

DS288 Numerical Methods

Assignment Number 4

November 5, 2024

Sandeep Kumar
24022

1. For the given four-bar mechanism (DA, AB, CB, and DC), ϕ is obtained using Newton's Method from Homework 2 (Question 3). The plot of ϕ vs. θ is shown in Figure 1 below. The initial values for θ_2 and θ_3 are taken as 30° and 0° respectively. and θ is increased by 1°

Answer:

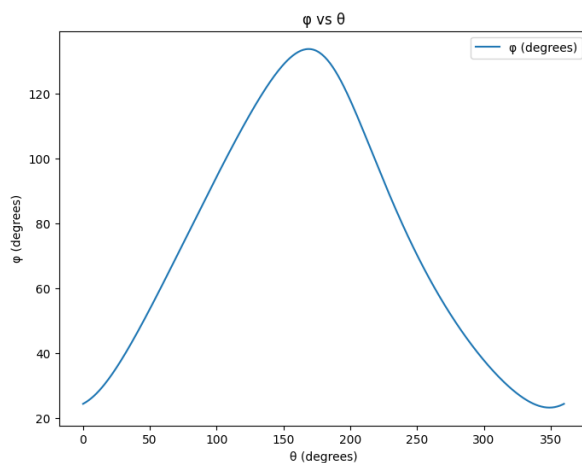


Figure 1: Plot of ϕ vs. θ

The derivative $\frac{d\phi}{d\theta}$ is obtained using the first forward difference and first central difference approximations. It is plotted in Figure 2 below:

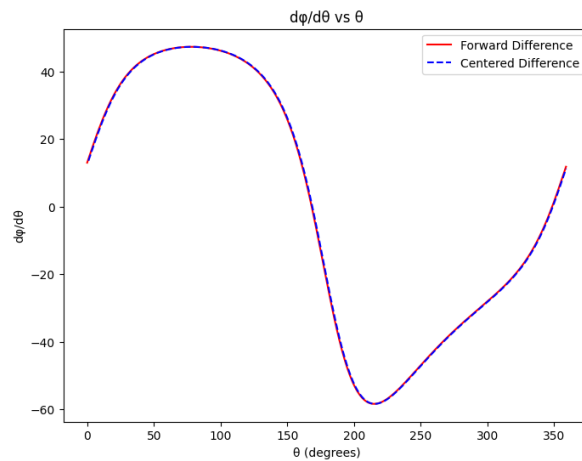


Figure 2: Plot of $\frac{d\phi}{d\theta}$ vs. θ using forward and central difference methods

Conclusion: From Figures 1 and 2, it can be seen that $\frac{d\phi}{d\theta}$ obtained using the first forward difference and first central difference methods almost overlap. Since the order of error for the first forward difference approximation is $O(hf'')$ and for the first central difference approximation is $O(h^2f''')$, we conclude that the central difference method is more accurate. For some initial values of θ_2 and θ_3 , Newton's method converges to some values of ϕ , which doesn't satisfy the four bar mechanism physically.

2. For the given bar mechanism (CE, EF, GF, and GC), β is obtained using Newton's Method from Homework 2 (Question 3). The plot of β vs. θ is shown in Figure 3 below. The initial values for θ_2 and θ_3 are taken as 330° and 30° respectively, and θ is increased by 1° .

Answer:

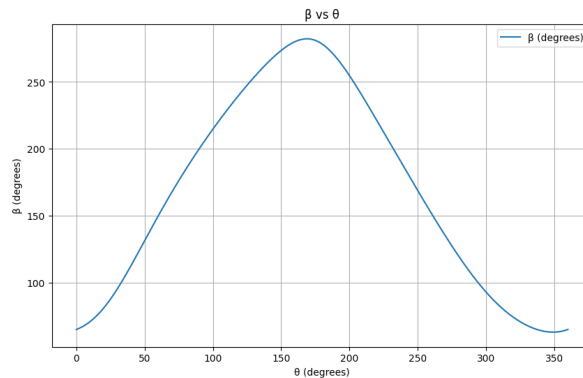


Figure 3: Plot of β vs. θ

The derivative $\frac{d\beta}{dt}$ is obtained using the first forward difference and first central difference approximations, as plotted in Figure 4 below:

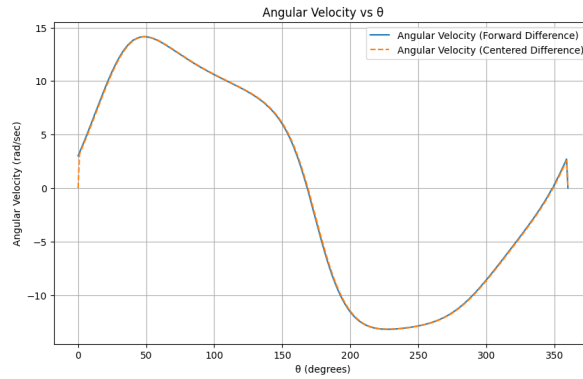


Figure 4: Plot of $\frac{d\beta}{dt}$ vs. θ using forward and central difference methods

The derivative $\frac{d^2\beta}{dt^2}$ is obtained using the first forward difference and first central difference approximations, as plotted in Figure 5 below:

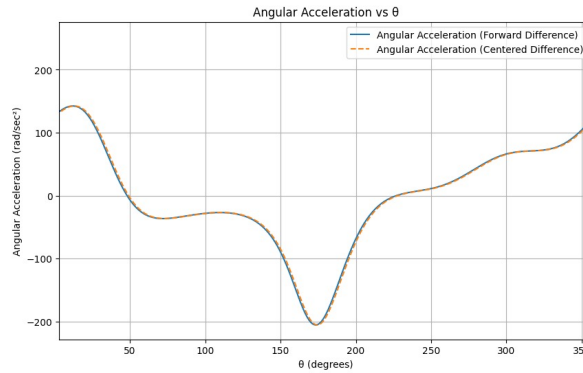


Figure 5: Plot of $\frac{d^2\beta}{dt^2}$ vs. θ using forward and central difference methods

Conclusion: β is calculated using Newton's Method for second four bar mechanism(CE, EF, GF, and GC). $\frac{d\beta}{dt}$ is calculated using first forward difference approximation and first central difference approximation, and plotted in the figure 4 against θ . it can be noticed that both first forward difference and central difference approximation give almost same results. but the order of error for forward difference and central difference approximation are $O(hf'')$ and $O(h^2f''')$ respectively. Similarly, $\frac{d^2\beta}{dt^2}$ is calculated using first forward difference approximation and first central difference approximation, and plotted in the figure 5 against θ . it can be noticed that both first forward difference and central difference approximation give almost same results. but the order of error for forward difference and central difference approximation are $O(hf''')$ and $O(h^2f''''')$ respectively. for some initial values of θ_2 and θ_3 , Newton's method converges to some values of β , which doesn't satisfy the four bar mechanism physically.

Appendix: Python Code

1 ## Question 1

```

2     import numpy as np
3     import matplotlib.pyplot as plt
4
5     # Constants
6     DA = 1.94 # r4 = DA
7     AB = 6.86 # r3 = AB
8     CB = 2.36 # r2 = CB
9     DC = 7.00 # r1 = DC
10
11    # Assign constants to variables
12    r1 = DC
13    r2 = CB
14    r3 = AB
15    r4 = DA
16
17    # Function definitions for Newton-Raphson
18    def f1(theta2, theta3, theta):
19        return r2 * np.cos(theta2) + r3 * np.cos(theta3) - r4 * np.cos(theta) - r1
20
21    def f2(theta2, theta3, theta):
22        return r2 * np.sin(theta2) + r3 * np.sin(theta3) - r4 * np.sin(theta)
23
24    # Jacobian matrix
25    def jacobian(theta2, theta3):
26        return np.array([
27            [-r2 * np.sin(theta2), -r3 * np.sin(theta3)],
28            [r2 * np.cos(theta2), r3 * np.cos(theta3)]
29        ])
30
31    # Newton-Raphson system function
32    def newton_system(r1, r2, r3, r4, thetas):
33        phi_vals = np.zeros_like(thetas, dtype=float) # Store phi values (angle of link
34        3)
35        phi_vals2 = np.zeros_like(thetas, dtype=float) # Store auxiliary values if needed
36
37        # Iterate over each theta to compute phi
38        for theta_deg in thetas:
39            i = int(theta_deg)
40            theta_rad = np.deg2rad(theta_deg) # Convert theta to radians for calculations
41
42            # Use previous solution as initial guess
43            theta2_old = np.deg2rad(30) if i == 0 else np.deg2rad(phi_vals[i - 1])
44            theta3_old = np.deg2rad(0) if i == 0 else np.deg2rad(phi_vals2[i - 1])
45
46            # Newton's Method parameters
47            tolerance = 1e-4
48            converged = False
49
50            # Newton-Raphson loop
51            while not converged:
52                # Compute function values and Jacobian
53                F = np.array([f1(theta2_old, theta3_old, theta_rad), f2(theta2_old,
54                theta3_old, theta_rad)])
55                J = jacobian(theta2_old, theta3_old)
56
57                # Newton's method update
58                delta = np.linalg.solve(J, F)
59                theta2_new = theta2_old - delta[0]
60                theta3_new = theta3_old - delta[1]
61
62                # Check convergence
63                if abs(theta2_new - theta2_old) < tolerance and abs(theta3_new -
                theta3_old) < tolerance:
                    converged = True

```

```

64         # Update for next iteration
65         theta2_old = theta2_new
66         theta3_old = theta3_new
67
68         # Store computed phi (angle of link 3)
69         phi_vals[i] = np.rad2deg(theta2_new)
70         phi_vals2[i] = np.rad2deg(theta3_new)
71
72     return phi_vals
73
74
75     thetas = np.arange(0, 361, 1) # Theta range in degrees
76     phi_vals = newton_system(r1, r2, r3, r4, thetas)
77
78     # Compute derivatives (forward and centered difference)
79     dphi_dtheta_forward = np.zeros_like(thetas, dtype=float)
80     dphi_dtheta_centered = np.zeros_like(thetas, dtype=float)
81
82     # Forward difference for dphi/dtheta
83     dphi_dtheta_forward[:-1] = (phi_vals[1:] - phi_vals[:-1]) / np.deg2rad(1)
84
85     # Centered difference for dphi/dtheta
86     dphi_dtheta_centered[1:-1] = (phi_vals[2:] - phi_vals[:-2]) / (2 * np.deg2rad(1))
87
88     ## Question 2
89
90     def normalize_angle(angle):
91         return angle % 360
92
93     alpha_vals = np.zeros_like(thetas, dtype=float)
94     alpha_vals = 149 + phi_vals
95     #print(alpha_vals)
96     alpha_vals = normalize_angle(alpha_vals)
97
98     r12 = 1.25
99     r22 = 1.26
100    r32 = 1.87
101    r42 = 2.39
102    # Function definitions for Newton-Raphson
103    def f12(theta2, theta3, theta):
104        return r22 * np.cos(theta2) + r32 * np.cos(theta3) - r42 * np.cos(theta) - r12
105
106    def f22(theta2, theta3, theta):
107        return r22 * np.sin(theta2) + r32 * np.sin(theta3) - r42 * np.sin(theta)
108
109
110
111    # Newton-Raphson system function
112    def newton_system2(thetas):
113        phi_vals = np.zeros_like(thetas, dtype=float)
114        phi_vals2 = np.zeros_like(thetas, dtype=float)
115        i = 0
116        # Iterate over each theta to compute phi
117        for theta_deg in thetas:
118
119            theta_rad = np.deg2rad(theta_deg) # Convert theta to radians for calculations
120
121            # Use previous solution as initial guess
122            theta2_old = np.deg2rad(330) if i == 0 else np.deg2rad(phi_vals[i - 1])
123            theta3_old = np.deg2rad(30) if i == 0 else np.deg2rad(phi_vals2[i - 1])
124
125            # Newton's Method parameters
126            tolerance = 1e-4
127            converged = False
128

```

```

129         # Newton-Raphson loop
130         while not converged:
131             # Compute function values and Jacobian
132             F = np.array([f12(theta2_old, theta3_old, theta_rad), f22(theta2_old,
133                               theta3_old, theta_rad)])
134             J = jacobian(theta2_old, theta3_old)
135
136             # Newton's method update
137             delta = np.linalg.solve(J, F)
138             theta2_new = theta2_old - delta[0]
139             theta3_new = theta3_old - delta[1]
140
141             # Check convergence
142             if abs(theta2_new - theta2_old) < tolerance:
143                 converged = True
144
145             # Update for next iteration
146             theta2_old = theta2_new
147             theta3_old = theta3_new
148
149             # Store computed phi (angle of link 3)
150             phi_vals[i] = np.rad2deg(theta2_new)
151             phi_vals2[i] = np.rad2deg(theta3_new)
152             i = i + 1
153
154         return phi_vals
155
156     def normalize_angle(angle):
157         return angle % 360
158
159     beta_vals = newton_system2(alpha_vals)
160
161
162     beta_vals = normalize_angle(beta_vals)
163
164     # Function for forward and centered differences
165     def compute_derivatives(beta_vals, thetas, omega):
166         # Forward difference for d / d
167         dbeta_dtheta_forward = np.zeros_like(beta_vals)
168
169         dbeta_dtheta_forward[:-1] = np.deg2rad(beta_vals[1:] - beta_vals[:-1]) /
170             np.deg2rad(1)
171
172         # Centered difference for d / d
173         dbeta_dtheta_centered = np.zeros_like(beta_vals)
174         dbeta_dtheta_centered[1:-1] = np.deg2rad(beta_vals[2:] - beta_vals[:-2]) / (2 *
175             np.deg2rad(1))
176
177         # Angular velocity (d / dt) using both methods
178         angular_velocity_forward = omega * dbeta_dtheta_forward
179         angular_velocity_centered = omega * dbeta_dtheta_centered
180
181         # Forward difference for d^2 / d^2
182         d2beta_dtheta2_forward = np.zeros_like(beta_vals)
183         d2beta_dtheta2_forward[:-2] = np.deg2rad(beta_vals[2:] - 2 * beta_vals[1:-1] +
184             beta_vals[:-2]) / (np.deg2rad(1) ** 2)
185
186         \# Centered difference for d^2 / d^2
187         d2beta_dtheta2_centered = np.zeros_like(beta_vals)
188         d2beta_dtheta2_centered[1:-1] = np.deg2rad(beta_vals[2:] - 2 * beta_vals[1:-1] +
189             beta_vals[:-2]) / (np.deg2rad(1) ** 2)
190
191         # Angular acceleration d^2 / dt^2 using both methods
192         angular_acceleration_forward = omega ** 2 * d2beta_dtheta2_forward

```

```
189         angular_acceleration_centered = omega ** 2 * d2beta_dtheta2_centered
190
191     return angular_velocity_forward, angular_velocity_centered,
192           angular_acceleration_forward, angular_acceleration_centered
193
194     angular_velocity_forward, angular_velocity_centered, angular_acceleration_forward,
195     angular_acceleration_centered = compute_derivatives(beta_vals, thetas, 7.5)
```