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:

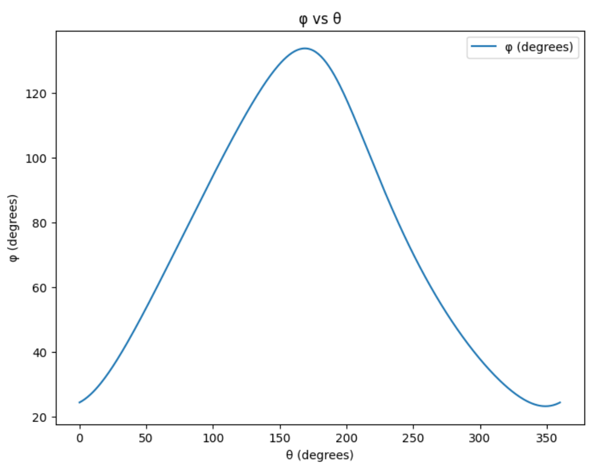

Figure 1: Plot of ϕ vs. θ

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:

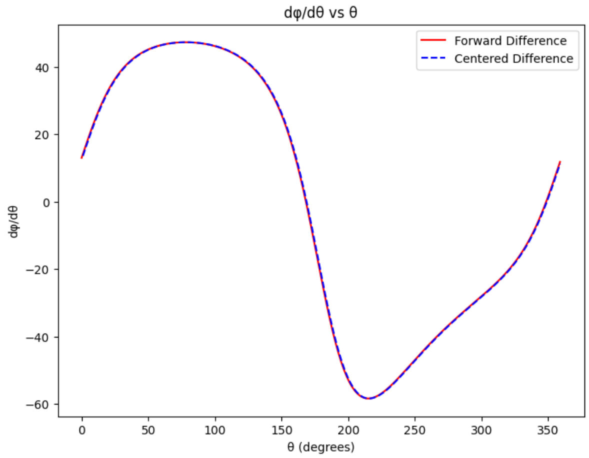

Figure 2: Plot of $\frac{d\phi}{d\theta}$ vs. θ using forward and central difference methods

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 $O(h^2 f''')$, 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:

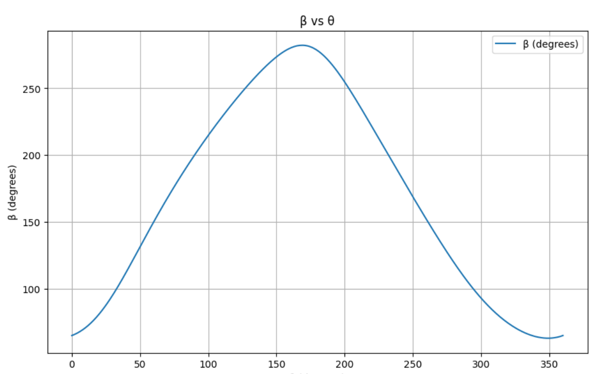

Figure 3: Plot of β vs. θ

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

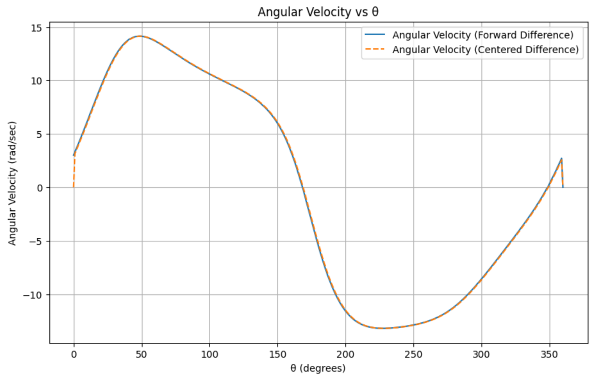

Figure 4: Plot of $\frac{d\beta}{dt}$ vs. θ using forward and central difference methods

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:

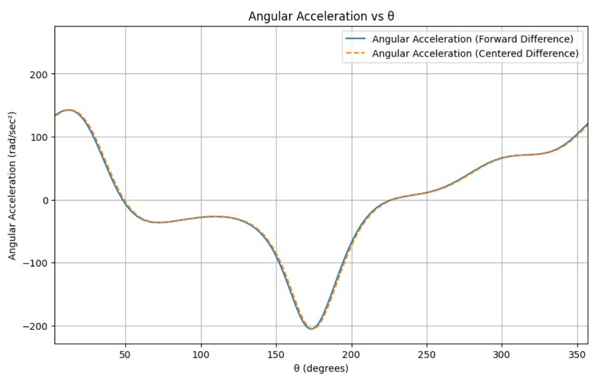

Figure 5: Plot of $\frac{d^2\beta}{dt^2}$ vs. θ using forward and central difference methods

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 $O(h^2 f''')$ 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 $O(h^2 f'''')$ 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

```
## Question 1
```

```
        import numpy as np
        import matplotlib.pyplot as plt

        # Constants
        DA = 1.94  # r4 = DA
        AB = 6.86  # r3 = AB
        CB = 2.36  # r2 = CB
        DC = 7.00  # r1 = DC

        # Assign constants to variables
        r1 = DC
        r2 = CB
        r3 = AB
        r4 = DA

        # Function definitions for Newton-Raphson
        def f1(theta2, theta3, theta):
            return r2 * np.cos(theta2) + r3 * np.cos(theta3) - r4 * np.cos(theta) - r1

        def f2(theta2, theta3, theta):
            return r2 * np.sin(theta2) + r3 * np.sin(theta3) - r4 * np.sin(theta)

        # Jacobian matrix
        def jacobian(theta2, theta3):
            return np.array([
                [-r2 * np.sin(theta2), -r3 * np.sin(theta3)],
                [r2 * np.cos(theta2),  r3 * np.cos(theta3)]
            ])

        # Newton-Raphson system function
        def newton_system(r1, r2, r3, r4, thetas):
            phi_vals = np.zeros_like(thetas, dtype=float)  # Store phi values (angle of link
                3)
            phi_vals2 = np.zeros_like(thetas, dtype=float)  # Store auxiliary values if needed

            # Iterate over each theta to compute phi
            for theta_deg in thetas:
                i = int(theta_deg)
                theta_rad = np.deg2rad(theta_deg)  # Convert theta to radians for calculations

                # Use previous solution as initial guess
                theta2_old = np.deg2rad(30) if i == 0 else np.deg2rad(phi_vals[i - 1])
                theta3_old = np.deg2rad(0) if i == 0 else np.deg2rad(phi_vals2[i - 1])

                # Newton's Method parameters
                tolerance = 1e-4
                converged = False

                # Newton-Raphson loop
                while not converged:
                    # Compute function values and Jacobian
                    F = np.array([f1(theta2_old, theta3_old, theta_rad), f2(theta2_old,
                        theta3_old, theta_rad)])
                    J = jacobian(theta2_old, theta3_old)

                    # Newton's method update
                    delta = np.linalg.solve(J, F)
                    theta2_new = theta2_old - delta[0]
                    theta3_new = theta3_old - delta[1]

                    # Check convergence
                    if abs(theta2_new - theta2_old) < tolerance and abs(theta3_new -
                        theta3_old) < tolerance:
                        converged = True

```

```
                # Update for next iteration
                theta2_old = theta2_new
                theta3_old = theta3_new

            # Store computed phi (angle of link 3)
            phi_vals[i] = np.rad2deg(theta2_new)
            phi_vals2[i] = np.rad2deg(theta3_new)

        return phi_vals


    thetas = np.arange(0, 361, 1)  # Theta range in degrees
    phi_vals = newton_system(r1, r2, r3, r4, thetas)

    # Compute derivatives (forward and centered difference)
    dphi_dtheta_forward = np.zeros_like(thetas, dtype=float)
    dphi_dtheta_centered = np.zeros_like(thetas, dtype=float)

    # Forward difference for dphi/dtheta
    dphi_dtheta_forward[:-1] = (phi_vals[1:] - phi_vals[:-1]) / np.deg2rad(1)

    # Centered difference for dphi/dtheta
    dphi_dtheta_centered[1:-1] = (phi_vals[2:] - phi_vals[:-2]) / (2 * np.deg2rad(1))

    ## Question 2

    def normalize_angle(angle):
        return angle % 360

    alpha_vals = np.zeros_like(thetas, dtype=float)
    alpha_vals = 149 + phi_vals
    #print(alpha_vals)
    alpha_vals = normalize_angle(alpha_vals)

    r12 = 1.25
    r22 = 1.26
    r32 = 1.87
    r42 = 2.39
    # Function definitions for Newton-Raphson
    def f12(theta2, theta3, theta):
        return r22 * np.cos(theta2) + r32 * np.cos(theta3) - r42 * np.cos(theta) - r12

    def f22(theta2, theta3, theta):
        return r22 * np.sin(theta2) + r32 * np.sin(theta3) - r42 * np.sin(theta)



    # Newton-Raphson system function
    def newton_system2(thetas):
        phi_vals = np.zeros_like(thetas, dtype=float)
        phi_vals2 = np.zeros_like(thetas, dtype=float)
        i = 0
        # Iterate over each theta to compute phi
        for theta_deg in thetas:

            theta_rad = np.deg2rad(theta_deg)  # Convert theta to radians for calculations

            # Use previous solution as initial guess
            theta2_old = np.deg2rad(330) if i == 0 else np.deg2rad(phi_vals[i - 1])
            theta3_old = np.deg2rad(30) if i == 0 else np.deg2rad(phi_vals2[i - 1])

            # Newton's Method parameters
            tolerance = 1e-4
            converged = False

```

```
                # Newton-Raphson loop
                while not converged:
                    # Compute function values and Jacobian
                    F = np.array([f12(theta2_old, theta3_old, theta_rad), f22(theta2_old, theta3_old, theta_rad)])
                    J = jacobian(theta2_old, theta3_old)

                    # Newton's method update
                    delta = np.linalg.solve(J, F)
                    theta2_new = theta2_old - delta[0]
                    theta3_new = theta3_old - delta[1]

                    # Check convergence
                    if abs(theta2_new - theta2_old) < tolerance:
                        converged = True

                    # Update for next iteration
                    theta2_old = theta2_new
                    theta3_old = theta3_new

                # Store computed phi (angle of link 3)
                phi_vals[i] = np.rad2deg(theta2_new)
                phi_vals2[i] = np.rad2deg(theta3_new)
                i = i + 1

            return phi_vals

        def normalize_angle(angle):
            return angle % 360

        beta_vals = newton_system2(alpha_vals)



        beta_vals = normalize_angle(beta_vals)

        # Function for forward and centered differences
        def compute_derivatives(beta_vals, thetas, omega):
            # Forward difference for d  / d
            dbeta_dtheta_forward = np.zeros_like(beta_vals)

            dbeta_dtheta_forward[:-1] = np.deg2rad(beta_vals[1:] - beta_vals[:-1]) / np.deg2rad(1)

            # Centered difference for d  / d
            dbeta_dtheta_centered = np.zeros_like(beta_vals)
            dbeta_dtheta_centered[1:-1] = np.deg2rad(beta_vals[2:] - beta_vals[:-2]) / (2 * np.deg2rad(1))

            # Angular velocity (d  /dt) using both methods
            angular_velocity_forward = omega * dbeta_dtheta_forward
            angular_velocity_centered = omega * dbeta_dtheta_centered

            # Forward difference for d    / d
            d2beta_dtheta2_forward = np.zeros_like(beta_vals)
            d2beta_dtheta2_forward[:-2] = np.deg2rad(beta_vals[2:] - 2 * beta_vals[1:-1] + beta_vals[:-2]) / (np.deg2rad(1) ** 2)

            \# Centered difference for d    / d
            d2beta_dtheta2_centered = np.zeros_like(beta_vals)
            d2beta_dtheta2_centered[1:-1] = np.deg2rad(beta_vals[2:] - 2 * beta_vals[1:-1] + beta_vals[:-2]) / (np.deg2rad(1) ** 2)

            # Angular acceleration $\(d    /dt  ) using both methods
            angular_acceleration_forward = omega ** 2 * d2beta_dtheta2_forward
```

```
            angular_acceleration_centered = omega ** 2 * d2beta_dtheta2_centered

            return angular_velocity_forward, angular_velocity_centered,
                angular_acceleration_forward, angular_acceleration_centered


        angular_velocity_forward, angular_velocity_centered, angular_acceleration_forward,
            angular_acceleration_centered = compute_derivatives(beta_vals, thetas, 7.5)
```