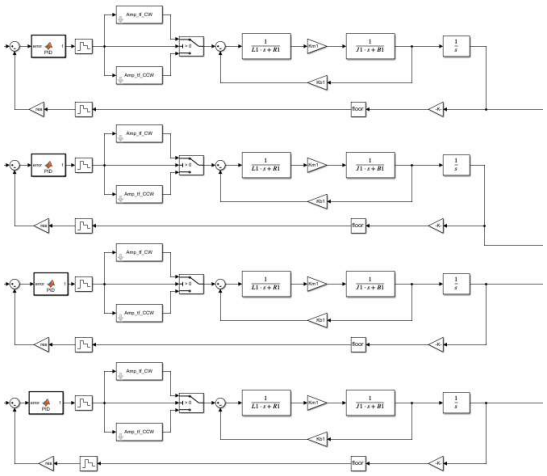

Figure 3.1 : Simulink model for each of the motors. Shoulder, Elbow, Wrist and Gripper

The 10 step tuning process[3] was used to tune the motors. First a system transfer function was calculated using the open-loop gain and a root locus graph was created using MATLAB. This graph was used to determine the location of the zeroes. The zeroes were placed so that they would attract the most dominant non-zero pole and thus increasing the gain. Once the zeroes were calculated a new root locus is made with the controller included in the transfer function. The ultimate gain Ku is measured using the new root locus and the K value is chosen to be up to %50 of the Ku. For this project the gain K was chosen to be %10 of the Ku measured. Using the calculations in the process[3], the PID controller gains were calculated.

3.2 Tuning for Co-Sim

Once the parameters for the motors were decided on the tuning for the Co-Sim model began. Each motor were connected to their respective joints in SimulationX, upper arm, forearm, wrist and gripper. Using tuned control parameters for the motors as a starting point the controller was tested again with the arms connected until an acceptable settle-time and overshoot were reached. Friction was included at the joints. For lowering overshoot the P gain and the I gain were lowered accordingly and the D gain was increased compared to the starting point. To keep the run time short, a lot of combinations were tested and a combination that would give zero overshoot and the highest rise-time was chosen. Once the joints were tuned with their respective motors then the whole arm was connected together. Using the tuned controller

parameters, the same process was repeated until the whole arm was performing with near to no overshoot and as high of a rise-time as possible. Graphs for the angles can be seen in Figure 3.2.

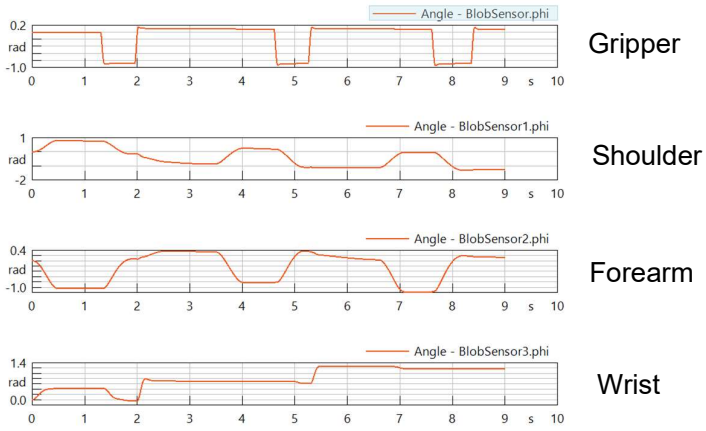

Figure 3.2 : Recorded movements of the joints after Co-Sim

As can be seen the arm has decent settle time and overshoot thats acceptable. It was decided that as long as it was not interfering with the arms movements or knocking down marshmallows minimal overshoot was acceptable. By allowing negligible overshoot the rise time could be greatly improved and the whole motion of the arm picking up and dropping the three marshmallows could be completed in 9 seconds. It can be seen in the graphs that a perfect settle is not reached at some points before transitioning to a different angle. This is a design choice and it is explained more in debt in Section 5. Path Planning. The tuned parameters can be seen in Figure 3.3 below.

	Initial Tuning for Simulink			Final Tuning for Co-Sim		
Gains :	P	I	D	P	I	D
Shoulder	0.2616	13.390	0.0432	0.2	0.03	3
Forearm	0.2616	13.390	0.0432	0.2	0.03	2.89
Wrist	0.2616	13.390	0.0432	0.2616	0.01	0.432
Gripper	0.1			0.1		

Figure 3.3 : The control gains for the PID controler

4. Robot Kinematics

The calculations were necessary to figure out exactly at which angles the motors needed to move to for the simulation. Writing code for this process was considered where the code would calculate the angles and input thesse angles into the Simulink model. However, to save runtime, another solution was improvised. Since writing code would mean extra runtime would have to be allocated, it was decided that the angles would be calculated using a 2D Solidworks simulation (Explained in further detail in Section 1 of the Mechanical Report). These angles were saved in MATLAB and were input into the Simulink model with an original design.

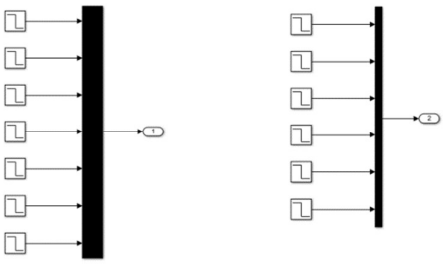

Figure 4.1 : Step functions with timings.

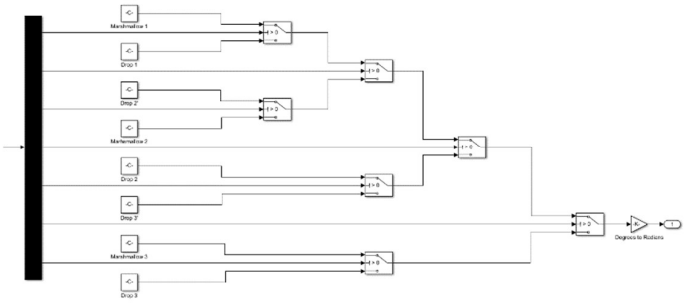

Figure 4.2 : Switch system to transition between different positions

Figure 4.1 is the input to the system. Each step function has a specific time in which they go from 1 to 0 (the opposite of what normally happens). This change from 1 to 0 signals the switches in Figure 4.2 to flip and thus the input is able to transition between each pre-determined location with also predetermined timings. Each motor has a subsystem identical to Figure 4.2 connected as an input. Each of these subsystems have motor specific angles input as MATLAB variables. For example when the first step function activates each motors input changes from the angles for PickUp1 to the angles for Drop1. Since all of these inputs, the angles and the times, are MATLAB variables it is very easy to adjust it.

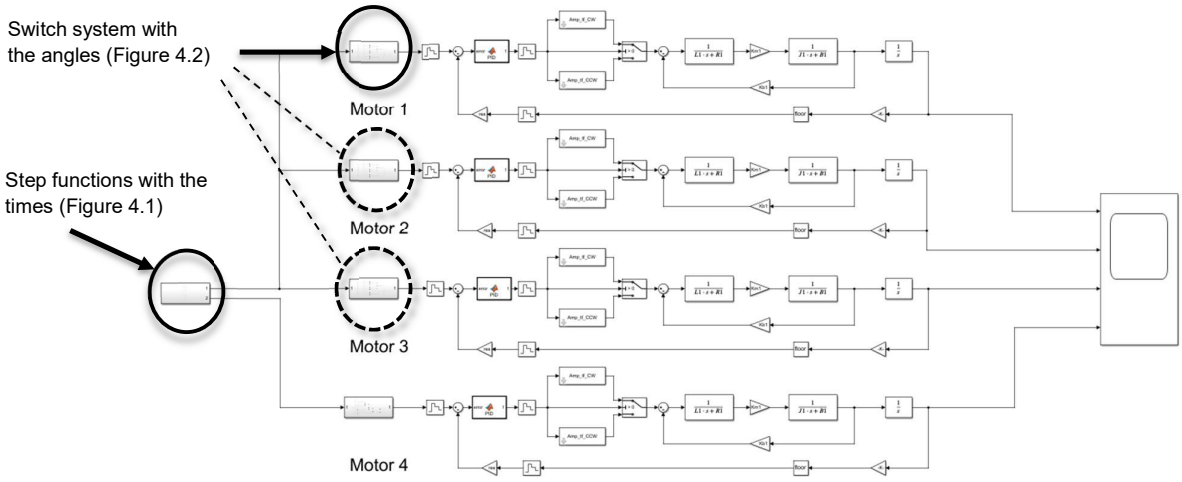

Figure 4.3 : The Simulink model with the input susbsystems highlighted

Note that motor 4 is the gripper and has a different system attached. Because the gripper was not responsible for the movement of the arm but for picking up the marshmallows, it has a different set of time inputs (there are two outputs in the subsystem in Figure 4.1) and its angles only switch between 0 and 45 degrees.

The simulation used for the calculations of the angles can be seen in Figure 4.4.

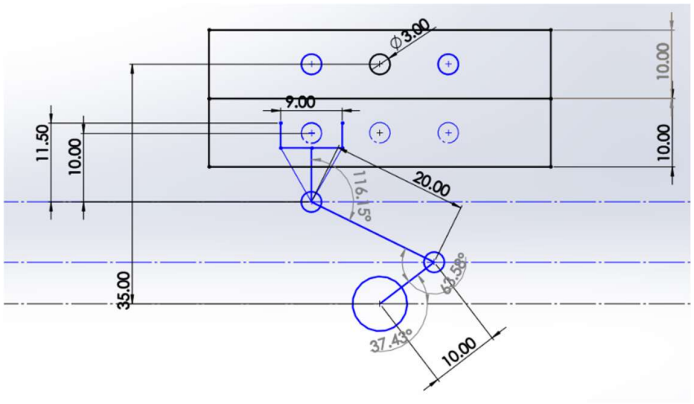

Figure 4.4 : The Soliworks simulation used for calculating the angles. This one in particular is the position of Drop 1

5. Path Planning

The Path Planning was also done manually as we knew the order in which the arm would move on. It was decided that the arm would go as :

$$Start \rightarrow PickUp1 \rightarrow Drop1 \rightarrow PickUp2 \rightarrow Drop2 \rightarrow PickUp3 \rightarrow Drop3$$

The ordering is from left to right and each respective drop is below its PickUp location as can be seen in Figure 4.4. This was the comon sense ordering but the main problem with this ordering was that the moves from Drop1 to PickUp2 or Drop2 to PickUp3 would take a diagonal path knocking over the marshmallow before attempting to pick it up. To fix this we added intermediate steps between a drop and pick up. The new order of movement became :

$$Start \rightarrow PickUp1 \rightarrow Drop1 \rightarrow Drop2 \rightarrow PickUp2 \rightarrow Drop2 \rightarrow Drop3 \rightarrow PickUp3 \rightarrow Drop3$$

In other words the arm would actually move to the drop position of the marshmallow before it moved to that mashmallow. This ensured that the arm would never take horizontal action towards the pick up locations thus knocking over the marshmallows to the side.

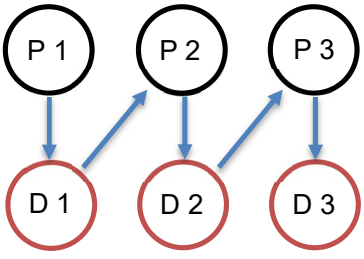

Figure 5.1 : The original path

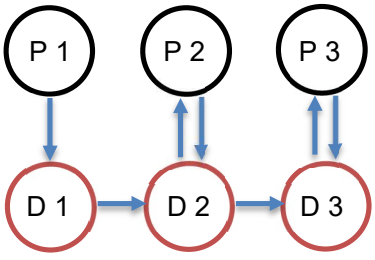

Figure 5.2 : The improved path

Since the intermediate step of horizontal movement was not technically necessary, the decision to not wait for the settle time was made. Taking Figure 5.2 as the reference, the move from D1 to D2 would not actually perfectly settle. As soon as the arm was at the acceptable position, the move to P2 would start. This is the reason for why the graphs in Figure 3.2 do not settle

perfectly at times but start moving to a different position nonetheless. Doing this allowed the arm to finish the process quicker while still doing it properly.

References

[1] Prof. Leo Stocco, UBC(2020) Digital Filters

https://canvas.ubc.ca/courses/59458/modules/items/2671456

[2] Prof. Leo Stocco, UBC(2020)Arduino Clock Cycles

http://ece.ubc.ca/~leos/pdf/e391/Project/ArduinoClockCycles.pdf

[3] Prof. Leo Stocco, UBC(2020) 10-Step PID Tuning Process

https://canvas.ubc.ca/courses/59458/files/12924258?module_item_id=2915586