3.5 DOF SCARA Robot Arm Controls Report

White Paper **Updated - April 13th, 2021.**

Luke Bidulka - 62893870 ECE, University of BC, Vancouver, Canada
Nikita Drozdovs - 93522522 ECE, University of BC, Vancouver, Canada
Zachary Chow - 40475931 ECE, University of BC, Vancouver, Canada

Abstract
In this paper, we describe how the control system was designed and simulated for a 3.5 DOF SCARA robot arm.
Section 1 outlines the design goals and requirements. Section 2 describes the detailed controls design of the arm and its subcomponents. The first sub-section of controls is an overview and an introduction to our system. The second sub-section covers PID implementation using custom MATLAB function block, and how we calculated each of the gains. Sub-section 3 is the implementation of the whole controller in Arduino and calculating the ISR frequency, to be chosen as our Simulink sample time. The fourth sub-section explains our PID tuning strategy and the reason we chose that specific method. Sub-section 5 explains how we convert cartesian coordinates into a desired motor angle using IK, and how we convert actual motor position into cartesian coordinates using FK. Our last sub-section describes the algorithm for planning our robot's path. Section 4 covers the performance of the design.

Nomenclature

CLK	Clock
IK	Inverse Kinematics
FK	Forward Kinematics
K_p	Proportional Gain
K_i	Integral Gain
K_D	Derivative Gain

1. Design Goals and Overview

There were several customer specified design requirements, constraints, and goals for the control design of this robot arm outlined below. In addition, for the sake of further customer satisfaction and robustness of design in the face of unforeseen implementation challenges, several additional goals were specified.

Requirements:
- 3.5 DOF SCARA robot with wrist and gripper.
- 3 marshmallows on stopped conveyor.
    - Cylindrical (3cm Diameter x 3cm Tall).
    - Negligible mass & stiffness.
- 3 marshmallows discarded.

    - Grab → Move 10 cm → Drop.
- 0% overshoot before gripping.

Constraints:
- 1 Arduino Leonardo.
- Motors operate at or below nominal voltage.

Goals:
- Minimum processing time

Additional Goals:

- Easily modifiable path planning and dynamic code.
- Modularity of design, to allow for different motors.
- Transferable to the real world.

2. Control System Design

2.1 Overview

The control system design was done in MATLAB and Simulink as shown in Figure 1.

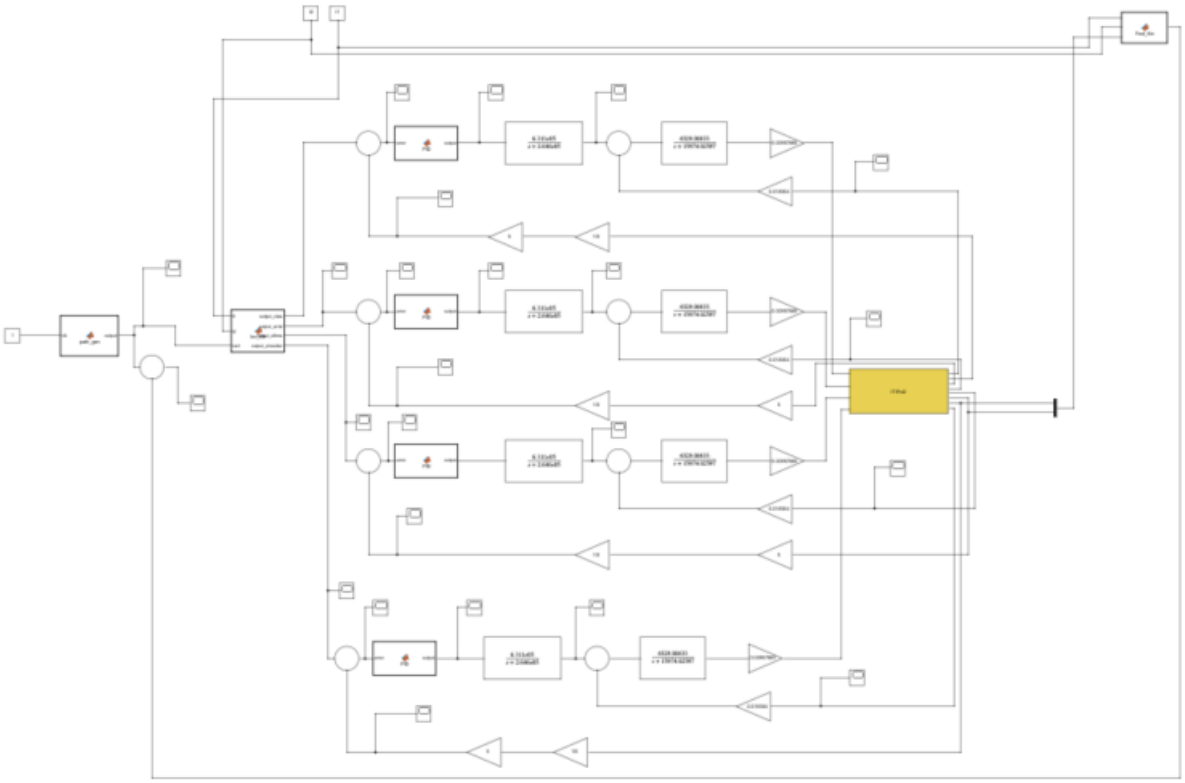

Figure 1: Full control model in Simulink

The system starts by generating an array of cartesian coordinates and a gripper state - either closed or open, cartesian coordinates define the path that the robot arm will follow. It then outputs those coordinates at a frequency of 0.01 seconds to the Inverse Kinematics MATLAB Function Block, which computes the desired angle in radians for the bicep, forearm, wrist, and the gripper. The desired angle is fed into its respective closed feedback loop, there we compute the motor torque and send it to SimX. In SimX we integrated a full SOLIDWORKS model instead of a simple model as our final iteration to reflect real life hardware more accurately. We also found the PID gains of both simulations to be similar but had to be slightly adjusted, thus it made sense to use the more realistic model.

2.2 PID Controller (MATLAB Function)

We designed our own PID controller by using a MATLAB function block. We have four different PID blocks, one for each motor, even though the motors are the same, the required proportional, integral, and derivative gains are different. Figure 2 shows an example of a PID block for the shoulder motor.

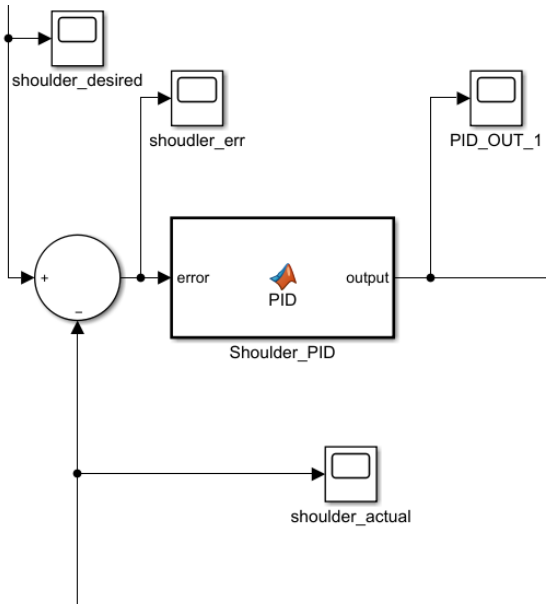

Figure 2: Shoulder PID Block

The input to the PID block is the error, which is just the desired position minus actual position. The code for calculating proportional, integral, and derivative values is shown in Appendix A.

2.2.1 PID Gain Calculations

Two of the gains were simple to calculate, for the proportional gain we just had to multiply K_p by the error, and the integral gain is just the sum of errors over time

multiplied by K_i, however the derivative gain was bit harder to find. For the derivative gain we decided place a filtering pole and implemented this by having a running sum array that favors the most recent calculation. This was accomplished by first having an array with 4 constant gains, with each term being $(e^{-Tau})^{\frac{modifier*sample\_time}{2}}$, Tau being the derivative filter constant and modifier being either 1,3,5 or 7, depending on the term. Then we to find the value that would be multiplied by the recently mentioned constant term, we had computed the slope of error vs sample time → $\frac{error-(previous\ error)}{\Delta t}$ and then multiplied it by K_d. Every value in the running sum array would be shifted to the left by 1, and then the most recent value would be multiplied by the constant term with modifier value being 1, thus having the most impact on the derivative term.

2.2.2 PID output limiter

We had to make sure that the output of our PID block stays within the limit of what our motors can handle. In our code, the sum of proportional, integral, and derivative gain was limited to either 5 V or -5 V, this resulted in our motor input voltage to be below 12 V.

2.3 Arduino C Code

The code that would replicate the behavior of all our MATLAB function blocks was written in C and compiled in Arduino IDE, the complete code can be seen in Appendix B. In the C code we implemented features such as path planning, inverse kinematics, encoder readings and PID itself. We use the encoder to get an actual joint position, which is used to calculate the error in the PID controller. We also made use of C functionality such as structs and pointers to simplify and minimize the code. Overall, our Arduino C code behaves the same way as the MATLAB function blocks, the code just had to be slightly adjusted to be compatible with C.

2.3.1 ISR Frequency

For our MATLAB simulation to be as realistic as possible, we had to compute ISR frequency of our Arduino loop. Even though the Simulink model is continuous, we can specify any discrete sample time of our MATLAB function blocks, thus we can compute and output values at a predictable time interval, which is needed for our PID calculations. However, being able to choose any discrete sample time is not realistic as we need to account for how long our code in C takes to run as if we would make the device in the real world. This was done by counting clock cycles for all our executions, we found this number to be 2823 clock cycles. Our Arduino Leonardo clock frequency is 16 MHz; thus, we can divide 2823 by 16 MHz, and we get 0.000176 seconds, this is our supposed

MATLAB sample time. However, we found that to be too fast to control and we were not able to get a zero-error steady state, thus we added a delay in the Arduino code for 823 Microseconds and brought our sample time down to 0.01 seconds.

2.4 Tuning Strategy

We used the same tuning strategy for all our joints and the steps were as follows:

1. Pick a constant destination value, for example go from [0.2 0.15] -> [0.15 0.3]
2. Make all the PID gains equal to zero.
3. Increase K_p to as large as possible while staying within our PID output limit and the error should have as close to steady oscillations as possible, see Figure 3.
    a. Note: Our PID output limit is 5 and -5, so that we stay below 12 V as an input to the motors

Figure 3: Shoulder Error, K_p=3

4. Increase K_D until the oscillations go away, so it is critically damped, see Figure 4.

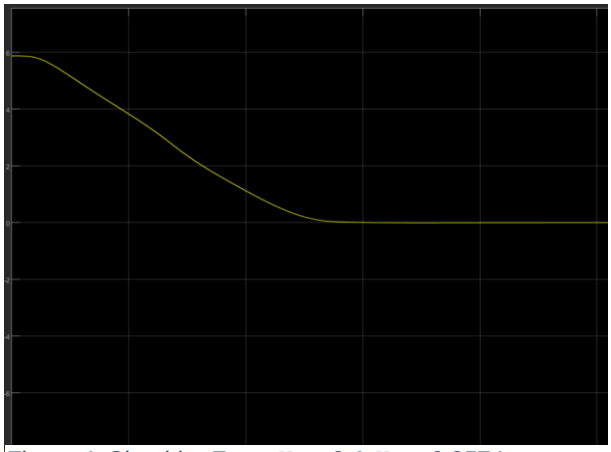

Figure 4: Shoulder Error, K_p = 3 & K_D = 0.2574

5. Look at PID output and if it is below 5V, try to increase K_p and K_D and see if we can get critically damped faster.
6. Slightly start increasing K_i until it brings us to our desired setpoint with the number of oscillations desired.
7. Figure 5 shows our final result.

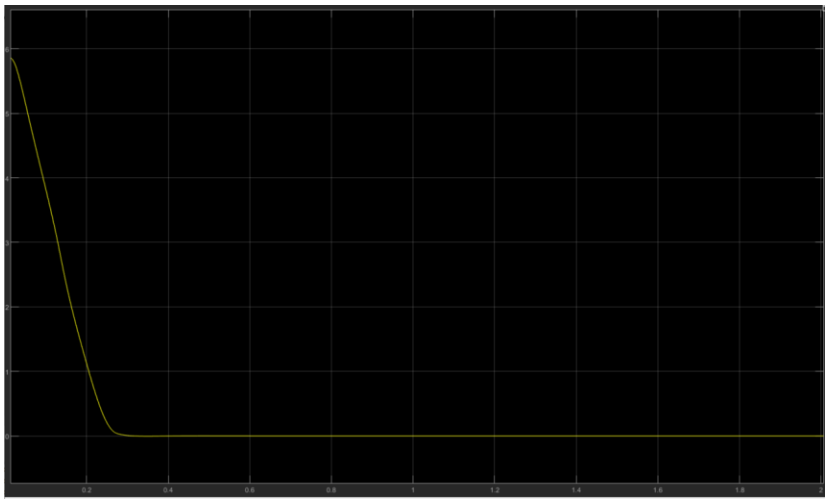

Figure 5: Shoulder tuning result

We decided to use this tunning strategy over the one introduced in class, because we found it more time efficient to co-simulate Simulink and SimX at the start of the project and tune the values according to the simple model, as we had a good idea of what the final product will look like. We also found that this method produced desirable results with less time investment. We were able to meet all the requirements and have a great system response with this method. However, if we would have used the tunning method of linearizing our model, finding our poles and zeros, performing a root locus analysis, and then tunning the gains based of those findings, we could have potentially arrived at values that proved to be more stable. During the tunning of our PID gains, we found that if derivative gain would be slightly increased, as the value approached a pole, our system would become uncontrollable and crash our simulation.

2.5 Robot Kinematics

For robot kinematics we implemented two MATLAB Function Blocks, one for inverse kinematics and one for forward kinematics.

2.5.1 Inverse Kinematics

See Figure 6 for an inverse kinematics block that receives three inputs - length of the bicep, length of the forearm, and a 1x3 array. The first two elements of the array are x and y coordinates respectively, and the last element is the status of the claw, either open or closed. IK block has 4 outputs, each being a desired

angle for the motors to go to (shoulder, elbow, wrist, and claw). For IK we must compute the desired angle for all 4 joints. See Appendix C for the MATLAB function code.

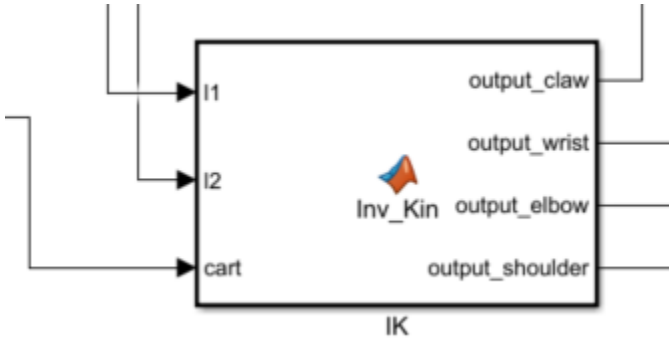

Figure 6: Inverse Kinematics MATLAB Block

2.5.1.1 Elbow

For the elbow we want to get the right handed solution, thus we take the absolute value of inverse cosine of $x^2 + y^2 - (length\ of\ forearm)^2 - (length\ of\ bicep)^2$ divided by $2 * (length\ of\ forearm * (length\ of\ bicep)$. This gives us a desired elbow angle, but we need the angle of the motor which is, due to the gearbox, 35 times less, so we multiply the result of the previous calculation by 35 to get the desired motor angle.

2.5.1.2 Shoulder

For the shoulder, the output angle is $(gamma - beta) * 35$, with gamma being inverse tan of y/x, and beta being inverse tan of $(length\ of\ forearm) * sin(angle\ of\ elbow)$ divided by $(length\ of\ bicep)\ -\ (length\ of\ forearm) * cos(angle\ of\ elbow)$.

2.5.1.3 Wrist

We wanted our wrist to always face perpendicular to the table, so that we can just move in a straight line to pick up the marshmallow. Thus the desired angle ends up being $-(angle\ of\ shoulder) - (angle\ of\ forearm)\ +\ (angle\ of\ conveyor) * 2 * \pi/360$, and then we once again multiply by 35.

2.5.1.4 Claw

The claw simply has 2 states, it is either open or closed, if the state passed into IK block is closed, then we output an angle of 0.3 radians and if the claw is supposed to be open that the output angle is -0.6 radians. We did not use a gearbox for the claw, because we found it to be way too unstable to control, and the only way to make it work with the gearbox was to make the weight over 130 grams, which was unreasonable.

2.5.2 Forward Kinematics

See Figure 7 for a forward kinematics block that receives 3 inputs - length of the bicep, length of the forearm, and a 1x2 array. The two elements of the array are bicep and forearm position, respectively. FK block has 1 output, which is a 1x2 array of x and y cartesian position of the arm. See Appendix D for the MATLAB function code.

Figure 7: Forward Kinematics MATLAB Block

2.5.2.1 Computation

To find the output cartesian coordinates, first we must divide both input angles by 35 to account for the gearbox. Then we can compute x the following way $(length\ of\ bicep) * cos(angle\ of\ bicep) + (length\ of\ elbow) * cos((angle\ of\ bicep) + (angle\ of\ elbow))$. We can find y the exact same way but replace cosine with sine. Then we output an array of x & y.

2.6 Path Planning

For path planning we implemented a MATLAB Function Block that computes the whole path of the arm and then outputs a coordinate on each positive clock edge, so every 0.001 seconds. See Appendix E for the code.
Path planning array is constructed by defining a sequence of events that needs to happen. For our path we decided to have 6 possible destinations: first garbage point, first marshmallow point, second garbage point, second marshmallow point, third garbage

point, and third marshmallow point. Each marshmallow is evenly spaced on a 30 cm long conveyor and their respective garbage point is at a straight-line distance of 10 cm. See Figure 8.

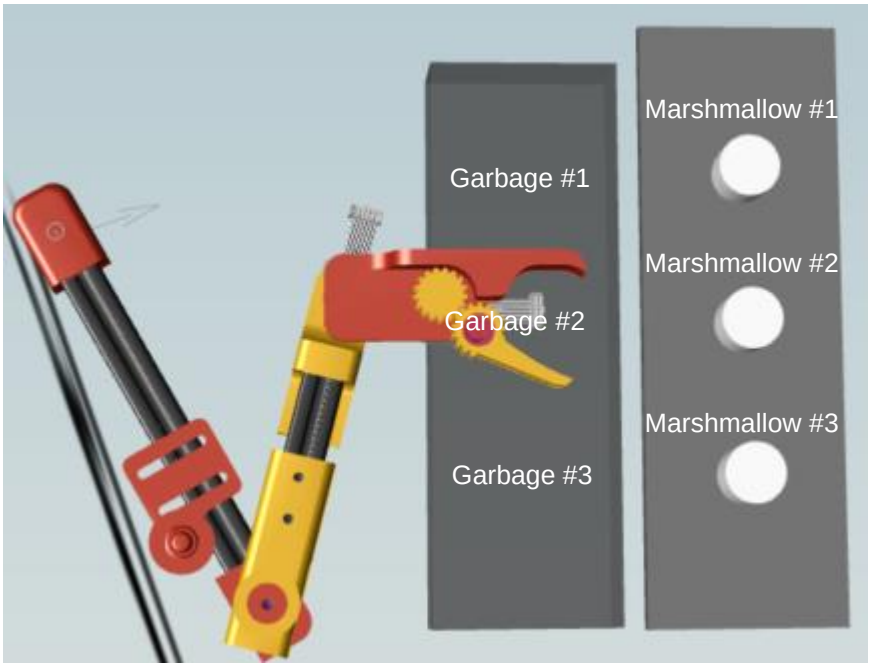

Figure 8: Robot Arm at Second Garbage Point

The path that we chose for our arm to move follows the following pattern:
Garbage #2 (starting position) → Garbage #1 → Marshmallow #1 → Garbage #1 → Garbage #2 → Marshmallow #2 → Garbage #3 → Marshmallow #3 → Garbage #3 → Garbage #2 (idle position)

2.6.1 Path Planning Algorithm

Our path planning algorithm resembles a state machine, as in we keep track of our current and next position and we iterate through these coordinates by incrementing an event counter that dictates our switch case statement. When the arm moves to any point that is not Marshmallow #1/2/3, we simply populate the path array with 1000 equally spaced points in a straight line from current to final position. This is done so that we can control the speed of the arm by increasing or decreasing the number of points in between our 2 positions. When the arm is on its way to Marshmallow #1/2/3, we slow down as we approach the marshmallow so that we have the most precision and zero overshot once we are ready to grab the marshmallow. This is done by performing a limited exponential interpolation, so that when we get closer to the marshmallow our division of current vs next point gets smaller and smaller until it hits a predefined limit. When we arrive at our destination, we wait for about 0.5 seconds before moving onto

the next task, this delay is because as we have seen from section 1.4 it takes about 0.3 seconds for the joint to not have any overshoot.

3. Performance Analysis

For easy of reference, the requirements, constraints, and goals outlined in Section 1 are repeated here.

Requirements:
- 3.5 DOF SCARA robot with wrist and gripper
- 3 marshmallows on stopped conveyor
    - Cylindrical (3cm Diameter x 3cm Tall)
    - Negligible mass & stiffness
- 3 marshmallows discarded
    - Grab → Move 10 cm → Drop
- 0% overshoot before gripping

Constraints:
- 1 Arduino Leonardo
- Motors operate at or below nominal voltage

Goals:
- Minimum processing time

Additional Goals:

- Easily modifiable, dynamic path planning code
- Modularity of design, to allow for different motors
- Transferable to the real world

Considering the controls design of the arm against them outlines, we can see that all of them are satisfactory met. In our SimX design we have 3 marshmallows stopped on a conveyor belt, with a garbage disposal point exactly 10 cm away from the respective marshmallow. We can efficiently approach every marshmallow, grab them and discard them. Figure 9 shows the error of the output from FK compared to desired position output from path planning MATLAB block, with y axis being error in m (from 0.01 to -0.01) and x axis being time in seconds. The figure shows first 3.5 seconds of the simulation, that being a path of Garbage #2 → Garbage #1 → Marshmallow #1 and close the claw to grab it.

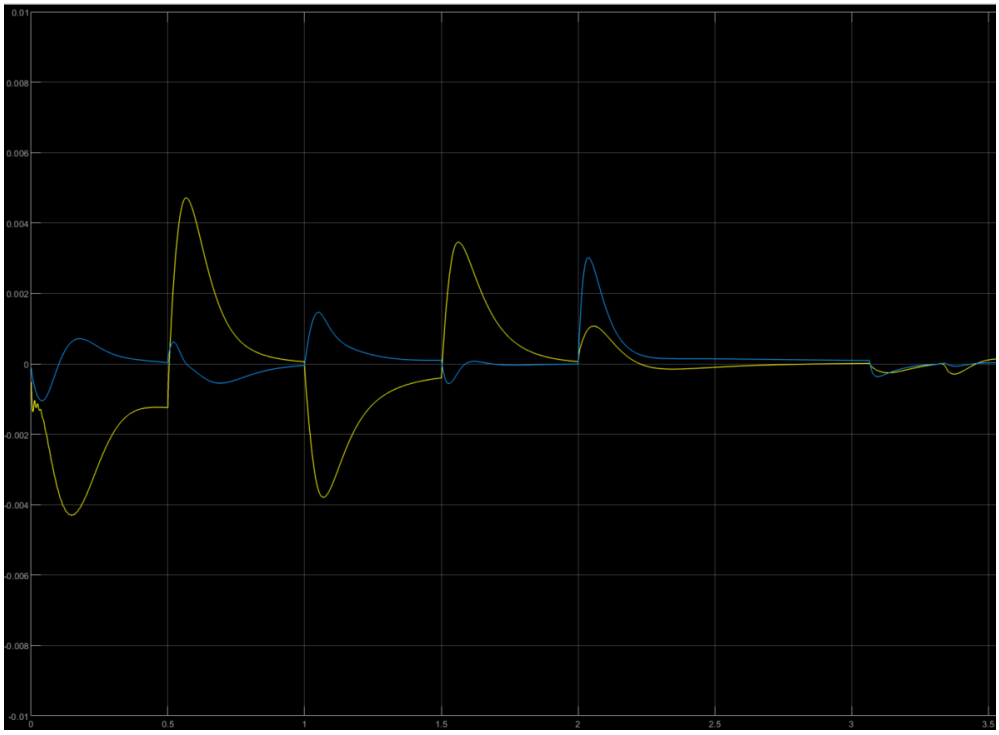

Figure 9: Cartesian error of actual vs desired position in meters

As we can see the error is at 0.005 meters or 0.5 cm at its worst during the movement from Garbage #2 → Garbage #1, but around 3 seconds when we approach the first marshmallow the error is around $1.5 * 10^{-4}$ meters or 0.15 mm, which can be considered negligible. Thus, showing that we have 0% overshoot when grabbing the marshmallow.
The design also follows all the constraints given to us, we are only using 1 Arduino Leonardo board, as seen from our 16 MHz clock frequency sued to calculate our sample time. Also, as can be seen in Appendix A, our PID code has an output limit which makes sure that our motors operate below a nominal voltage of 12 V.

Our design also achieves a goal of having a minimum processing time. This was achieved by picking the maximum gain values that would not result our PID output constantly being at the limit. Also, because our code includes a time division parameter than regulates the speed of our simulation by affecting the number of desired points in between each destination, we were able to test and come up with the most optimal number that ensured we maintained accuracy while speeding up the simulation as much as possible.
Our path planning computes the position of each marshmallow and garbage point based on the position and angle of the conveyor belt. Thus, our code was quick to modify and test when we were adjusting our conveyor location multiple times over the course of the project. Additionally, as mentioned before our PID code has an output limiter which ensures we stay below the nominal voltage, so if we decided to switch motors at the end of the project, we would just have to adjust the PID output limit and not worry about going over the new nominal voltage. Our C code is written as if we were creating this design in real life and flashing an Arduino Leonardo, such that it includes path planning, IK, FK, PID, object-like structs for each motor, and we make

use of the encoders to get our actual motor position. With these facts in mind, the additional goals are deemed to be met.

However, there is an immediate improvement that could be made to the design. As we saw in Figure 9, there is overshoot when we try to move in between garbage points, that is because we do not try to slow down as we approach the garbage point, since that was not a requirement, meaning we chose to maintain speed over accuracy. This could be fixed if we selected motors with higher nominal voltage so that we can increase the value of K_p, K_i, and K_D, this would allow us to have a snappier response and get to our steady state faster.

Appendix A – MATLAB PID Code

MATLAB code for the PID of the shoulder, the other joints are exactly the same just with different gain values.

```
function output = PID(error)
     persistent pid_integrator pid_prev_error pid_lim_min pid_lim_max
                sample_time Kp Ki Kd Tau diff_array diff_mult

     if isempty(pid_integrator)
        pid_integrator = 0;
        pid_prev_error = 0;
        pid_lim_min = -5;
        pid_lim_max = 5;
        sample_time = 0.001;
        Kp = 3;
        Ki = 0.001;
        Kd = 0.2574;
        Tau = 600;
        diff_array = zeros(1,4);
        diff_mult = [exp(-Tau)^(7*sample_time/2) exp(-Tau)^(5*sample_time/2)
                     exp(-Tau)^(3*sample_time/2) exp(-Tau)^(sample_time/2)];
    end

    %Proportional
    proportional = Kp*error;

    %Integral
    pid_integrator = pid_integrator + 0.5*Ki*sample_time*(error + pid_prev_error);

    %Derivative
    diff_array(4) = Kd*(error - pid_prev_error)/sample_time;
    pid_differentiator = 0;
    for i = 1:length(diff_array)
       pid_differentiator = pid_differentiator + (diff_array(i)*diff_mult(i));
    end

    diff_array = circshift(diff_array,[0, -1]);
    pid_prev_error = error;

    output = proportional + pid_integrator + pid_differentiator;

    if output > pid_lim_max
       output = pid_lim_max;
    elseif output < pid_lim_min
       output = pid_lim_min;
    end
end
```

Appendix B – Arduino Controller Code

Arduino code written in C for the control of all motors. This code includes a source and a header file.

Source File:

```
#include "PID_Arduino.h"

#define PI 3.14159265
#define CLK 1
#define SHOUDLER_DT 2
#define ELBOW_DT 3
#define WRIST_DT 4
#define CLAW_DT 5
#define SHOULDER_MOTOR 6
#define ELBOW_MOTOR 7
#define WRIST_MOTOR 8
#define CLAW_MOTOR 9

PID Shoulder;
PID Elbow;
PID Wrist;
PID Claw;

float bicep_length = 0.2;
float forearm_length = 0.15;
float sample_time = 0.001;
int path_points_position = 0;
int path_points[8500][3];
int currentStateCLK;
int lastStateCLK;

void setup() { // 12 Cycles
  pinMode(CLK,INPUT);
  pinMode(SHOUDLER_DT,INPUT);
  pinMode(ELBOW_DT,INPUT);
  pinMode(WRIST_DT,INPUT);
  pinMode(CLAW_DT,INPUT);
  pinMode(SHOULDER_MOTOR, OUTPUT);
  pinMode(ELBOW_MOTOR, OUTPUT);
  pinMode(WRIST_MOTOR, OUTPUT);
  pinMode(CLAW_MOTOR, OUTPUT);

  generate_path();
  float modifier;

  lastStateCLK = digitalRead(CLK);

  // Shoulder Init
  Shoulder.Kp = 3;
  Shoulder.Ki = 0.001;
  Shoulder.Kd = 0.2574;
  Shoulder.Tau = 600;
  Shoulder.limMin = -5;
  Shoulder.limMax = 5;
  Shoulder.integrator = 0;
  Shoulder.prevError = 0;
  Shoulder.encoderCounter = 0;
  modifier = 7;
  for (int i = 0; i < 4; i++) {
    Shoulder.diffMult[i] = pow(exp(-Shoulder.Tau),modifier*sample_time/2);
    modifier -= 2;
  }
```

```
  // Elbow Init
  Elbow.Kp = 2.2;
  Elbow.Ki = 0;
  Elbow.Kd = 0.11;
  Elbow.Tau = 600;
  Elbow.limMin = -5;
  Elbow.limMax = 5;
  Elbow.integrator = 0;
  Elbow.prevError = 0;
  Elbow.encoderCounter = 0;
  modifier = 7;
  for (int i = 0; i < 4; i++) {
    Elbow.diffMult[i] = pow(exp(-Elbow.Tau),modifier*sample_time/2);
    modifier -= 2;
  }

  // Wrist Init
  Wrist.Kp = 1;
  Wrist.Ki = 0.00007;
  Wrist.Kd = 0.03;
  Wrist.Tau = 600;
  Wrist.limMin = -5;
  Wrist.limMax = 5;
  Wrist.integrator = 0;
  Wrist.prevError = 0;
  Wrist.encoderCounter = 0;
  modifier = 7;
  for (int i = 0; i < 4; i++) {
    Wrist.diffMult[i] = pow(exp(-Wrist.Tau),modifier*sample_time/2);
    modifier -= 2;
  }

  // Claw Init
  Claw.Kp = 81;
  Claw.Ki = 0;
  Claw.Kd = 2;
  Claw.Tau = 600;
  Claw.limMin = -5;
  Claw.limMax = 5;
  Claw.integrator = 0;
  Claw.prevError = 0;
  Claw.encoderCounter = 0;
  modifier = 7;
  for (int i = 0; i < 4; i++) {
    Claw.diffMult[i] = pow(exp(-Claw.Tau),modifier*sample_time/2);
    modifier -= 2;
  }
}

void loop() { // 10 Cycles
  inverse_kinematics(); // 489 Cycles
  path_points_position += 1; // 4+18 = 22 Cycles

  calculate_pid(&Shoulder, SHOUDLER_DT); // 572 Cycles
  calculate_pid(&Elbow, ELBOW_DT); // 572 Cycles
  calculate_pid(&Wrist, WRIST_DT); // 572 Cycles
  calculate_pid(&Claw, CLAW_DT); // 572 Cycles

  analogWrite(Shoulder.Output, SHOULDER_MOTOR); // 6 Cycles
  analogWrite(Elbow.Output, ELBOW_MOTOR); // 6 Cycles
  analogWrite(Wrist.Output, WRIST_MOTOR); // 6 Cycles
  analogWrite(Claw.Output, CLAW_MOTOR); // 6 Cycles

  delayMicroseconds(823); // Delay by 0.823 mS to facilitate easy tuning
  // Total Number of Clock Cycles = 2823 Cycles
}
```

```
// Overall = 489 Cycles
float inverse_kinematics() {
  float x = path_points[path_points_position][0]; // 4 Cycles
  float y = path_points[path_points_position][1]; // 4 Cycles
  float claw_status = path_points[path_points_position][2]; // 4 Cycles

  // 4+10+10+10+18+10+18+10+18+20+18+18 = 164 Cycles
  float q1 = abs(acos((pow(x,2) + pow(y,2) - pow(bicep_length,2) - pow(forearm_length,2)) / (2*bicep_length*forearm_length)));
  float gamma = atan(y/x); // 4+10+20 = 34 Cycles
  float beta = atan((forearm_length*sin(q1))/(bicep_length + forearm_length*cos(q1))); // 4+10+18+10+20+18+18+10 = 108 Cycles
  float q0 = abs(gamma - beta); // 4+10+18 = 32 Cycles

  // 11 + 4 + 4 = 19 Cycles
  if (claw_status == 0) {
    Claw.desiredValue = -0.6;
  }
  else {
    Claw.desiredValue = 0.3;
  }
  Shoulder.desiredValue = 35 * q0; // 4+18 = 22 Cycles
  Elbow.desiredValue = 35 * q1; // 4+18 = 22 Cycles
  Shoulder.desiredValue = 35 * (-0.8726 + q0 + q1); // 4+18+18+18+18 = 76 Cycles
}

// Overall = 572 Cycles
void calculate_pid(PID *pid, int encoderPin) {
  float actual_value = get_encoder_value(pid, encoderPin); // 4 + 124 = 128 Cycles
  float error = pid->desiredValue - actual_value; // 4 + 18 = 22 Cycles

  // Proportional
  float proportional = pid->Kp*error; // 4 + 18 = 22 Cycles

  // Integral
  pid->integrator = pid->integrator + 0.5*pid->Ki*sample_time*(error + pid->prevError); // 4 + 18 + 18 + 18 + 18 = 76 Cycles

  pid->diffArray[3] = pid->Kd*(error - pid->prevError)/sample_time; // 4 + 18 + 18 + 20 = 60 Cycles
  float pid_differentiator = 0; // 4 Cycles

  // 4 * (4 + 5 + 18 + 18) = 180 Cycles
  for (int i=0; i < 4; i++){
      pid_differentiator = pid_differentiator + (pid->diffArray[i]*pid->diffMult[i]);
  }

  // 4 * (5 + 4) = 36 cycles
  for (int i=0; i < 4; i++){
      pid->diffArray[i]=pid->diffArray[i+1];
  }

  pid->prevError = error; // 4 Cycles
  pid->Output = proportional + pid->integrator + pid_differentiator; // 4+18+18 = 40 Cycles
}
```

```
// Overall = 124 Cycles
float get_encoder_value(PID *pid, int pin){
  currentStateCLK = digitalRead(CLK); // 4+8 = 12 Cycles

  // 11+11+4+18+4+18 = 66 Cycles
  if (currentStateCLK != lastStateCLK  && currentStateCLK == 1){
    if (digitalRead(pin) != currentStateCLK) {
      pid->encoderCounter--;
    } else {
      pid->encoderCounter++;
    }
  }

  // Remember last CLK state
  lastStateCLK = currentStateCLK; // 4 Cycles

  return pid->encoderCounter * (360/1024); // 4+18+20 = 42 Cycles
}

void generate_path() {
  const int t = 1000;
  const int wait_length = 750;
  const int claw_open = 0;
  const int claw_close = 1;
  int claw_old_state = claw_open;
  const int idle = 0;
  const int first_marsh = 1;
  const int first_marsh_idle = 2;
  const int second_marsh = 3;
  const int second_marsh_idle = 4;
  const int third_marsh = 5;
  const int third_marsh_idle = 6;
  const int garbage = 7;
  float table_length = 0.3;
  int table_angle = 50;
  int event_count = 1;
  int count = 1;
  int array_position = 0;
  float x_value_increment = 0.005;
  float table_origin[2] = {0.15, 0.3};
  float gripper_length = 0.098;
  float gripper_adjustment = 0.01;
  float degrees_to_radians = PI/180;

  float current_point[2];
  float next_point[2];
  float starting_point[2] = {0.2, 0.15};
  float first_marsh_point[2] = {(table_origin[0]-table_length/4*sin(table_angle*degrees_to_radians))
                                -gripper_adjustment*sin(table_angle*degrees_to_radians)
                                -gripper_length*sin((90-table_angle)*degrees_to_radians),
                                (table_origin[1]+table_length/4*cos(table_angle*degrees_to_radians))
                                +gripper_adjustment*cos(table_angle*degrees_to_radians)
                                -gripper_length*cos((90-table_angle)*degrees_to_radians)};
  float second_marsh_point[2] = {table_origin[0]-gripper_adjustment*sin(table_angle*degrees_to_radians)
                                -gripper_length*sin((90-table_angle)*degrees_to_radians),
                                 table_origin[1]+gripper_adjustment*cos(table_angle*degrees_to_radians)
                                -gripper_length*cos((90-table_angle)*degrees_to_radians)};
```

```
float third_marsh_point[2] = {(table_origin[0]+table_length/4*sin(table_angle*degrees_to_radians))
                              -gripper_adjustment*sin(table_angle*degrees_to_radians)
                              -gripper_length*sin((90-table_angle)*degrees_to_radians),
                              (table_origin[1]-table_length/4*cos(table_angle*degrees_to_radians))
                              +gripper_adjustment*cos(table_angle*degrees_to_radians)
                              -gripper_length*cos((90-table_angle)*degrees_to_radians)};
float first_garbage_point[2] = {first_marsh_point[0]-0.1*sin((90-table_angle)*degrees_to_radians),
                                first_marsh_point[1]-0.1*cos((90-table_angle)*degrees_to_radians)};
float second_garbage_point[2] = {second_marsh_point[0]-0.1*sin((90-table_angle)*degrees_to_radians),
                                 second_marsh_point[1]-0.1*cos((90-table_angle)*degrees_to_radians)};
float third_garbage_point[2] = {third_marsh_point[0]-0.1*sin((90-table_angle)*degrees_to_radians),
                                third_marsh_point[1]-0.1*cos((90-table_angle)*degrees_to_radians)};

int path_vector_length = 11;
int path_vector[path_vector_length] = {garbage, first_marsh_idle, first_marsh, garbage, second_marsh_idle,
                                       second_marsh, garbage, third_marsh_idle, third_marsh, garbage, idle};

while (event_count <= path_vector_length) {
    int event = path_vector[event_count];
    float x_value = 8;

    switch (event){
        case garbage:
            if (event_count == 1){
                current_point[0] = starting_point[0];
                current_point[1] = starting_point[1];
                next_point[0] = second_garbage_point[0];
                next_point[1] = second_garbage_point[1];
            }
            else if (path_vector[event_count - 1] == first_marsh){
                current_point[0] = first_marsh_point[0];
                current_point[1] = first_marsh_point[1];
                next_point[0] = first_garbage_point[0];
                next_point[1] = first_garbage_point[1];
            }
            else if (path_vector[event_count - 1] == second_marsh){
                current_point[0] = second_marsh_point[0];
                current_point[1] = second_marsh_point[1];
                next_point[0] = second_garbage_point[0];
                next_point[1] = second_garbage_point[1];
            }
            else {
                current_point[0] = third_marsh_point[0];
                current_point[1] = third_marsh_point[1];
                next_point[0] = third_garbage_point[0];
                next_point[1] = third_garbage_point[1];
            }
        case first_marsh_idle:
            current_point[0] = second_garbage_point[0];
            current_point[1] = second_garbage_point[1];
            next_point[0] = first_garbage_point[0];
            next_point[1] = first_garbage_point[1];
        case first_marsh:
            current_point[0] = first_garbage_point[0];
            current_point[1] = first_garbage_point[1];
            next_point[0] = first_marsh_point[0];
            next_point[1] = first_marsh_point[1];
        case second_marsh_idle:
            current_point[0] = first_garbage_point[0];
            current_point[1] = first_garbage_point[1];
            next_point[0] = second_garbage_point[0];
            next_point[1] = second_garbage_point[1];
        case second_marsh:
            current_point[0] = second_garbage_point[0];
            current_point[1] = second_garbage_point[1];
            next_point[0] = second_marsh_point[0];
            next_point[1] = second_marsh_point[1];
```

```
            case third_marsh_idle:
                current_point[0] = second_garbage_point[0];
                current_point[1] = second_garbage_point[1];
                next_point[0] = third_garbage_point[0];
                next_point[1] = third_garbage_point[1];
            case third_marsh:
                current_point[0] = third_garbage_point[0];
                current_point[1] = third_garbage_point[1];
                next_point[0] = third_marsh_point[0];
                next_point[1] = third_marsh_point[1];
            case idle:
                current_point[0] = third_garbage_point[0];
                current_point[1] = third_garbage_point[1];
                next_point[0] = second_garbage_point[0];
                next_point[1] = second_garbage_point[1];
        }

        float x_point = current_point[0];
        float y_point = current_point[1];
        float increment_x;
        float increment_y;

        while ((abs(x_point - current_point[0]) <= abs(next_point[0] - current_point[0])) &&
               (abs(y_point - current_point[1]) <= abs(next_point[1] - current_point[1]))) {
            if (event == first_marsh || event == second_marsh || event == third_marsh){
                increment_x = (next_point[0] - current_point[0]) / (1*pow(x_value,3));
                increment_y = (next_point[1] - current_point[1]) / (1*pow(x_value,3));
                if (1*pow(x_value,3) < 5000){
                    x_value = x_value + x_value_increment;
                }
            }
            else {
                increment_x = (next_point[0] - current_point[0]) / t;
                increment_y = (next_point[1] - current_point[1]) / t;
            }
            x_point = x_point + increment_x;
            y_point = y_point + increment_y;
            path_points[array_position][0] = x_point;
            path_points[array_position][1] = y_point;
            path_points[array_position][2] = claw_old_state;
            array_position = array_position + 1;
        }

        for (int i = 0; i < wait_length; i++){
          if ((event == first_marsh || event == second_marsh || event == third_marsh) && (i > wait_length/2)){
            path_points[array_position][0] = x_point;
            path_points[array_position][1] = y_point;
            path_points[array_position][2] = claw_close;
            claw_old_state = claw_close;
          }
          else if ((event == garbage) && (i > wait_length/2)) {
            path_points[array_position][0] = x_point;
            path_points[array_position][1] = y_point;
            path_points[array_position][2] = claw_open;
            claw_old_state = claw_open;
          }
          else {
            path_points[array_position][0] = x_point;
            path_points[array_position][1] = y_point;
            path_points[array_position][2] = claw_old_state;
          }

          array_position = array_position + 1;
        }

        event_count = event_count +1;
    }
}
```

Header File:

```
#ifndef PID_CONTROLLER_H
#define PID_CONTROLLER_H

typedef struct {
  float Kp;
  float Ki;
  float Kd;

  float Tau;

  float limMin;
  float limMax;

  float diffMult[4];
  float diffArray[4];
  float integrator;
  float prevError;

  float desiredValue;

  float encoderCounter;

  float Output;
} PID;

#endif
```

Appendix C – MATLAB IK Code

MATLAB code for the IK

```
% Inv Kinematics
function [output_claw,output_wrist,output_elbow,output_shoulder] = Inv_Kin(l1, l2,
cart)
    x = cart(1); % Task-space x coord
    y = cart(2); % Task-space y coord

    % Get q1 first, need it to calculate q0. We will always
    % take the "right-handed" solution where q1 is (+).
    q1 = abs(acos((x^2 + y^2 - l1^2 - l2^2)/(2*l1*l2)));

    % Get q0
    gamma = atan(y/x);
    beta = atan((l2*sin(q1))/(l1 + l2*cos(q1)));

    q0 = gamma - beta;

    % Ouput our angle vector
    if cart(3) == 0
        output_claw = -0.6;
    else
        output_claw = 0.3;
    end
    output_wrist = 35 * (-q0 - q1 + 70*(2*3.1415/360));
    output_elbow = 35 * q1;
    output_shoulder = 35 * q0;

end
```

Appendix D – MATLAB FK Code

MATLAB code for the FK

```
% Fwd Kinematics
function output = Fwd_Kin(l1, l2, angles)

    q0 = angles(1) / 35; % Configuration-space shoulder angle
    q1 = angles(2) / 35; % Configuration-space elbow angle

    % Get task-space coordinates
    x = l1*cos(q0) + l2*cos(q0+q1);
    y = l1*sin(q0) + l2*sin(q0+q1);

    % Output the task-space cartesian coordinats
    output = [x;y];

end
```

Appendix C – MATLAB Path Planning Code

MATLAB code for path planning, the output of path_gen is every 0.001 seconds.

```
function output  = path_gen(clk)
    persistent count path_points path_size

    if isempty(count)
        count = 1;
        path_points = generate_path();
        path_size = length(path_points);
    end

    if clk == 1
        if count ~= path_size
            count = count + 1;
        end
    end

    output = path_points(count,:);
end

function output = generate_path()
    t = (0:0.1:50)'; % Time
    wait_length = 500;
    claw_open = 0;
    claw_close = 1;
    claw_old_state = claw_open;
    idle = 0;
    first_marsh = 1;
    first_marsh_idle = 2;
    second_marsh = 3;
    second_marsh_idle = 4;
    third_marsh = 5;
    third_marsh_idle = 6;
    garbage = 7;
    table_length = 0.3;
    table_angle = 70;
    table_origin = [0.15 0.30];
    gripper_len = 0.098;
    gripper_adjustment = 0.01;
```

```
current_point = [0 0]; %Initialize
next_point = [0 0]; %Initialize
starting_point = [0.2 0.15];
first_marsh_point = [table_origin(1)-table_length/4*sind(table_angle)-...
                     gripper_adjustment*sind(table_angle)-...
                     gripper_len*sind(90-table_angle)...
                     table_origin(2)+table_length/4*cosd(table_angle)+...
                     gripper_adjustment*cosd(table_angle)-...
                     gripper_len*cosd(90-table_angle)];
second_marsh_point = [table_origin(1)-gripper_adjustment*sind(table_angle)-...
                      gripper_len*sind(90-table_angle)...
                      table_origin(2)+gripper_adjustment*cosd(table_angle)-...
                      gripper_len*cosd(90-table_angle)];
third_marsh_point = [table_origin(1)+table_length/4*sind(table_angle)-...
                     gripper_adjustment*sind(table_angle)-...
                     gripper_len*sind(90-table_angle)...
                     table_origin(2)-table_length/4*cosd(table_angle)+...
                     gripper_adjustment*cosd(table_angle)-...
                     gripper_len*cosd(90-table_angle)];
first_garbage_point = [first_marsh_point(1)-0.1*sind(90-table_angle)...
                       first_marsh_point(2)-0.1*cosd(90-table_angle)];
second_garbage_point = [second_marsh_point(1)-0.1*sind(90-table_angle)...
                        second_marsh_point(2)-0.1*cosd(90-table_angle)];
third_garbage_point = [third_marsh_point(1)-0.1*sind(90-table_angle)...
                       third_marsh_point(2)-0.1*cosd(90-table_angle)];

path_vector = [garbage first_marsh_idle first_marsh garbage second_marsh_idle...
               second_marsh garbage third_marsh_idle third_marsh garbage idle];
path_points = zeros(100000,3);

event_count = 1;
array_position = 1;
x_value_increment = 0.005;

while event_count <= length(path_vector)
    event = path_vector(event_count);
    x_value = 8;

    switch event
        case garbage
            if event_count == 1
                current_point = starting_point;
                next_point = second_garbage_point;
            elseif path_vector(event_count - 1) == first_marsh
                current_point = first_marsh_point;
                next_point = first_garbage_point;
            elseif path_vector(event_count - 1) == second_marsh
                current_point = second_marsh_point;
                next_point = second_garbage_point;
            else
                current_point = third_marsh_point;
                next_point = third_garbage_point;
            end
        case first_marsh_idle
            current_point = second_garbage_point;
            next_point = first_garbage_point;
        case first_marsh
            current_point = first_garbage_point;
            next_point = first_marsh_point;
        case second_marsh_idle
            current_point = first_garbage_point;
            next_point = second_garbage_point;
```

```
            case second_marsh
                current_point = second_garbage_point;
                next_point = second_marsh_point;
            case third_marsh_idle
                current_point = second_garbage_point;
                next_point = third_garbage_point;
            case third_marsh
                current_point = third_garbage_point;
                next_point = third_marsh_point;
            case idle
                current_point = third_garbage_point;
                next_point = second_garbage_point;
        end

        x_point = current_point(1);
        y_point = current_point(2);


        while ((abs(x_point - current_point(1)) <= abs(next_point(1) - current_point(1))) &&...
               (abs(y_point - current_point(2)) <= abs(next_point(2) - current_point(2))))
            if (event == first_marsh || event == second_marsh || event == third_marsh)
                increment_x = (next_point(1) - current_point(1)) / (1*(x_value^3));
                increment_y = (next_point(2) - current_point(2)) / (1*(x_value^3));
                if (1*(x_value^3) < 5000)
                    x_value = x_value + x_value_increment;
                end
            else
                increment_x = (next_point(1) - current_point(1)) / length(t);
                increment_y = (next_point(2) - current_point(2)) / length(t);
            end
            x_point = x_point + increment_x;
            y_point = y_point + increment_y;
            path_points(array_position,:) = [x_point y_point claw_old_state];
            array_position = array_position + 1;
        end

        for i = 1:wait_length
            if (event == first_marsh || event == second_marsh || event == third_marsh) &&...
                (i > wait_length/2)
                path_points(array_position,:) = [x_point y_point claw_close];
                claw_old_state = claw_close;
            elseif (event == garbage) && (i > wait_length/2)
                path_points(array_position,:) = [x_point y_point claw_open];
                claw_old_state = claw_open;
            else
                path_points(array_position,:) = [x_point y_point claw_old_state];
            end
            array_position = array_position + 1;
        end

        event_count = event_count +1;
    end

    path_points(array_position:end, :) = [];
    output = path_points;
end
```