# SCARA Robot Control System

**White-Paper** **Updated: April 2, 2022**

Jinil Patel, Group G3-250, ECE, University of British Columbia, Vancouver, BC, Canada

**Abstract**

The SCARA robotic arm is controlled using 2 tuned PID controller and 1 P controller is used to achieve 3 degrees of freedom. Motor data from Maxon motors and linearized electrical circuits imported from Multisim were used to model and simulate the control system in Simulink. In addition, SimulationX model was paired with Simulink (referred as Co-Sim) to simulate physical model of the SCARA arm.

Section 1 of this report describes the designing of PID controller in MATLAB and calculation of control frequency of STM 32 microcontroller. Section 2 describes tuning of PID controller. Section 3 describes the robot kinematics. Section 4 describes the trajectory planning to pick up and drop the object at the desired location and within the desired time set by the user. Section 5 describes homing sequence. Section 6 describes the sensor logic and Section 7 integrates all the sub parts to form control system for SCARA arm.

**Nomenclature:**

PID: Proportional, Integral, and Derivative

FIR: Finite Input Response

CF: Control Frequencys

FPGA: Field Programmable Gate Array

Motor 1: Shoulder motor

Motor 2: Elbow motor

**Requirements:**

- Rise time < 0.3 seconds
- Overshoot < 10%
- Steady-Statete error < 1%
- User input position and speed
- Automatic Path Generation
- Homing Sequence

**Constraints:**

- Movement of the second arm should be limited between $\frac{-\pi}{2}$ to $\frac{\pi}{2}$

## 1. PID Controller

The PID controller has three parts: Proportional (P), Integral (I), Derivative (D).

The P part of the PID controller is calculated as:

$$Propotional = Error$$

The I part of the PID controller is sum of error over time and represents accumulated error that has been corrected.

$$Integral = Integral + \frac{Error + Previous\ Error}{2}$$

The D part of the PID Controller requires implementing FIR Filter to limit the high frequency gain and noise resulting in following equation:

$$Derivative = \sum_{i=1}^{sample\ points} Scaling * p * e^{-p*i*dt} * \frac{new\ error - previous\ error}{dt}$$

Where Scaling is normalization for filter coefficients, $p * e^{-p*i*dt}$ is time domain representation of low pass filter with pole p = 2*CF which is convoluted with the derivative.

Output of the PID controller is given by:

$$Output = Kp * Propotional + Ki * Integral + Kd * Derivative$$

The figure below compares the user-generated PID code's step response with Simulink PID block.

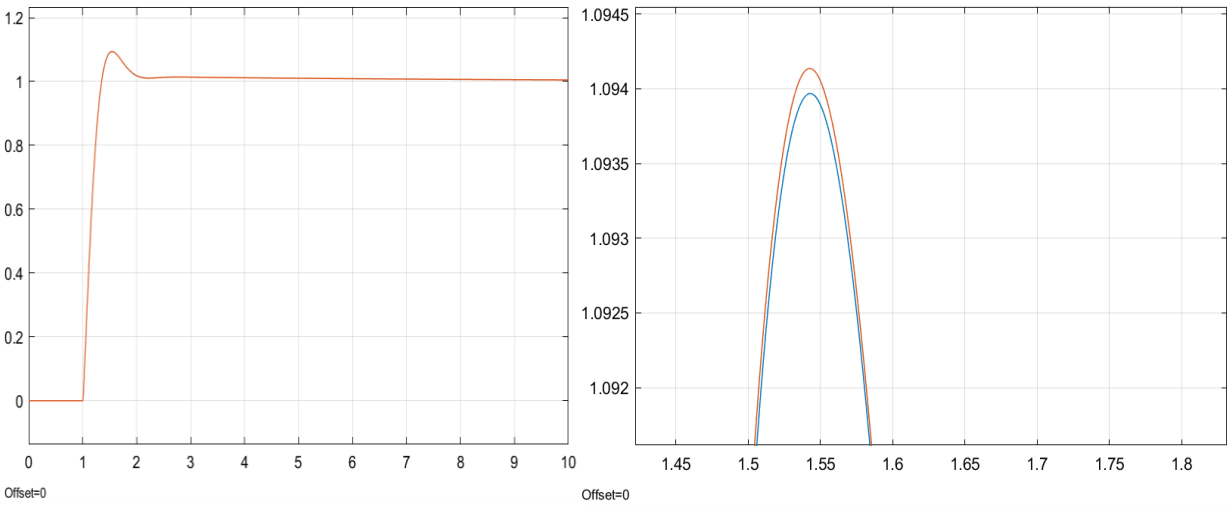



Figure 1: Comparison between Simulink PID Block (Blue Curve) and User generated PID Block (Red Curve). Figure on the right is zoomed in version for differentiating between the two PID's.

The Matlab code was then translated into C code and Control Frequency was calculated from the look up table based upon the number of clock cycles for command execution. Please refer to appendix for both MATLAB and C code.

One time execution of C code takes 730 clock cycles.

Total cycles = 730*2 = 1460

ISR rate = $\frac{STM\ 32\ Clock\ Frequency}{Total\ Cycles} = 72 * \frac{10^6}{1460} =$ 49,315Hz

ISR rate used is 24,658 Hz or $T_s$ = 40.5 µs to give code roughly double the time then it requires.

## 2. PID Tuning

Each motor was tuned separately using 10 step process. Once the estimated gain values were obtained heuristic tuning was used to get the desired response from the motors. Rise time and overshoot was minimized while keeping in check that steady state error does not exceed 1%. The tuning was done by using step function as smooth curves are not ideal for mimicking sudden input signal changes. After achieving desired response both motors were individually tuned using Co-sim and finally integrated together to simulate the physical model of SCARA arm.

Figure below shows Simulink model used for tuning the motors. Both motors had similar models with different parameters.

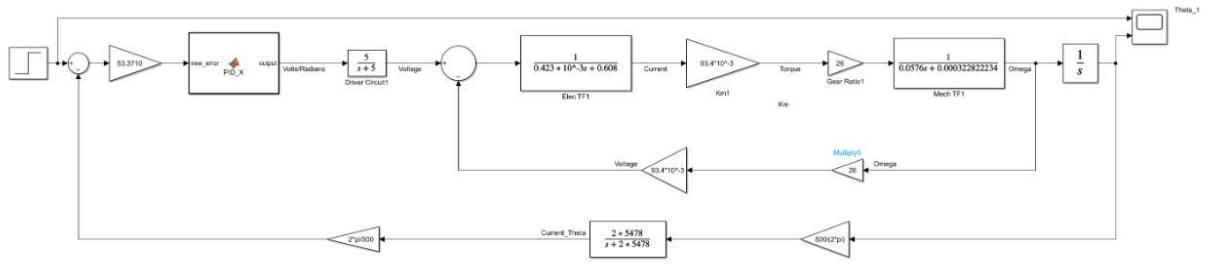



Figure 2: Simulink model for tuning the motor.

| | K | $K_p$ | $K_i$ | $K_d$ |
|---|---|---|---|---|
| **Simulink** | 53.3710 | 0.3669 | 0.05 | 0.0336 |
| **Co-Sim** | 0.3 | 1.1669 | 0.00005 | 0.3936 |

Table 1: Motor 1 Gain values

| | K | $K_p$ | $K_i$ | $K_d$ |
|---|---|---|---|---|
| **Simulink** | 53 | 0.1669 | 0.29 | 0.336 |
| **Co-Sim** | 0.3 | 1.1669 | 0.0005 | 0.03936 |

Table 2: Motor 2 Gain values

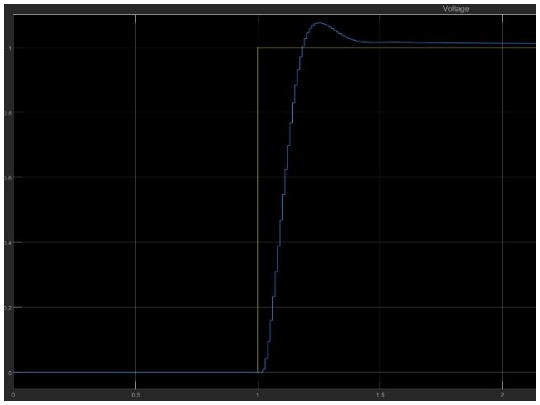


Figure 3: Co-Sim Tuned response of motor 1

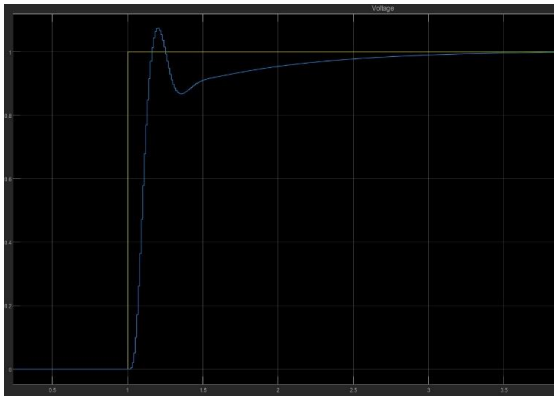


Figure 4: Co-Sim Tuned response of motor 2

## 3. Robot Kinematics

Hand calculations were done to generate inverse kinematics and direct kinematics block. The direct kinematics block and inverse kinematics block were tested by cascading together and checking the output in the scope. Figure 5 shows the test setup and results. Please refer to appendix for MATLAB code and hand calculation for direct and inverse kinematics.

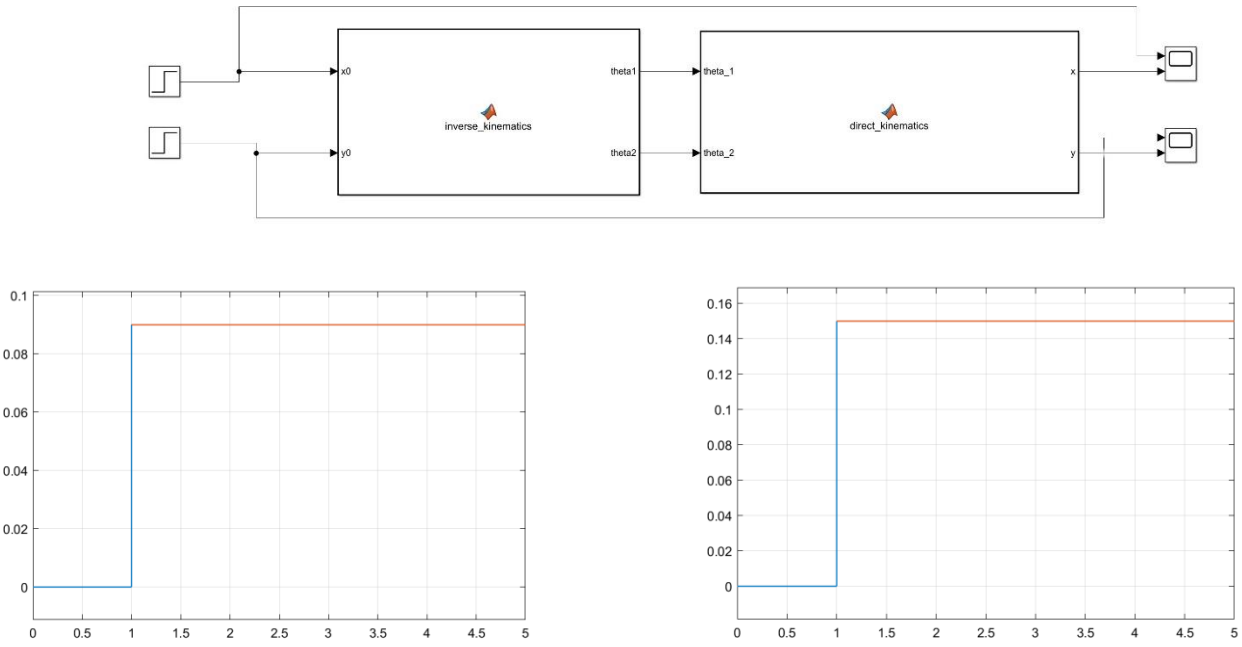



Figure 5: Output of direct kinematics block (red), desired output (blue)

## 4. Trajectory planning and Homing Sequence

The trajectory planning sub-system takes the coordinates of position 1, position 2 and time in which user wants to go from position 1 to position 2. Inverse kinematics is performed on the coordinates of the two position to get the desired angle. The path_planning block generates the cubic function for path by computing coefficients of following equations:

$$Path = a * t^3 + b * t^2 + c * t + d$$

Differentiating above equation gives

$$Velocity = 3 * a * t^2 + 2 * b * t + c$$

Know variables are Position 1, Position 2, and initial velocity = final velocity = 0 m/s

Using above information path can be calculated to go from position 1 to position 2. Figure 5 shows the trajectory planning sub system, figure 6 and 7 shows tuned motors tracing the path generated by the trajectory planning sub system.

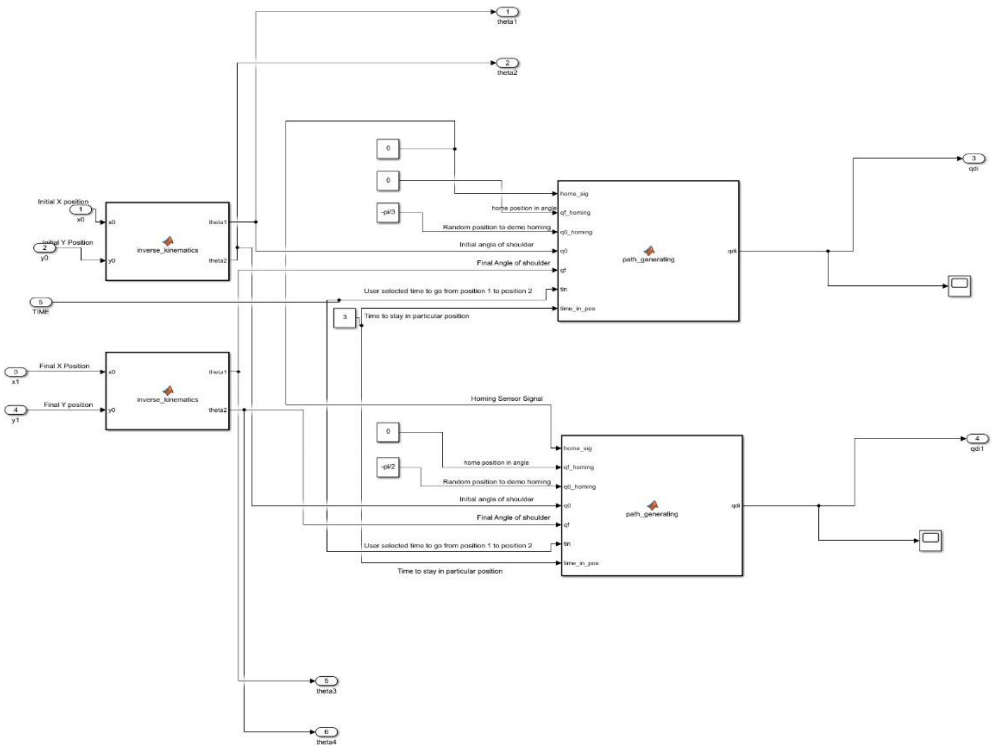



Figure 6: Trajectory Planning Sub-System

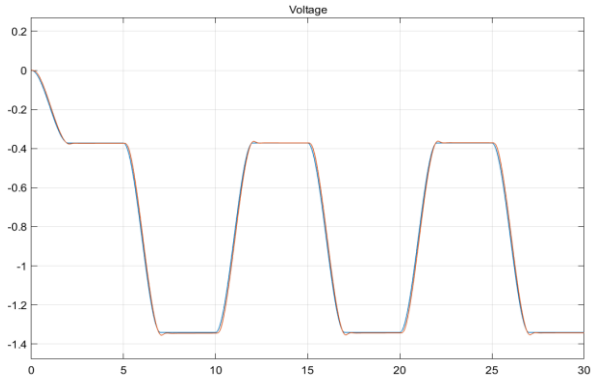



Figure 6: Trajectory Planning Sub-System generated path for motor 1 (blue). Output of the controller (red) for motor 1

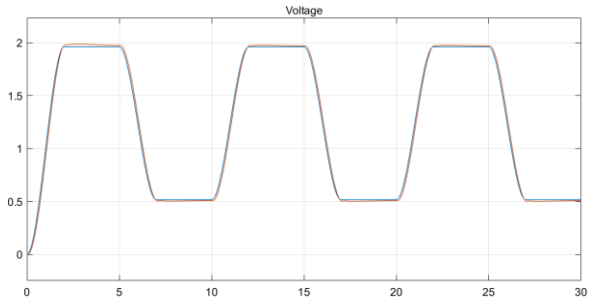



Figure 7: Trajectory Planning Sub-System generated path for motor 2 (blue). Output of the controller (red) for motor 2

Please refer to MATLAB code path_planning in appendix for homing logic implementation

## 5. Homing Sequence

Homing sensor signal is set high when the motor 1 is at 0 and motor 2 is at $\frac{\pi}{2}$ (homing position). If the robot is powered on in non-homing position the motor 1 starts to sweep -π radians and when it reaches home, homing sensor signal becomes high which is sent to FPGA to reset the counter. Motor 2 starts to sweep -π radians as well, because home is at $\frac{\pi}{2}$ range of motion for motor two is between $\frac{-\pi}{2}$ to $\frac{\pi}{2}$. To demonstrate homing logic, in figure 5, I am manually setting homing sensor signal to 0 which indicates we are not at home. I am also manually entering q0_homing position which is random starting point when SCARA is powered on. This results in sudden jerk at 0 seconds as simulation does not allow to start from non-zero position. The two signals q0_homing and qf_homing will be discarded in microcontroller implementation as they are just for demoing. It can be observed from figure 8 that when SCARA is powered on from non-zero position it first homes itself and then follows the path generated by path_generating.

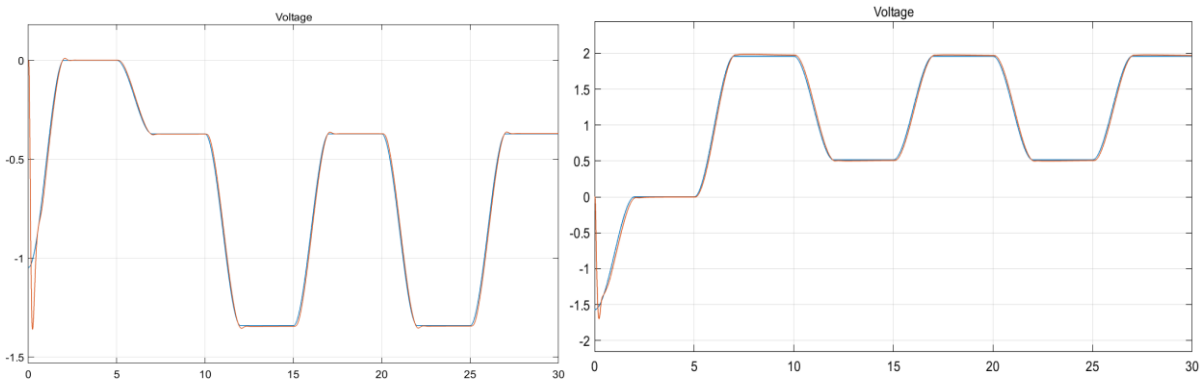



Figure 8: Homing Logic for motor 1 (blue curve). Output of controller (red) on left. Homing logic for motor 2 on right

Please refer to MATLAB code path_planning in appendix for homing logic implementation

## 6. Sensor Logic

For the sensor we have used FPGA to keep track of motor position. In the feedback look we have sensor resolution of 500/(2*π). The signal is passed through sensor filter to eliminate the noise. The C function get_angle() receives the count value from the shift register used in FPGA. The value of count is converted to decimal number and then actual angle is calculated using following formulas:

$$Actual\ Angle\ for\ motor\ 1 = (500 - count) * \frac{2\pi}{500}$$

$$Actual\ Angle\ for\ motor\ 2 = \frac{\pi}{2} + (500 - sum) * \frac{2\pi}{500}$$

The (500 – count) term in the equations is a result of implementation of quadrature decoder. Inside quadrature decoder 500 corresponds to 0 degrees to deal with negative numbers in Verilog. The homing position for second motor is at $90^0$ which is accounted for in second equation. Please refer to appendix for C code.

## 7. SCARA robot control system

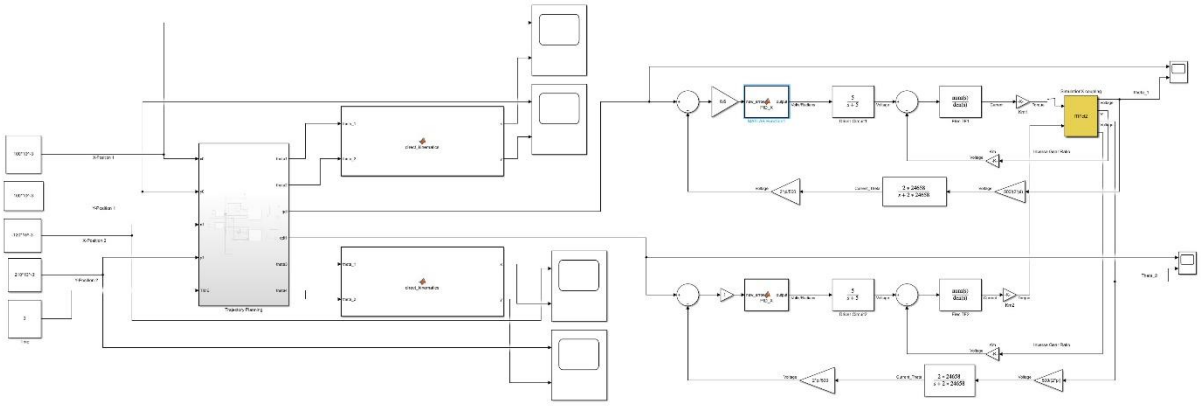


Figure 9: SCARA Robot Control System without bonus part

# Appendix

## 1. PID MATLAB Code

```
function output = PID_X(new_error)

persistent previous_error
persistent integral_term
persistent derivative
persistent convolution_filter

CF = 24658;
p = 2*CF;
dt = 1/CF;
K = 4.6666;
Ki = 0.008;
Kd = 20.8;
sample_points = 8;

if isempty(previous_error) % initialize all the variables
    previous_error = 0;
    integral_term = 0;
    derivative = zeros(1,sample_points);
    convolution_filter = zeros(1,sample_points);
    L_norm = 0;

    for i = 1:sample_points
        L_norm = L_norm + p*exp(-p*i*dt); % get the denominator to perform L1 Normalization to handle overflow
    end

    scaling = 1/L_norm; % get scaling factor for L1 Norm


    for i = 1:sample_points
        convolution_filter(i) = scaling*p*exp(-p*i*dt); % get real time impulse response for low pass filter
    end
end

derivative(2:sample_points) = derivative(1:sample_points-1); % store derivative in reverse order to perform convolution and that way we can remove
derivative(1) = (new_error - previous_error)/dt; % calculate the derivative

derivative_term = dot(derivative,convolution_filter); % convolute derivative signal with filter impulse response

integral_term = integral_term + dt*(previous_error + new_error)/2; % compute integral

propotional_term = new_error; % proptotional
previous_error = new_error; % new error is stored in previous error for future computation
output = K*propotional_term + Ki*integral_term + Kd*derivative_term; % output of the PID controller
```

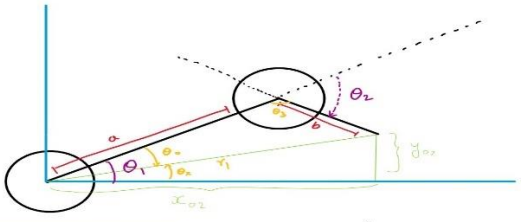
## 2. C code for PID and Sensor Logic

```
void setup() //initialize all the variables
{
    previous_error = 0;
    integral_term = 0;
    derivative_term = 0;
    for(i = 0; i<8; i++)
    {
        derivative[i] = 0;
    }

    for(i = 0; i<8; i++) //get the denominator to form L1 Norm to handle overflow
    {
        L_norm = L_norm + p*exp(-p*i*dt);
    }

    scaling = 1/L_norm; //get scaling factor for L1 Norm

    for(i = 0; i<8; i++)
    {
        convolution_filter[i] = scaling*p*exp(-p*i*dt); //Digital low pass filter in time
domain
    }
}
```

```
23
24 void loop() //PID Code
25 {
26     float current_angle;
27     desired angle = analogRead(1);
28     current_angle = get_angle(); //read from Pin 1
29     new error = desired_angle - current_angle;
30
31     for(i = 7; i>7; i--)
32     {
33         derivative[i] = derivative[i-1]; // store derivative in reverse order to perform
convolution and that way we can remove the oldest sample first
34     }
35
36    derivative[0] = (new_error - previous_error)/dt; // Compute Derivative
37
38    for(i = 0; i < 8; i++)
39    {
40        derivative_term = derivative_term + derivative[i]*convolution_filter[i]; //Convolute
derivative signal with impulse response of low pass filter
41    }
42
43    integral_term = integral_term + dt*(new_error + previous_error)/2; // Compute integral
44
45    propotional_term = new_error; //compute propotional term
46
47    previous_error = new_error; //new error is stored in previous error for future computation
48
49    output = K*propotional_term + Ki*integral_term + Kd*derivative_term; //output of PID
controller
50
51    analogWrite(2) //o utput at pin 2
52    delay(#)
53
54 }
```

```
55
56  float get_angle(int motor) //sensor logic
57 {
58     int i;
59     int sum = 0;
60     float actual_angle;
61     encoded_signal = get_count(); //get count sets the clock to read from shift register
62
63     for(i = 0; i <10; i++)
64     {
65         sum  = sum + (encoded_signal % 10)*2^i; //convert binary count into decimal
66     }
67
68     if(motor == 1)
69     {
70         actual_angle = (500 - sum)*2*3.14/500; //if it is motor one compute the angle as home
is 0 degrees
71     }
72     else
73     {
74         actual_angle = pi/2 + (500 - sum)*2*3.14/500; //if it is second motor add pi/2 as
homing is at pi/2
75     }
76     return(actual_angle)
77 }
```

## 3. Inverse Kinematics Calculation




$r_1^2 = x_{o2}^2 + y_{o2}^2 \rightarrow$ Eq 1

reference is 'a'

once you have $r_1$ we can get $\theta_o$ {in this case it will be -ve because clockwise}

$\theta_1 = \theta_x - \theta_o$ { in this case since $\theta_o$ is -ve it becomes $\theta_x + \theta_o$ }

$b^2 = a^2 + r_1^2 - 2ar_1\cos(\theta_o)$ { get $\theta_o$ from here }

$\theta_x = \tan^{-1}(y_{o2}/x_{o2})$

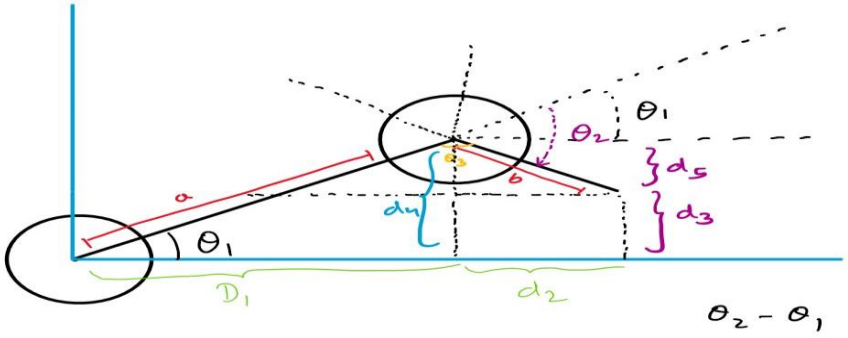
$r_1^2 = a^2 + b^2 - 2ab\cos(\theta_3) \rightarrow$ get $\theta_3$

$\theta_2 = 180 - \theta_3$

## 4. Inverse Kinematics Code

```
function [theta1, theta2] = inverse_kinematics(x0,y0)

a = 110*10^-3; %arm 1 length
b = 140*10^-3; %arm 2 length

r1 = sqrt(x0^2+y0^2);

thetaX = atan(y0/x0);

arg1 = (a^2 + r1^2 - b^2)/(2*a*r1);
arg1 = max(min(arg1,1),-1);
theta0 = acos(arg1);

theta1 = thetaX - theta0; %arm 1 angle

arg2 = (a^2 + b^2 - r1^2)/(2*a*b);
arg2 = max(min(arg2,1),-1);

theta3 = acos(arg2);

theta2 = pi - theta3; %arm 2 angle
```

## 5. Direct Kinematics Calculation




$$x = d_1 + d_2 = a\cos\theta_1 + b\cos(180 - \theta_3 - \theta_1)$$

$$\boxed{x = a\cos\theta_1 + b\cos(\theta_3 + \theta_1)}$$

$$\sin(\theta_1) = \frac{d_4}{a} \Rightarrow d_4 = a\sin\theta_1$$

$$d_5 = b\sin(180 - \theta_3 - \theta_1) = b\sin(\theta_3 + \theta_1)$$

$$y = d_3 = d_4 - d_5$$

$$\boxed{y = a\sin\theta_1 + b\sin(\theta_1 + \theta_3)}$$

## 6. Direct Kinematics Code

```
function [x, y] = direct_kinematics(theta_1,theta_2)

a = 110*10^-3; %arm 1 length
b = 140*10^-3; %arm 2 length

x = a*cos(theta_1) + b*cos(theta_2 + theta_1);
y = a*sin(theta_1) + b*sin(theta_1 + theta_2);
```

## 7. Path Planning MATLAB Code

```
function qdi   = path_generating(home_sig, qf_homing, q0_homing, q0, qf, tin, time_in_pos)
persistent count;
persistent previous_pos;
persistent return_trag;
persistent homing_logic;
persistent return_trag_home;

if isempty(count) %initialize
        count = 0;
        previous_pos =0;
        return_trag = false;
        homing_logic = true;
        return_trag_home = false;
end
if(home_sig == 1)
    home_signal = true;
else
    home_signal = false;
end

t0 = 0;
tf = tin;
t = count/24658;

if(t>tin)
    if(t>tin+time_in_pos) %if the time exceeds time to take to go from position 1 to position 2 and hold time then reset the time
        count = 0;
        t = 0;
        return_trag = ~return_trag;

        if (return_trag_home) %if going from position 1 to position 2 now go from position 2 to position 1
            return_trag_home = false;
        end
        if(homing_logic) %make homing logic false if it was true before. This is just for demo. In implementation homing sensor logic will take ca
            homing_logic = false;
            return_trag_home = true;
        end
    else
    qdi = previous_pos; %initial position = previous position
    d_qdi = 0; % velocity at initial position is 0
    count = count + 1;
    return
    end
end
```

```
if(homing_logic == true)
    q0 = [q0_homing,0]; % if homing logic is true prioritize homing
    qf = [qf_homing,0];
    if(home_signal == true)
        %current_agle = 0 // update postion (only in c code)
        homing_logic = false;
        return_trag_home = true;
        return_trag = ~return_trag;
        temp = q0;
        q0 = [qf_homing,0];
        qf = [temp,0];
    end
elseif(return_trag_home == true)

    temp = q0; %if following path and reached postion 2 from position 1 then interchange the coordinates to go from position 2 to position 1
    q0 = [qf_homing,0];
    qf = [temp,0];
elseif(return_trag == false) %else go from position 1 to postion 2
    q0 = [q0,0];
    qf = [qf,0];
else
    temp = q0;
    q0 = [qf,0];
    qf = [temp,0];
end


X = zeros(4, 4); %initialize time vector for equation position = a*t^3 + b*t^2 + c*t + d
B = zeros(4, 1); %inintialize B vector

% generate X matrix
X(1, 1) = 1;
X(1, 2) = t0;
X(1, 3) = t0^2;
X(1, 4) = t0^3;

X(2, 1) = 0;
X(2, 2) = 1;
X(2, 3) = 2 * t0;
X(2, 4) = 3 * t0^2;

X(3, 1) = 1;
X(3, 2) = tf;
X(3, 3) = tf^2;
X(3, 4) = tf^3;

X(4, 1) = 0;
X(4, 2) = 1;
X(4, 3) = 2 * tf;
X(4, 4) = 3 * tf^2;



%generate B matrix
B(1, 1) = q0(1);
B(2, 1) = q0(2);
B(3, 1) = qf(1);
B(4, 1) = qf(2);

%solve for coefficients
A = X\B;

a = [A(1, 1), A(2, 1), A(3, 1), A(4, 1)];

qdi = a(1) + a(2) * t + a(3) * t^2 + a(4) * t^3;
previous_pos = qdi;
d_qdi = a(2) + 2 * a(3) * t + 3 * a(4) * t^2;
count = count + 1;
return
end
```