2.0 Introduction

The second part of this assignment focuses of improving the response of the Robotic Arm Model and increasing its complexity by adding movement to the upper arm as well. This is initially done through tuning the value of the Controller Gain (G_C). In the first section the system used a Proportional Controller, this investigation goes further to design an Integral Proportional Controller by introducing an Integral term, $K_I I$. Through these implementations, the optimal G_C value will decrease the oscillations and K_I will reduce steady state error. Finally, through Newton's Divided Difference Approach the value of θ_U, the upper arm angle will be interpolated.

2.1 Gc Control Gain Variation Results & Analysis

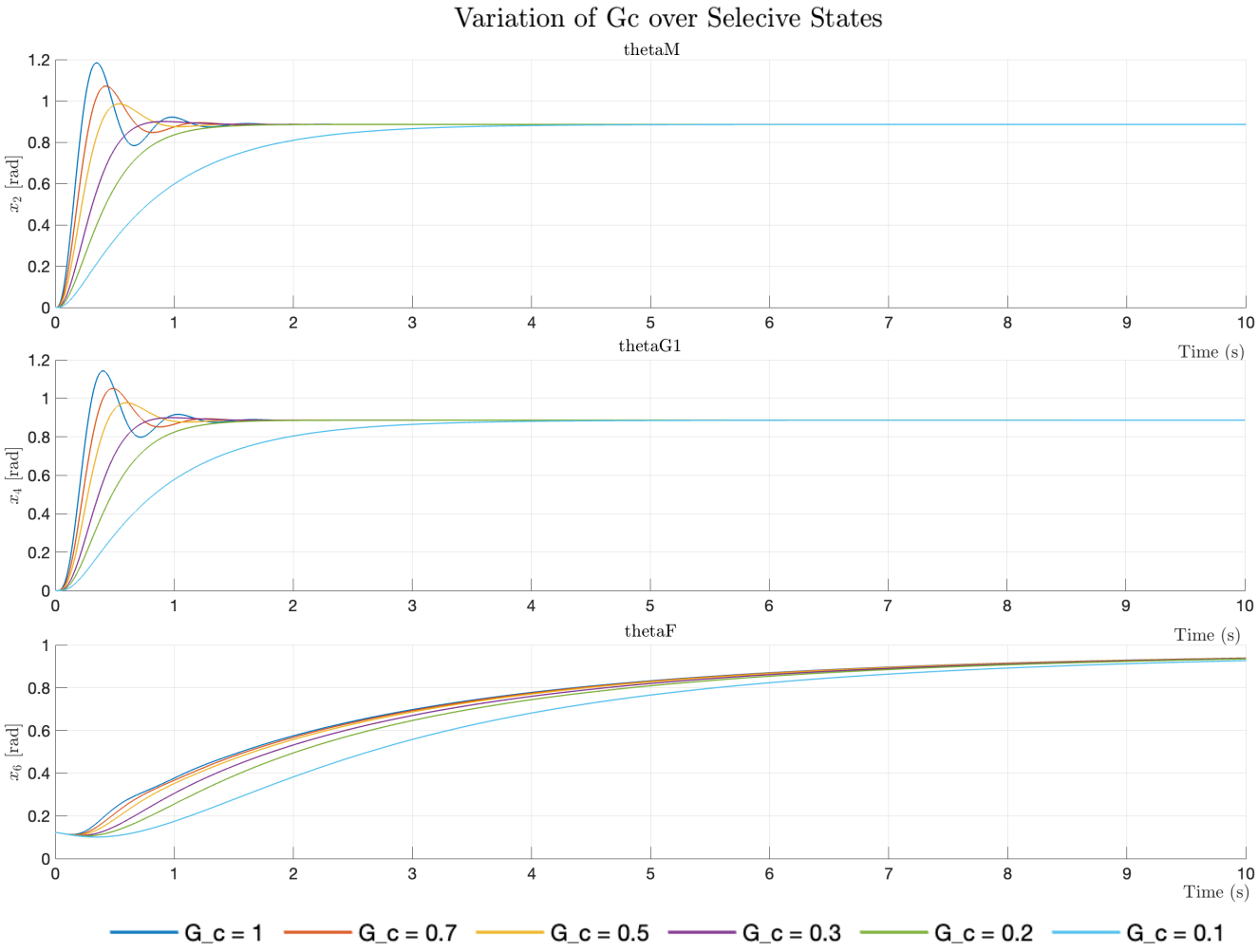

Figure 1: Variation of G_C

Figure 1 illustrates how the decrease of G_C decreases the oscillations of the system. In higher values such as 0.5, 0.7 and 1 there is overshoot observed on the response of the system. The value chosen for G_C is ultimately 0.2, as observed even slight differences on G_C effect the oscillatory response of the states. The final choice was between 0.1, 0.2 or 0.3. The G_C value of 0.1 was not chosen because even though there is has no overshoot observes, the system takes a longer time to reach the set-point. The value of 0.3 achieves the set-point at approximately the same time as 0.2, this however was not chosen due to the slight overshoot. At the chosen value of 0.2 for G_C there is faster rise time, minimal overshoot and it reaches trim condition faster than other values.

2.2 Integration Implementation

Adding an Integral Term in the control signal allows the response to reach the desired 55° slightly faster and enhance accuracy by decreasing steady-state error. Different values of K_I were investigated in order find the one that allows the response to reach the trim condition fastest with no further impact on the overshoot. The Integral Term implementation inside my MATLAB code:

```
theta_ref = K_R * deg2rad(55);      % (1)
error = theta_ref - (K_S * x(2));   % (2) Calc error
I = I + (error * stepsize);         % (3) Update the integral state
Ve = Gc * (error + K_I * I);        % (4) Control voltage with KI
Va = Ve * K_G;                      % (5) Drives the arm
```

The control section of the main script was changed to the code above. The accumulated integral of the error over time (I) was initially set to zero, line (3) of the code implements the numerical integration of the error, this helps eliminate steady-state errors by considering the cumulative error over time. Line (4) of the code implements the PI controller equation given in the specification.

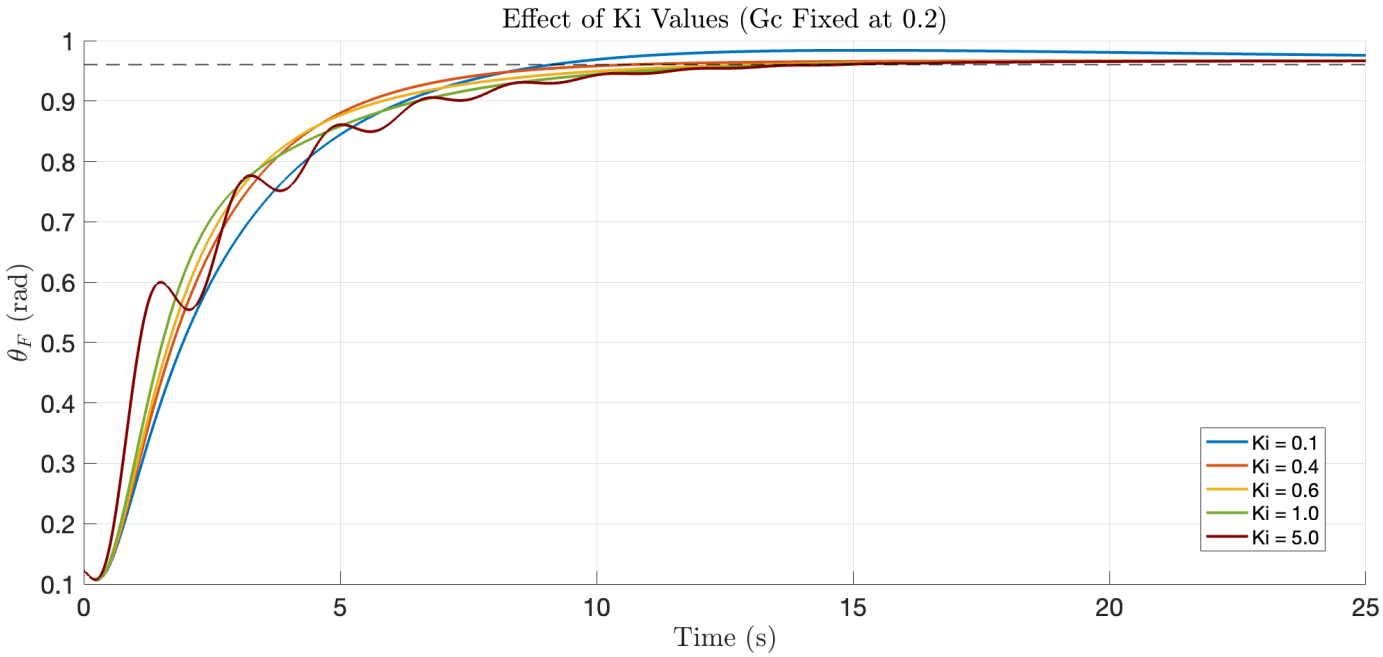

Figure 2: Variation of K_I and its Effect on the Response

Figure 2 allows us to visualize the effect of the K_I on the response of the system and the time it takes to reach trim conditions, the dotted line which represents 0.96, 55° that is θ_{ref}. A higher K_I increases the contribution of the integral term, which is supposed to accelerate the correction of errors and can speed up the settling time. As illustrated in Figure 2 when K_I is too large, 5.0, it can introduce instability, leading to oscillations. It was found that values above 1.0 created oscillations in the response. The value chosen for K_I is 0.4, this was due to its faster response to reaching 55°. Compared to other values it is the one did not create excessive overshoot and had the fastest response. However, it is significant to mention that no K_I value eliminates steady-state error entirely.

2.3 Interpolation Algorithm Calculation

Newton's Divided Differences Interpolation Polynomial Table:

x	θ_U	$\Delta f[x_0, x_1]$	$\Delta^2 f[x_0, x_1, x_2]$	$\Delta^3 f[x_0, x_1, x_2, x_3]$	$\Delta^4 f[x_0, x_1, x_2, x_3, x_4]$	$\Delta^5 f[x_0, x_1, x_2, x_3, x_4, x_5]$
0	3.0					
2.5	10.5	$\frac{10.5-3}{2.5-0} = 3$				
6.5	25.5	$\frac{25.5-10.5}{6.5-2.5} = 3.75$	$\frac{3.75-3}{6.5-0} = 0.11$			
17	30	$\frac{30-25.5}{17-6.5} = 0.43$	$\frac{0.43-3.75}{17-2.5} = -0.23$	$\frac{-0.229-0.11}{17-0} = -0.02$		
22.5	37	$\frac{37-30}{22.5-17} = 1.27$	$\frac{1.27-0.43}{22.5-6.5} = 0.05$	$\frac{0.05-(-0.23)}{22.5-2.5} = 0.014$	$\frac{0.014-(-0.02)}{22.5-0} = 0.0015$	
28	50.5	$\frac{50.5-37}{28-22.5} = 2.45$	$\frac{2.45-1.27}{28-17} = 0.11$	$\frac{0.107-0.053}{28-6.5} = 0.0025$	$\frac{0.0025-0.014}{28-2.5} = -0.00045$	$\frac{-0.00045-0.0015}{28-0} = -0.0000707$

The Newton's Interpolation Polynomial based on this table is:

$$P_n(x) = f(x_0) + (x - x_0)f[x_0, x_1] + \cdots + (x - x_0)(x - x_1)\cdots(x - x_{n-1})f[x_0, x_1, \ldots, x_n].$$

$$P(x) = 3+(x)(3+(x-2.5)(0.115+(x-6.5)(-0.02+(x-17)(0.0015+(x-22.5)\times-0.00007)$$

2.4 Interpolation Matlab Code, Results & Analysis

MATLAB Code

```
% ------------------------ newtons_coeff.m file -------------------------

function [coefficients, table] = newtons_coeff(x, y)
    n = length(y);                % Number of data points
    table = zeros(n, n);          % Initialize matrix for table
    table(:, 1) = y(:);           % Fill the first column
    for j = 2:n                   % Loop columns
        for i = j:n               % Loop rows
            table(i, j) = (table(i, j-1) - table(i-1, j-1)) / ...
                          (x(i) - x(i-j+1)); % Compute/Populate
        end
    end
    coefficients = diag(table);  % Extract diagonal elements
end

% ----------------- newton_interpolation.m file -------------------------

function interpolated_value = newton_interpolation(t, x_points, coefficients)
    interpolated_value = coefficients(1);
    term = 1;                        % Initialize product term
    for k = 2:length(coefficients) % Loop coefs - start from 2nd
        term = term * (t - x_points(k-1)); % Update product term
        interpolated_value = interpolated_value + coefficients(k)  * term;
    end
end
```

```
% ------------------------------ main.m file ------------------------------
% ADDITIONS OUTSIDE LOOP:
time_var = [0, 2.5, 6.5, 17, 22.5, 28];
theta_u_var = [3.0, 10.5, 25.5, 30, 37, 50.5];
[theta_u_coefficients, newton_table] = newtons_coeff(time_var, theta_u_var);
% ADDITIONS INSIDE LOOP - DYNAMIC SECTION:
theta_U = newton_interpolation(time, time_var, theta_u_coefficients);    % (1)

% Newton's Interpolation Polynomial validate results
% theta_U = 3 + (time) * (3 + (time - 2.5) * (0.115 + (time - 6.5) * ...
%       (-0.02 + (time - 17) * (0.0015 + (time - 22.5) * (-0.00007))))); % (2)
```

In order to interpolate θ_U, Newton's Divided Difference Interpolation Polynomial was implemented within MATLAB. Two helper functions were created, newtons_coeff.m and newton_interpolation.m, along with minor changes on the main script. The interpolation method was validated through two approaches: first, by dynamically calculating the coefficients and constructing the interpolation equation using the functions above (line "(1)" in code and the helper functions); and second, by using Newton's Interpolated Polynomial (line "(2)" in code). Both approaches produced identical graphs, confirming the method's accuracy.

Results and Analysis

```
Newton Divided Difference Table:
         x         y        Δ^1       Δ^2       Δ^3       Δ^4       Δ^5
    0.0000    3.0000
    2.5000   10.5000    3.0000
    6.5000   25.5000    3.7500    0.1154
   17.0000   30.0000    0.4286   -0.2291   -0.0203
   22.5000   37.0000    1.2727    0.0528    0.0141    0.0015
   28.0000   50.5000    2.4545    0.1074    0.0025   -0.0005   -0.0001
```

Figure 3: Terminal Calculation of Newton's Coefficients - Validates newtons_coeff.m

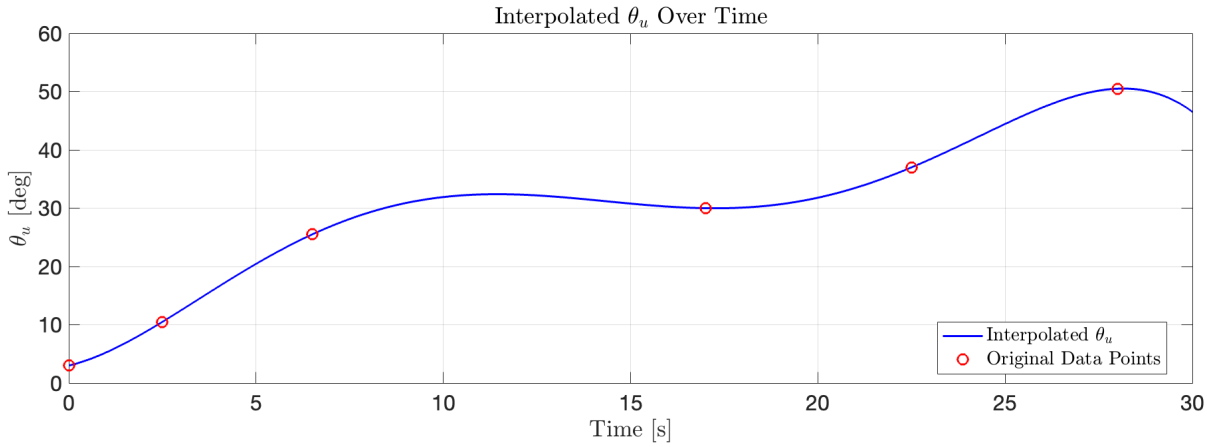

Figure 4: Interpolation of θ_U and Time Points Given - Validates newton_interpolation.m

To confirm the accuracy of the coefficient function and interpolation function, I created a console table as seen in Figure 3. This illustrates the Newton's Divided Differences Table that can be validated against the handwritten derivation from Section 2.3. Additionally, Figure 4 demonstrates the interpolated θ_U over a 30-second interval, ensuring that

the interpolation is correctly implemented and passes through the specified data points provided in the task sheet.

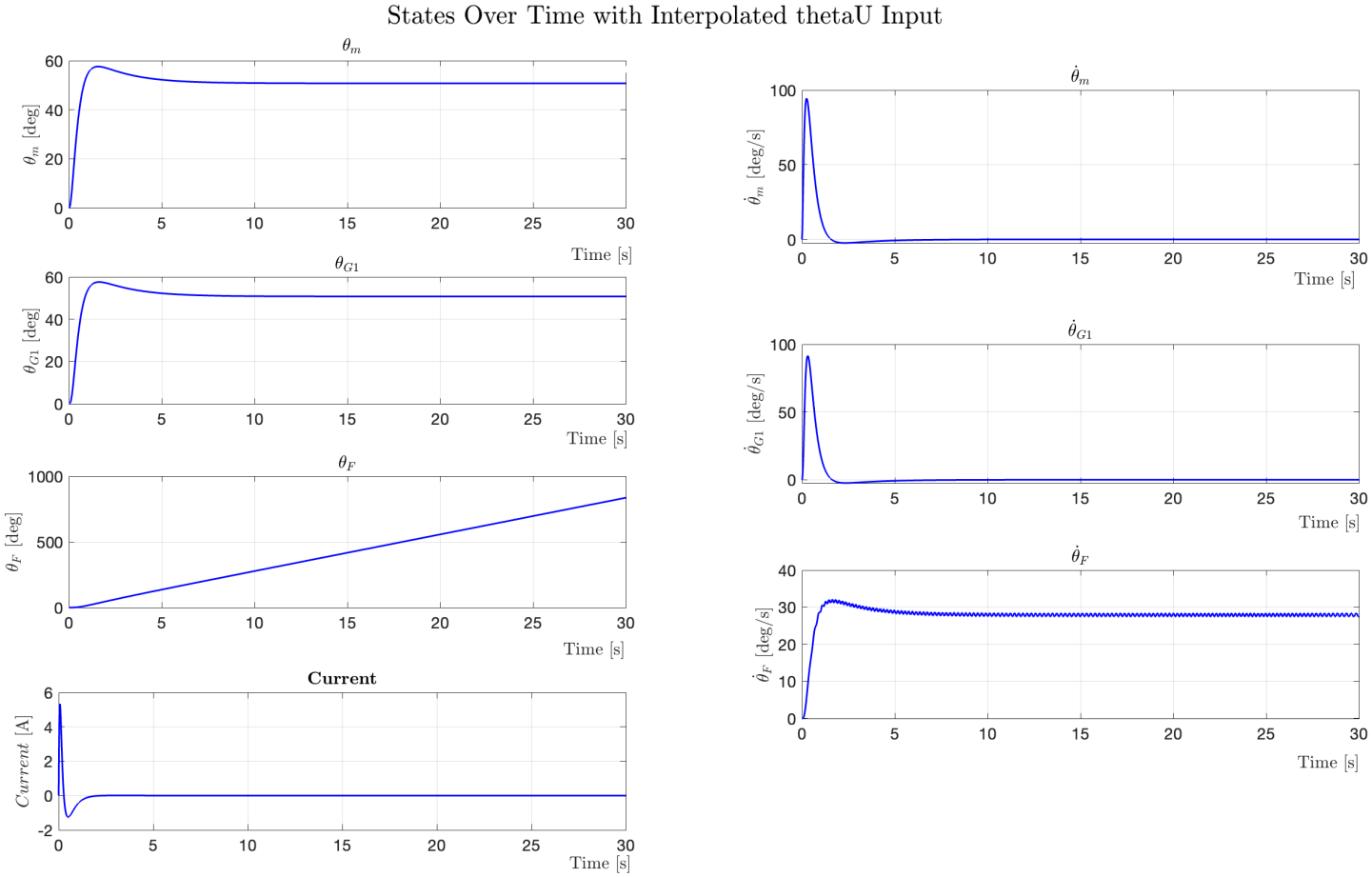

Figure 5: All Seven States with Interpolated θ_U

The angle of the upper arm was varied over time. In the first part of this assignment it was assumed that the upper arm of the Robotic Arm was kept stationary, however, the system is now more complex, both the the upper arm and the forearm move. Figure 5 depicts the response of all states over 30 seconds with the chosen G_C, K_I and interpolated θ_U. There is minimal overshoot in θ_M and θ_{G1}, however, overall the responses has majorly improved from the first part of this assignment. There is a reduction of abrupt changes in the system response. The interpolation of θ_U results in a smoother, gradual change in θ_F as expected. The forearm angle and upper arm angle are dynamically related, hence, changes in θ_U propagate into θ_F.

2.5 Conclusion

By carefully tuning both gains, a response can be achieved that minimizes oscillations, reduces overshoot, and decreases steady-state error, ultimately allowing the system to converge more quickly to θ_{ref}. Through decreasing the value of the Proportional Gain to 0.2 and slowly increasing Integral Gain to 0.4 an optimal response was obtained. However, as illustrated in Figure 5, it is significant to highlight that the system still has some minimal overshoot and not completely eliminated steady-state error. This may be due to the system's increased complexity which could necessitate the tuning of the other gains further and require deeper damping effects. Another possibility is that the design inherently includes an integral controller, which could explain why no K_I eliminates the steady-state error and why slightly further increases in K_I have an adverse effect.