Simulation of Engineering Systems Report 2

James Henrik Middleton

November 2025

Introduction

The goal of the second part of this assignment is to improve the controller functionality of the design from part 1. The primary improvement that will be made is the fine-tuning of the Gain parameter G_C. Once the system is performing better we will investigate the effect of changing the key parameter θ_U (Upper arm deflection angle) has on the rest of the system. We will achieve this by using interpolation on θ_U at specific time points supplied by the lab sheet.

1 Changing the Gain Term

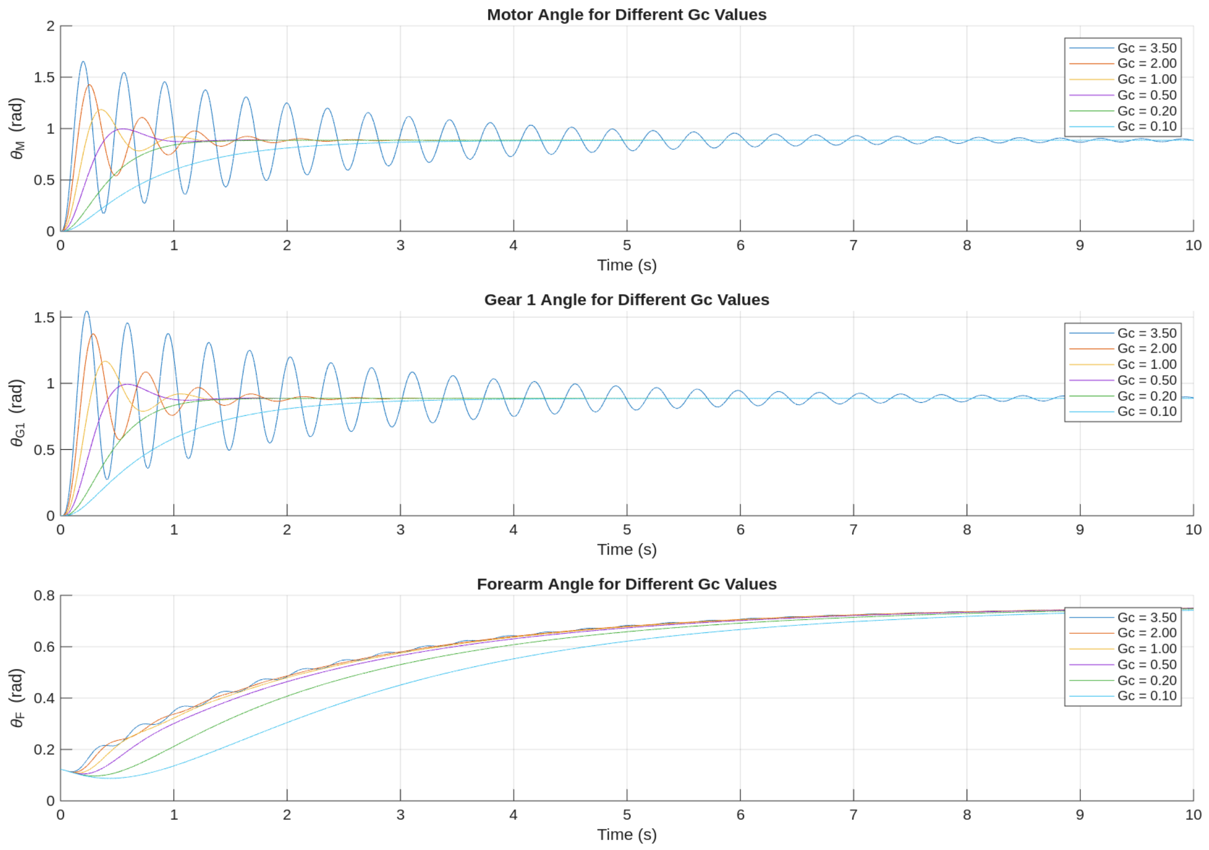

Figure 1: GC Variations

Figure 1 shows the affect altering GC has on θ_M, θ_{G1}, and θ_F. In each case reducing the size of G_C reduces oscillation significantly. For the original value of 3.5 provided by the labsheet the oscillations are very large. Values above 0.5 increase the amount of oscillations in the system, and values below 0.2 introduce significant damping. Based on this data a value of 0.2 is the most practical decision for G_C.

2 Selecting the Integration Coefficent

In order to improve the functionality of the simulation a new integration term was added. This term uses a coefficient called Ki to act as gain and a coeffecent call I to collect the sum of total past errors. Combining these terms and adding the current error gives the overall integration coefficient used to create Ve.

```
e = Thetaref - (x(2) * Ks); % Calculate error
I = I + (e * stepsize); % Sum of errors over time (Integral State)
KI = Ki * I; % Define Integral Term
Ve = Gc * (e + KI); % Calculate controller Voltage with integral term
Va = Ve * Kg; % Actuator Input Voltage
```

Integration Code

The code above Shows the method used to calculate the error, integration state, and integral term. I is a sum of the past errors Ve is calculate by multiplying G_C in the system and e is the current error. The integral term KI is calculate by multiplying the integral state (I) by the integral coefficient (Ki). KI is then used to calculate the controller voltage and therefore the Actuator Input voltage.

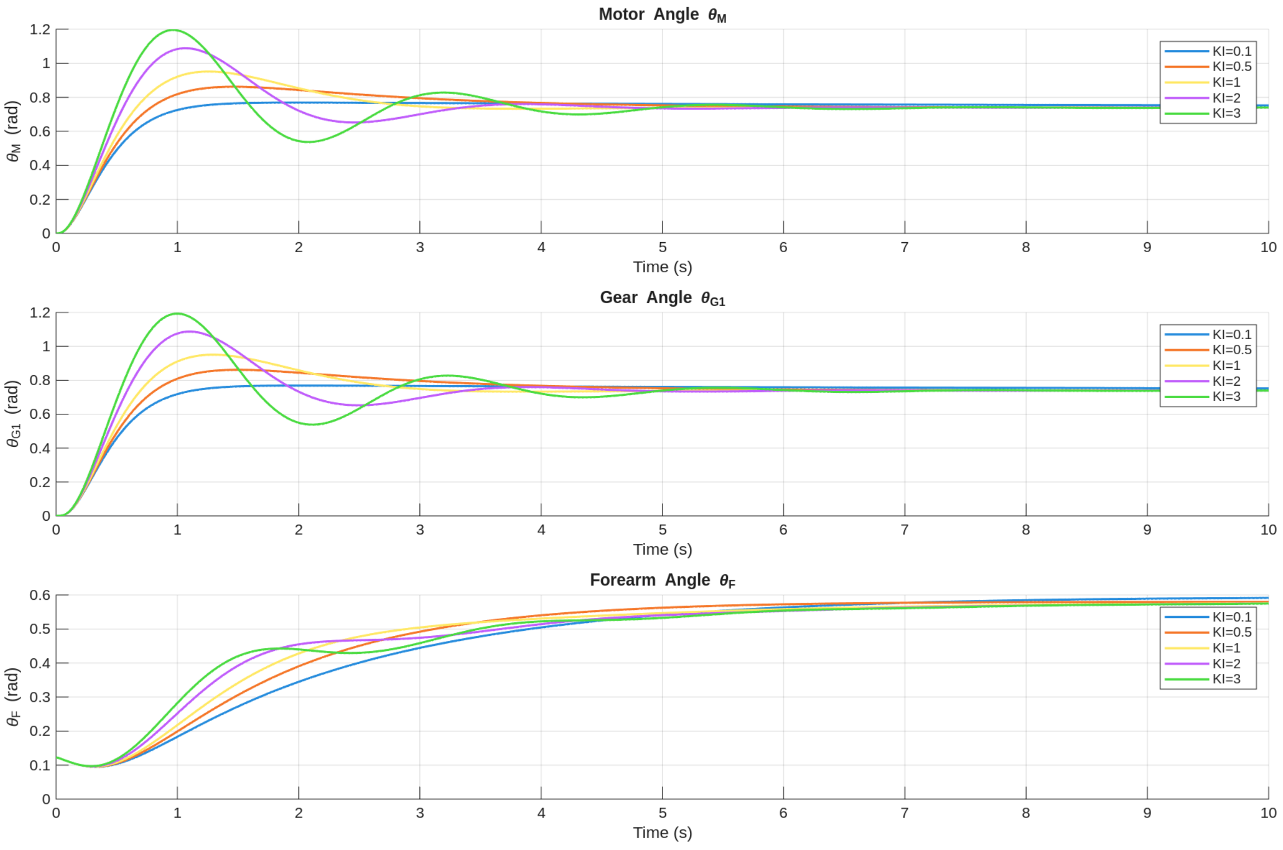

Figure 2: Effect of Varing the Integration Coefficient (Gc fixed at 0.2)

Figure 2 shows the effect that multiple different values of Ki have on the system deflection angles. Values of Ki above 1 lead to oscillations in the system, and values below approximately 0.4 slow down movement. A Ki value of 0.5 allows the arm to move smoothly and reach θ_{ref} the fastest out of the tested values, and so was selected to be used for the final design.

3 Interpolation of θ_U

3.1 Manual Interpolation

The table below shows the table produced from performing manual polynomial interpolation using Newton's Divided Differences.

x	$f[x]$	Order 1	Order 2	Order 3	Order 4	Order 5
0	3	5	-0.2273	5.3163×10^{-4}	7×10^{-4}	5×10^{-5}
1.5	10.5	3.75	-0.2193	0.0153	8×10^{-4}	
5.5	25.5	0.7895	0.0781	-0.0075		
15	33	2	-0.0982			
21	45	0.625				
29	50					

The coefficients extracted from this table can then be used in Newtons Polynomial Interpolation Equation.

$$f(x) = 3 + (x-0)*(5+(x-1.5)*(-0.22727+(x-5.5)*(0.00053+(x-15)*(0.0007+(x-21)*(-0.00005)))))$$
$$\Rightarrow f(0)=3, f(1.5)=10.5, f(5.5)=25.5, f(15)=33, f(21)=45, f(29)=50$$

The code below shows how this equation has been converted into MATLAB code.

```
function y = Manual_Interpolation(x)
    y = 3 + (x-0).*( 5 + (x-1.5).*( -0.2272727 + (x-5.5).*( ...
        0.0005316321 + (x-15).*( 0.0007009277 + (x-21).*( -0.0000527013 ) ) ) ) );
end
```

This manual meethod was used to validate the automatic functions at specific time intervals.

3.2 Interpolation through code

The code below shows the method used to derive the polynomial coefficients from a static array. It automatically creates a Divided Differences table from two equal sized input arrays. Each coefficient from the table is then extracted to be used in futher calculation. It has been validated against the manual interpolation table.

```
function [table, coefficients] = Calculate_Divided_Differences(x, y)
    n = length(y);
    table = zeros(n, n+1); % initialize matrix for storing values
    table(:, 1) = x(:); % Set Column 1 as the x values
    table(:, 2) = y(:); % Set Column 2 as the y values

    for j = 3:n+1 % Start at column 3 for divided differences
        for i = j-1:n
            % Divided difference formula
            % The denominator uses x(i) and x(i-(j-2))
            table(i,j) = (table(i,j-1) - table(i-1,j-1)) / (x(i) - x(i-(j-2)));
        end
    end

    coefficients = zeros(n,1); % Extract Divided Difference Coefficients
    for i = 1:n
        coefficients(i) = table(i, i+1);
    end
end
```

Divided Differences Code

The table below was produced by performing newtons divided difference evaluation on the table of values provided by the data sheet. The coefficients produced can be passed into the polynomial evaluation function to produce θ_U.

```
 x     f_x       1          2            3             4             5
___    ____    _______    ________    __________    ___________    ___________

  0       3          0           0             0             0             0
1.5    10.5          5           0             0             0             0
5.5    25.5       3.75    -0.22727             0             0             0
 15      33    0.78947     -0.2193    0.00053163             0             0
 21      45          2    0.078098      0.015251    0.00070093             0
 29      50      0.625   -0.098214    -0.0075027   -0.00082741    -5.2701e-05
```

Divided Differences Table From MATLAB Code

The code below details the process used to statically interpolate θ_U using the coefficients from the divided differences. In the static method all values are passed to the array "thetaUvals" to be printed later in the interpolated graph. The 'k' loop effectively emulates the passing of time, with xf being analogous to the time value. This allows interpolation outside of an external loop. However inserting this function into the main program would not work. The main function requires a dynamic function to calculate one θ_U value for each iteration of the Main loop, to be passed into the robot arm and rk4int functions. In the Dynamic Interpolation section the code from Polynomial Evaluation is refactored and simplified for integration into the main loop.

```
function thetaUvals = Polynomial_Evaluation(x, stepsize, coefficients)
    n = length(x);
    xf = 0:stepsize:x(n);
    thetaUvals = zeros(size(xf));

    for k = 1:length(xf)
        thetaU = coefficients(1); % Initial value (Starts as the first coefficient)
        prodTerm = 1; % Initialize product term (Acculmulation of the factors)

        for i = 2:n
            prodTerm = prodTerm * (xf(k) - x(i-1)); % Increase the product term by one degree
            thetaU = thetaU + coefficients(i) * prodTerm; % Multiply the coefficient by the product term
        end
        thetaUvals(k) = thetaU; % Add values to output matrix
    end
end
```

Polynomial Evaluation Code

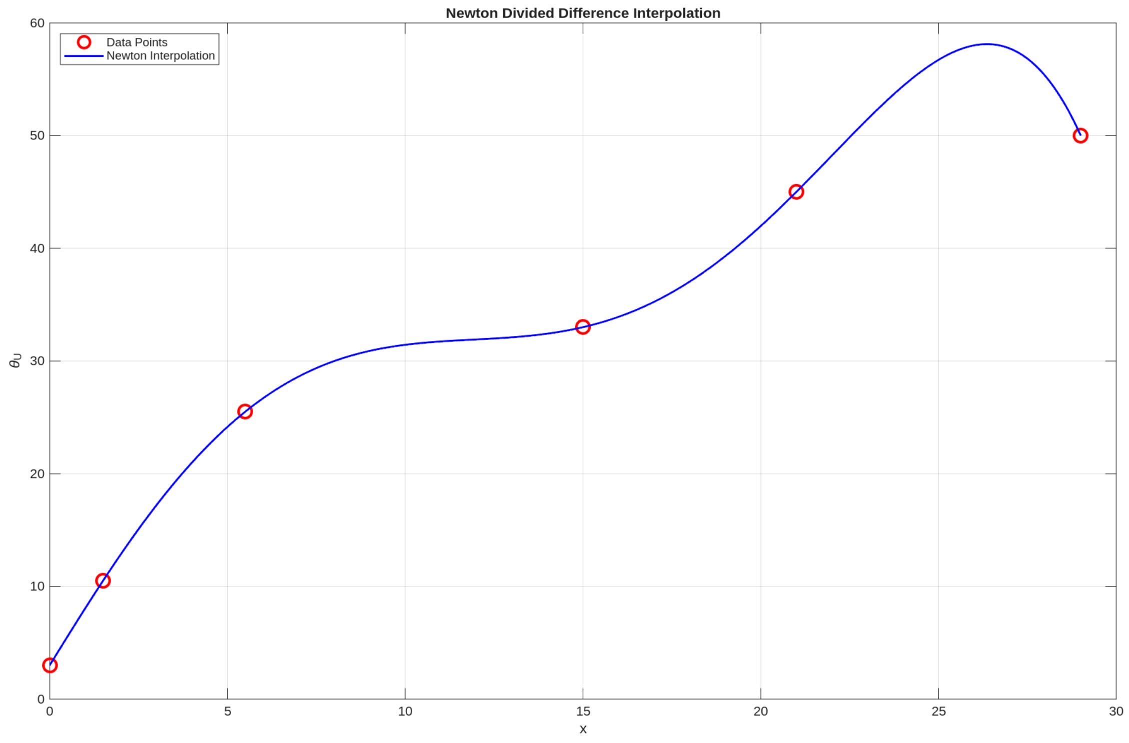

Figure 3: Interpolated Graph of θ_U From Sample Data

Figure 3 shows the graph of θ_U produced from static interpolation of the points provided. The line passes through all of the provided points which indicates that the method is valid. Towards the end of the line there is an oscillation that goes above the highest $theta_U$ value that was provided. This is known as Runge's phenomenon and is very common in high order polynomial interpolation.

4 Dynamic Interpolation

```
function thetaU = Interpolation_Realtime(x, xvalues, coefficients)
    thetaU = coefficients(1); % Initial value (Starts as the first coefficient)
    prodTerm = 1; % Initialize product term (Acculmulation of the factors)

    for i = 2:length(coefficients)
        prodTerm = prodTerm * (x - xvalues(i-1)); % Increase the product term by one degree
        thetaU = thetaU + coefficients(i) * prodTerm; % Multiply the coefficient by the product term
    end
end
```

Realtime Interpolation

The code above displays the function used to calculate polynomial interpolation at specific points in realtime.

```
% Calculate Interpolation Coefficients
[table, coeffs] = Calculate_Divided_Differences(xvals,yvals);
thetaUvals = Polynomial_Evaluation(xvals, stepsize, coeffs);
```

Interpolation Function calls

```
% Calculate Realtime thetaU
thetaU = Interpolation_Realtime(time, xvals, coeffs);

% Update state derivatives
xdot = robot_arm(x, Va, thetaU);

% Integration
x = rk4int(@robot_arm, x, stepsize, Va, thetaU);
```

Dynamic Interpolation

The code above shows how the interpolation code has been inserted into the Main function. Outside of the main loop the table is created and the coefficients are initialized. Inside of the main loop the realtime interpolation function is applied to produce interpolated θ_U values at each timestep interval, which are passed into the robot arm and rk4int functions.

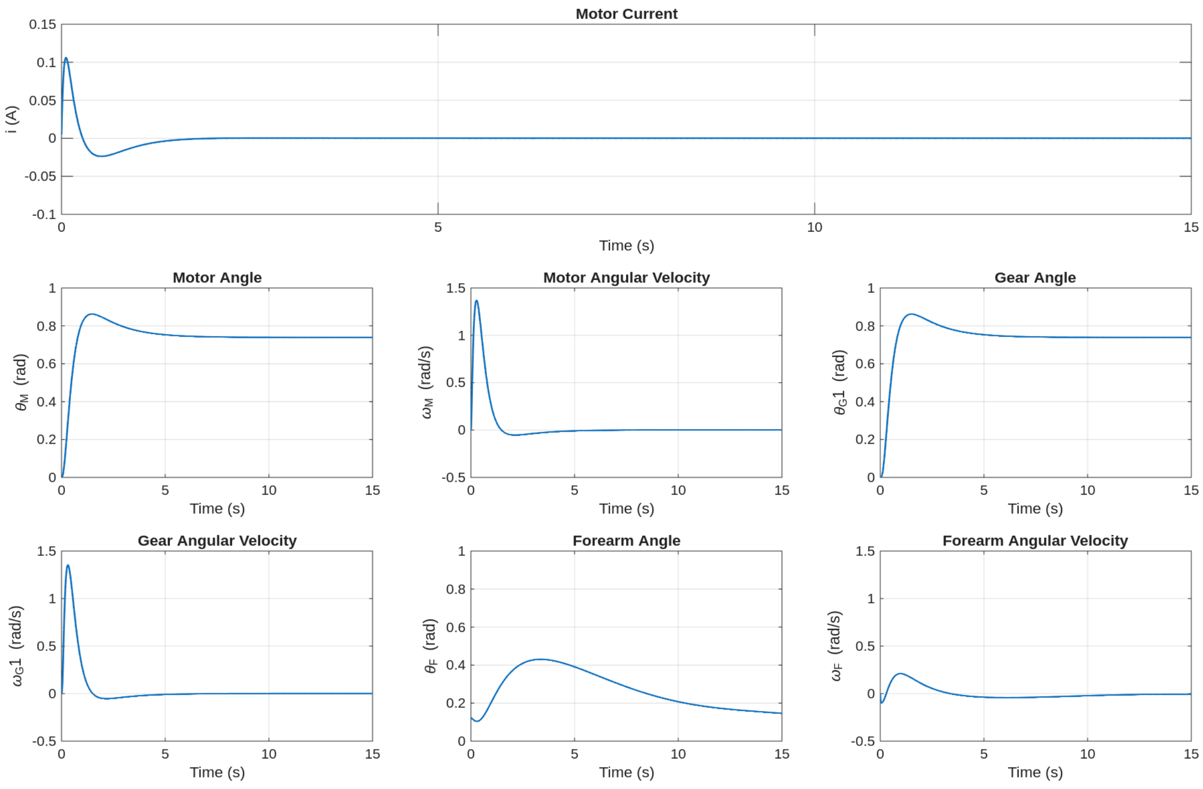

Figure 5: Graphs of State Variables with Dynamic interpolation of θ_U

The graphs in Figure 5 were produced by altering G_C to 0.2, introducing the integration term K_i, and the dynamic interpolation of θ_U within the main loop. The motor current stabilized significantly, rising and falling smoothly and quickly, coming to rest at 0 in approximately 1.2 seconds. The motor Angle and angular velocity stabilized significantly, following a steep and then shallow curve, with movement mostly stopping at approximately 2 seconds and coming to complete rest at approximately 5 seconds. The Gear Angle and Angular Velocity likewise stabilized significantly, following an almost identical path to the Motor Angle. The Forearm Angle Smoothly raises steadily to an angle of approximately 0.41 radians. It then gradually falls back to 7°. Likewise forearm angular velocity rises and then dips slightly below zero until coming to rest, which allows for the forearm angle to decrease. This is due to the effect gravity has on the system. If Gravity is removed then this does not happen, and if the forearm length or weight is increased it gets worse. To prevent this there would have to be constant current applied to the circuit to maintain an equal upwards acceleration against gravity.

5 Conclusion

Overall this Simulation is a significant improvement on the original. By carefully tuning gain values jitter has been practically removed from the system and it operates smoothly. This shows how important the simulation phase can be when designing a product. However some improvement can still be made on the system. The arm does not reach the desired apogee of 55°, stopping at just under halfway. From experimentation it has been deduced that this is due to the effect gravitational constant has on the arm. Once the motor stops recieving current, the torque applied to the system in the upwards direction reduces to zero, and the effect of gravity takes over pulling the arm down. In order to prevent this it would be possible to implement a system with a constant low current to combat gravity, or increase the moment of inertia of the components so that gravity is not strong enough to pull the arm down.