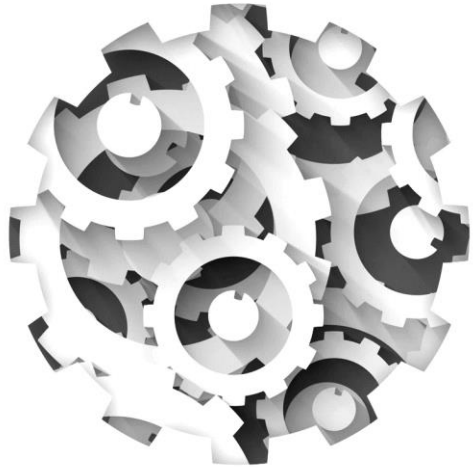
Theory of Mechanism

Complete Kinematic Analysis (Theoretical Basis, Symbolic Computation, Step-by-Step Code Derivation & Implementation, Analysis & Animation)

Dr.Caglar Uyulan



Newton-Raphson Method

The method was pioneered by the eminent mathematician Sir Isaac Newton, and further refined by Joseph Raphson. Their collective contributions have given us a robust tool for root-finding in non-linear equations.
At its core, the Newton-Raphson method is an iterative numerical technique. It's used to find roots of a real-valued function, say $f(x) = 0$, where direct analytical solutions are unfeasible or overly complex. This method stands out for its efficiency and rapid convergence, provided a good initial guess.

The iteration starts with a preliminary guess, x_0, for the root.
Then, the method refines this guess by employing the function's derivative.
The formula used is:

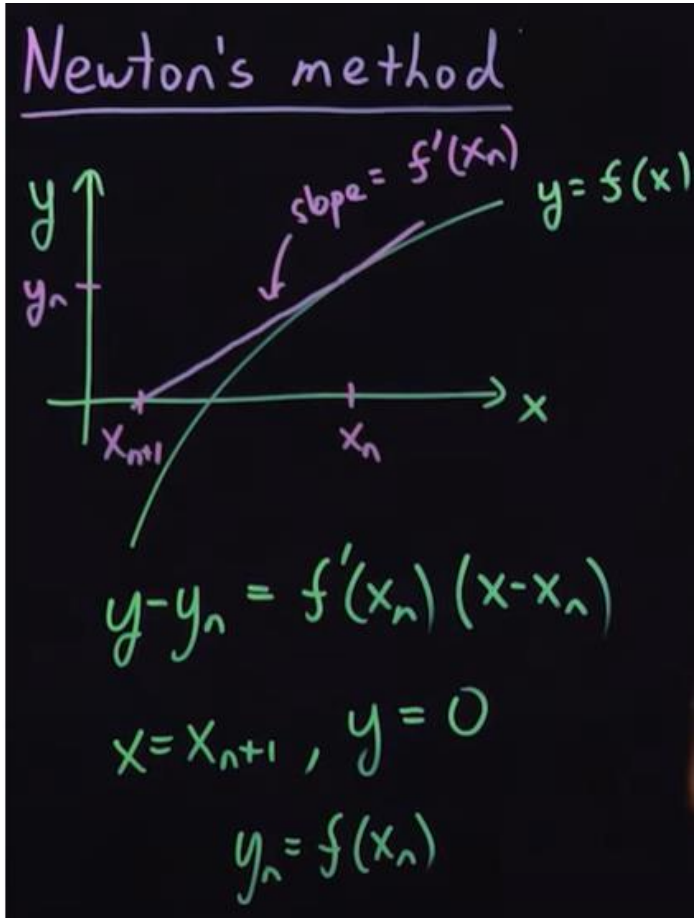
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Here, $f'(x)$ is the derivative of f, providing the slope at any given point on the curve. By repeatedly applying this formula, the Newton-Raphson method converges to the actual root, optimizing the guess in each iteration.

Application in Mechanism Analysis: Consider robotic systems or intricate mechanical components. The Newton-Raphson method aids in solving equations that govern the motion and kinematics of these systems. By iteratively refining positions and velocities of mechanism elements, we can accurately predict and analyze their behavior over time.

Implementation Considerations: The effectiveness of the Newton-Raphson method relies heavily on the initial guess. A closer initial guess to the actual root leads to faster convergence. However, be aware of potential pitfalls. If the initial guess is too far off, or if the function behaves unpredictably near the root (like having a flat slope), the method might diverge or converge slowly.

The Role of Derivatives: The use of the first derivative is critical in this method. It allows the algorithm to linearize the function near the current estimate using the first-order Taylor series expansion. The slope of the tangent at the current estimate, derived from this linearization, guides us toward the root. As we home in on the root, the tangent line increasingly approximates the function's behavior at the root, enhancing the accuracy of subsequent estimates.



$$f'(x_n) = \frac{y_{n+1} - y_n}{x_{n+1} - x_n}$$

$$y_{n+1} - y_n = f'(x_n)(x_{n+1} - x_n)$$

$$0 - f(x_n) = f'(x_n)(x_{n+1} - x_n)$$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

y should be equal to zero, therefore $x(n+1)$ should be iteratively shifted right.

In this case, x=x(n+1), y=0, $y_n = f(x_n)$ are substituted into the derivative equation.

$$error = -\frac{f(x_n)}{f'(x_n)}$$

should be minimized iteratively

Algorithm Flowchart (pseudocode)

Pseudocode Explanation

1. Initialization

- **Start with an initial guess (`x_0`):** This is your starting point. The choice of `x_0` is crucial as it can affect the convergence speed and success of the method.

2. Iteration Process

- **Steps 2-4 (First Iteration):**
  - **Calculate the first derivative (`f'(x_0)`):** Determine the slope of the function at the initial guess. This step is foundational as the derivative guides the iteration.
  - **Calculate the slope of the tangent line at `x_0`:** This is implicit in the Newton-Raphson formula. The slope is essentially `f'(x_0)`.
  - **Improve the guess (`x_1`):** Apply the formula $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$. This step adjusts the initial guess closer to the root using the tangent line's slope.
- **Steps 5-7 (Subsequent Iterations):**
  - These steps are a repetition of the first iteration, but starting from the new guess (`x_1`), then `x_2`, and so on.
  - In each iteration, the function's derivative at the new guess is used to find the next, more accurate guess.
  - The process continues, iteratively refining the guess.

3. Convergence Check

- **Step 6:** After each iteration, check if the current guess is close enough to the root. This is typically done by checking if the difference between successive guesses (e.g., `x_1` and `x_0`) is below a pre-defined threshold, indicating the desired level of accuracy.

4. Repeat or Terminate

- **Step 7:** If the desired accuracy is not achieved, set the new guess as the starting point for the next iteration (i.e., `x_0 = x_1`) and repeat the process.
- **End:** Once the root is found to the desired accuracy, the process stops.

Flowchart Overview

The flowchart provides a visual representation of the iterative process:

1. **Start:** Begin the algorithm.
2. **Initialize `x_0`:** Set the initial guess.
3. **Calculate `f'(x_0)`:** Find the derivative at `x_0`.
4. **Calculate `x_1`:** Use the Newton-Raphson formula.
5. **Accuracy Check:** Determine if `x_1` is accurate enough.
6. **Iteration:** If not accurate enough, set `x_0 = x_1` and return to step 3.
7. **End:** Stop the process when the root is found to the desired accuracy.

The pseudocode and flowchart together provide both a conceptual and practical framework for implementing the Newton-Raphson method. They underscore the method's iterative nature and the critical role of derivatives in guiding each step towards the accurate root. This method, as you can see, elegantly balances simplicity and mathematical sophistication, making it a valuable tool in engineering and scientific computations.

NewtonRaphson_optimized.m

Overall Structure

The script applies the Newton-Raphson method to find the roots of a system of nonlinear equations. It defines the system, its Jacobian matrix, and then iteratively refines an initial guess until it converges to a solution or reaches a maximum number of iterations.

Detailed Explanation

1. **Defining the System of Equations (`fn`):**
   - `fn = @(v) [v(1)^2+v(2)^2-2*v(3); v(1)^2+v(3)^2-(1/3); v(1)^2+v(2)^2+v(3)^2-1];`
   - This line defines a function handle `fn`, representing a system of three nonlinear equations with variables `v(1)`, `v(2)`, and `v(3)`.
   - The system is defined as:
     - Equation 1: $v_1^2 + v_2^2 - 2v_3 = 0$
     - Equation 2: $v_1^2 + v_3^2 - \frac{1}{3} = 0$
     - Equation 3: $v_1^2 + v_2^2 + v_3^2 - 1 = 0$

2. **Defining the Jacobian Matrix (`jacob_fn`):**
   - `jacob_fn = @(v) [2*v(1) 2*v(2) -2; 2*v(1) 0 2*v(3); 2*v(1) 2*v(2) 2*v(3)];`
   - This line defines another function handle `jacob_fn`, which calculates the Jacobian matrix of the system of equations.
   - The Jacobian matrix for this system is:
     - $\begin{bmatrix} 2v_1 & 2v_2 & -2 \\ 2v_1 & 0 & 2v_3 \\ 2v_1 & 2v_2 & 2v_3 \end{bmatrix}$
3. **Initialization:**
   - `tolerance = 10^-5;` sets the tolerance level for the convergence check.
   - `v = [1; 1; 0.1];` initializes the vector `v` with an initial guess for the root.
   - `max_iterations = 20;` sets the maximum number of iterations.
4. **Executing the Newton-Raphson Method:**
   - `[v_final, iterations, final_error] = NewtonRaphson_nl(v, fn, jacob_fn, max_iterations, tolerance);`
   - This line calls the function `NewtonRaphson_nl` with the initial guess, the function, its Jacobian, the maximum iterations, and the tolerance.
   - It returns the final approximation of the root (`v_final`), the number of iterations performed (`iterations`), and the final error (`final_error`).

5. **Printing the Results:**
   - The `fprintf` statement prints the final results to the console.
6. **The `NewtonRaphson_nl` Function:**
   - The function `NewtonRaphson_nl` implements the actual Newton-Raphson iteration.
   - **Input Argument Handling:**
     - Sets default values for `tolerance` and `max_iterations` if they are not provided.
   - **Initial Setup:**
     - Initializes `v_final` with the initial guess and calculates the function value at this guess.
   - **Iteration Loop:**
     - Within a `while` loop, it calculates the Jacobian matrix at the current estimate (`v_final`).
     - Solves the linear system to find `delta` (correction term).
     - Updates the guess (`v_final = v_final - delta`).
     - Updates the function value and the error norm (`final_norm`).
   - **Convergence Check:**
     - If the error norm is less than the tolerance or the number of iterations reaches the maximum limit, it breaks the loop.
   - **Returning Results:**
     - Finally, it returns the last estimate of the root, the total number of iterations, and the final error norm.

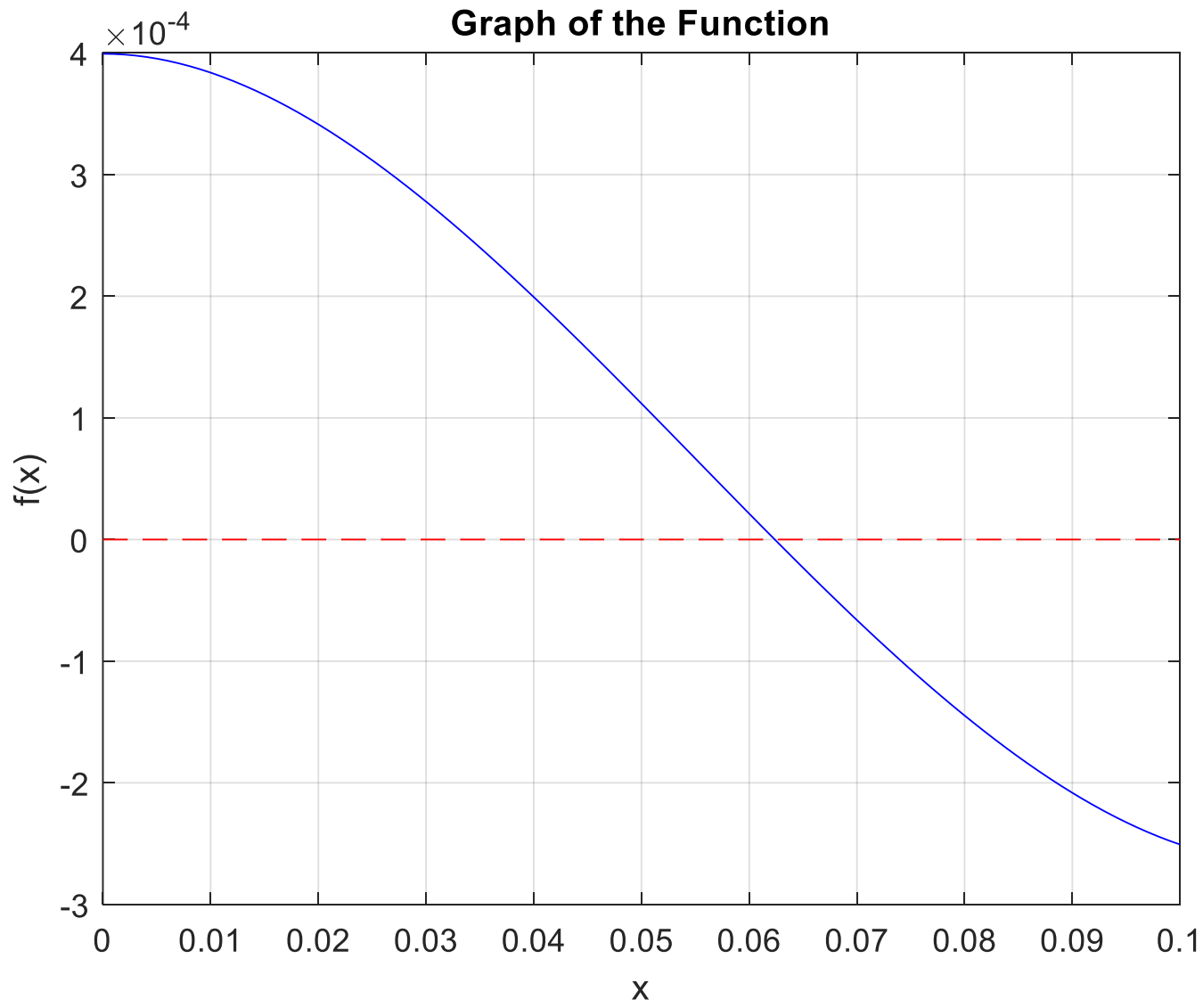
This script effectively implements the Newton-Raphson method for solving a system of nonlinear equations, providing a practical example of numerical methods in action.

NewtonRaphson_alternative.m

1. **Environment Setup:**
   - `clc;`: Clears the command window.
   - `close all;`: Closes all open figures or windows.
   - `clear all;`: Clears all variables from the workspace.
2. **Defining the Function and Its Derivative:**
   - `syms x;`: Declares `x` as a symbolic variable.
   - `f = x^3 - 0.165*x^2 + 3.993e-4;`: Defines the function $f(x) = 0.165x^2 + 3.993 \times 10^{-4}$.
   - `g = diff(f);`: Computes the derivative of `f` with respect to `x`, $g(x) = 3x^2 - 0.33x$.
3. **Setting Precision and Initial Approximation:**
   - `n = 5;`: Sets the number of decimal places to 5.
   - `epsilon = 5*10^-(n+1);`: Defines the tolerance (`epsilon`) for th Raphson method as 5×10^{-6}, based on the desired precision.
   - `x0 = 0.05;`: Initializes the starting point (`x0`) for the Newton-Rap iteration as 0.05.

4. **Newton-Raphson Iteration:**
   - A `while true` loop is used for iteration.
   - `f0 = vpa(subs(f, x, x0));`: Calculates the value of `f` at `x0` using symbolic substitution and evaluates it to a floating-point number with high precision.
   - `f0_der = vpa(subs(g, x, x0));`: Similarly, calculates the derivative of `f` at `x0`.
   - `x_new = x0 - f0/f0_der;`: Applies the Newton-Raphson formula to find the next approximation of the root.
   - `err = abs(x_new - x0);`: Computes the absolute error between the new and old approximations.
   - The `if` condition checks if the error is less than `epsilon` (the tolerance). If so, it breaks the loop.
   - `x0 = x_new;`: Updates the approximation for the next iteration.
   - `iter = iter + 1;`: Increments the iteration counter.
   - If `iter > 100`, the loop breaks, preventing infinite iterations.
5. **Final Root and Iteration Display:**
   - `x_new = x_new - rem(x_new, 10^-n);`: Adjusts `x_new` to the specified number of decimal places.
   - `fprintf` statements: Display the root found and the number of iterations taken.
6. **Plotting the Function:**
   - `x_range = linspace(0, 0.1);`: Creates a range of `x` values from 0 to 0.1.
   - `f_vals = x_range.^3 - 0.165*x_range.^2 + 3.993*10^-4;`: Calculates `f(x)` for each value in `x_range`.
   - `figure;`: Creates a new figure.
   - `plot(x_range, f_vals, 'b', x_range, zeros(size(x_range)), 'r--');`: Plots `f(x)` in blue and the x-axis (y=0) in red dashed line.
   - `grid on;`: Adds a grid to the plot.
   - `title`, `xlabel`, and `ylabel`: Label the plot and its axes.

The script effectively combines numerical methods for root finding with symbolic computation and graphical representation, illustrating the versatility of MATLAB in mathematical computing.


Graph of the Function
×10^-4
4
3
2
1
0
-1
-2
-3
f(x)
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
x

Implementation to the Kinematic Analysis of the Mechanisms

- A MATLAB code snippet that outlines a numerical method, likely the Newton-Raphson method, for analyzing the motion of a mechanical system, should be constructed.
- It's structured to iterate and refine guesses of the positions and velocities of the system's components.

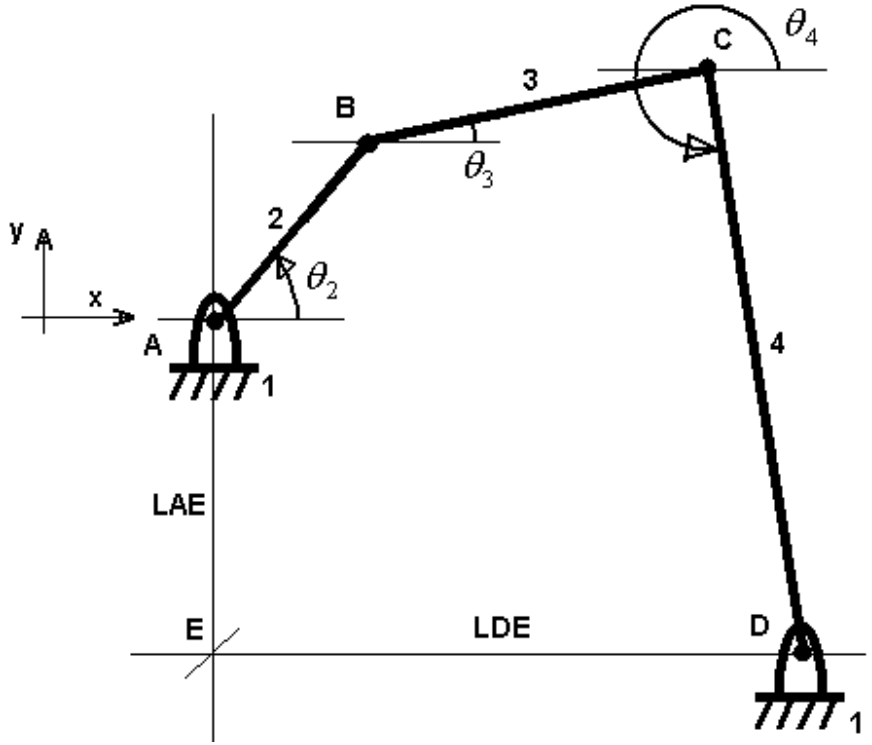
Let's break down the abstraction of the code:

1. **Defining a Function for Motion Equations:**
   - The function `motion_eqns(guess)` represents the equations of motion for the mechanism.
   - The input `guess` is expected to be a vector containing the current guesses for the positions and velocities of the mechanism's components.
   - The function calculates and returns `derivatives`, which are the derivatives of these positions and velocities. The specific calculation depends on the mechanism's dynamics and kinematics.
2. **Initial Guess:**
   - `guess = [...];`: This line initializes the `guess` vector. The actual values should be filled in depending on the specific mechanism and the initial conditions or assumptions about the system's state.
3. **Setting Iteration Parameters:**
   - `max_iterations = 100;`: Defines the maximum number of iterations for the algorithm to prevent infinite loops.
   - `tolerance = 1e-6;`: Sets a tolerance level for convergence. The iteration will stop if the norm of the `derivatives` is less than this value, indicating that the solution has sufficiently converged.
4. **Newton-Raphson Iteration for Position Analysis:**
   - The `for` loop runs up to `max_iterations` times, implementing the Newton-Raphson method.
   - Inside the loop:
     - `derivatives = motion_eqns(guess);`: Calculates the `derivatives` of positions and velocities based on the current `guess`.
     - `guess = guess + derivatives;`: Updates the `guess` by adding the `derivatives`. This step is crucial in the Newton-Raphson method, where the guess is iteratively refined.
     - The `if` statement checks if the norm of `derivatives` is less than the `tolerance`. If true, it indicates convergence, and the loop is exited.

5. **Displaying the Final Solution:**
    * `disp(guess);`: Displays the final values of the `guess` vector, which represents the positions and velocities of the mechanism's components at convergence.

In summary, this code is set up for a numerical analysis of a mechanical system's motion, specifically using the Newton-Raphson method for iteratively finding the positions and velocities of the system's components. The `motion_eqns` function and the initial `guess` would need to be defined based on the specific mechanical system being analyzed.

FOUR BAR MECHANISM: KINEMATIC ANALYSIS



AB= 0.3m, BC= 0.45m, CD= 0.7m
DE= 0.75m, AE= 0.35m.

$\theta_2 = 0 \rightarrow 360^o$

$\theta_3, \theta_4 = ?$

Position Analysis

$$f_1 = l_2 \cos\theta_2 + l_3 \cos\theta_3 + l_4 \cos\theta_4 - LDE = 0 \quad (1)$$

$$f_2 = l_2 \sin\theta_2 + l_3 \sin\theta_3 + l_4 \sin\theta_4 + LAE = 0 \quad (2)$$

Jacobian Matrix

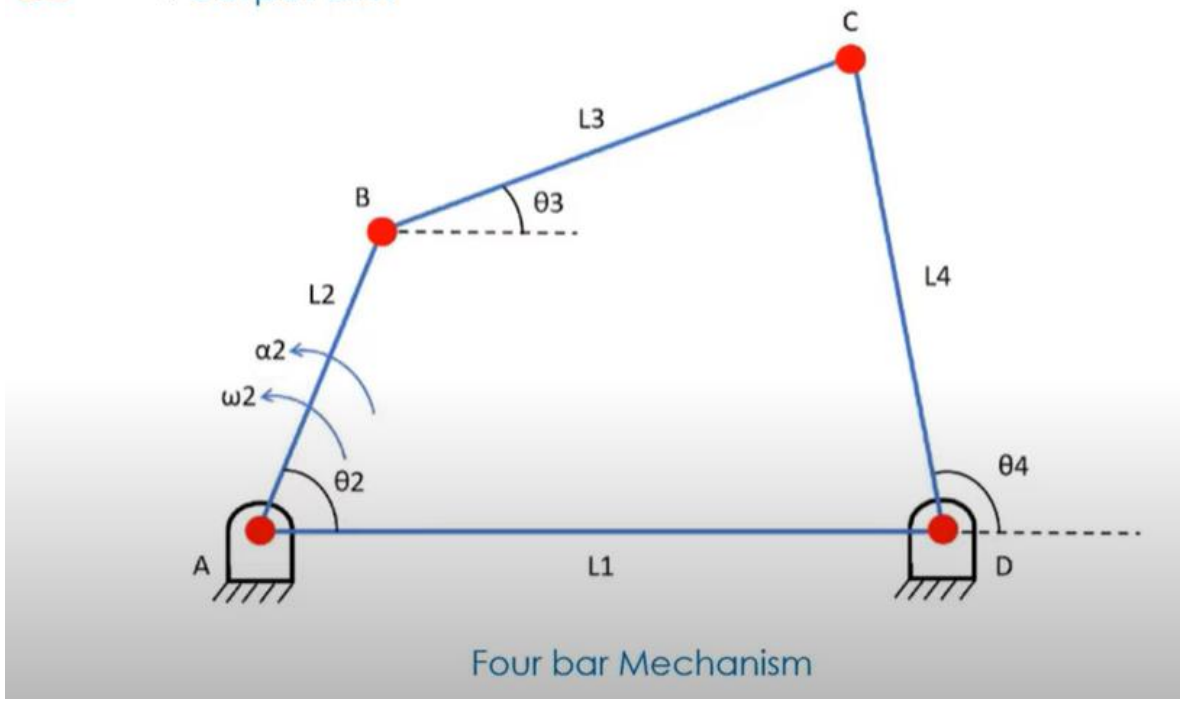
$$\begin{bmatrix} \partial f_1/\partial\theta_1 & \partial f_1/\partial\theta_2 \\ \partial f_2/\partial\theta_1 & \partial f_2/\partial\theta_2 \end{bmatrix}$$

Matrix Form: (for Newton-Raphson code)

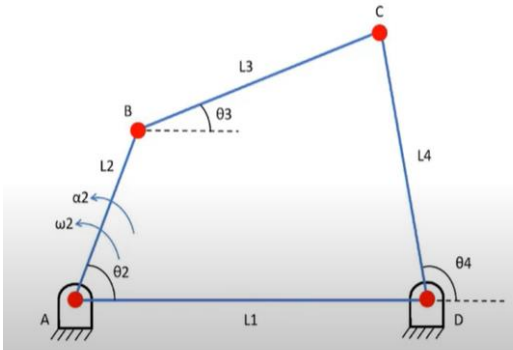
$$\begin{bmatrix} -l_3 \sin\theta_3 & -l_4 \sin\theta_4 \\ l_3 \cos\theta_3 & l_4 \cos\theta_4 \end{bmatrix} \begin{Bmatrix} \varepsilon_1 \\ \varepsilon_2 \end{Bmatrix} = \begin{Bmatrix} -l_2 \cos\theta_2 - l_3 \cos\theta_3 - l_4 \cos\theta_4 + LDE \\ -l_2 \sin\theta_2 - l_3 \sin\theta_3 - l_4 \sin\theta_4 - LAE \end{Bmatrix}$$

https://www.youtube.com/watch?v=04CsSL8Y338 (**Position, Velocity, Acceleration Analysis of Four bar Mechanism in MATLAB**)

AD : Fixed Link
AB : Input Link
BC : Coupler
CD : Output Link



Inputs : $L_1, L_2, L_3, L_4, \theta_2$
Outputs : θ_3, θ_4



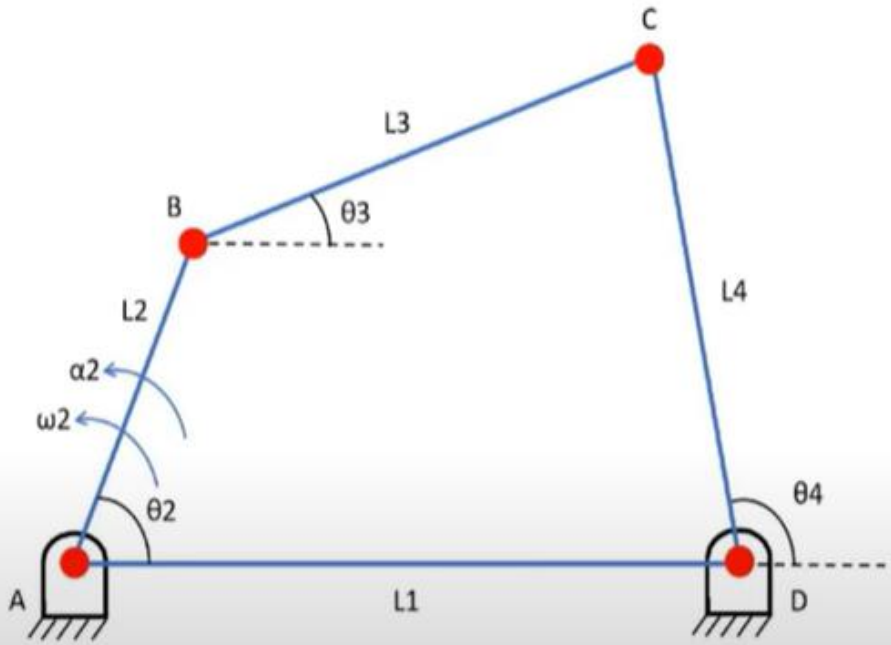
Loop Closure Method

$$-\ell_1\hat{\mathbf{i}}$$

$$\ell_2\cos\theta_2\hat{\mathbf{i}} + \ell_2\sin\theta_2\hat{\mathbf{j}}$$

$$\ell_3\cos\theta_3\hat{\mathbf{i}} + \ell_3\sin\theta_3\hat{\mathbf{j}}$$

$$\ell_4\cos\left(180^0+\theta_4\right)\hat{\mathbf{i}} + \ell_4\sin\left(180^0+\theta_4\right)\hat{\mathbf{j}}$$



i-components :

$$f_1 = -L_1 + L_2\cos\theta_2 + L_3\cos\theta_3 + L_4\cos(180 + \theta_4) = 0$$

j-components :

$$f_2 = L_2\sin\theta_2 + L_3\sin\theta_3 + L_4\sin(180 + \theta_4) = 0$$

Newton-Raphson Method

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

$$f'(x_n)\,\Delta x = -f(x_n)$$

$$\begin{bmatrix} \frac{\partial f_1(x_1,x_2)}{\partial x_1} & \frac{\partial f_1(x_1,x_2)}{\partial x_2} \\ \frac{\partial f_2(x_1,x_2)}{\partial x_1} & \frac{\partial f_2(x_1,x_2)}{\partial x_2} \end{bmatrix} \begin{Bmatrix} \Delta x_1 \\ \Delta x_2 \end{Bmatrix} = -\begin{Bmatrix} f_1(x_1,x_2) \\ f_2(x_1,x_2) \end{Bmatrix}$$

Newton-Raphson Method

$$\begin{bmatrix} \frac{\partial f_1}{\partial \theta_3} & \frac{\partial f_1}{\partial \theta_4} \\ \frac{\partial f_2}{\partial \theta_3} & \frac{\partial f_2}{\partial \theta_4} \end{bmatrix} \begin{Bmatrix} \Delta\theta_3 \\ \Delta\theta_4 \end{Bmatrix} = -\begin{Bmatrix} f_1(\theta_3,\theta_4) \\ f_2(\theta_3,\theta_4) \end{Bmatrix}$$

$$\theta_3^{n+1} = \theta_3^n + \Delta\theta_3$$
$$\theta_4^{n+1} = \theta_4^n + \Delta\theta_4$$

$$\begin{bmatrix} -\ell_3\sin\theta_3 & -\ell_4\sin(180^0+\theta_4) \\ \ell_3\cos\theta_3 & \ell_4\cos(180^0+\theta_4) \end{bmatrix} \begin{Bmatrix} \Delta\theta_3 \\ \Delta\theta_4 \end{Bmatrix} = -\begin{Bmatrix} \begin{pmatrix} -\ell_1+\ell_2\cos\theta_2+\ell_3\cos\theta_3 \\ +\ell_4\cos(180^0+\theta_4) \end{pmatrix} \\ \ell_2\sin\theta_2+\ell_3\sin\theta_3 \\ +\ell_4\sin(180^0+\theta_4) \end{Bmatrix}$$

Velocity + Acceleration Analysis

$\omega_3, \omega_4, \alpha_3, \alpha_4 = ?$

fourbarmechanism_kinematicanalysis.m

Let's break down the provided MATLAB code, which is used for analyzing a four-bar mechanism, step by step:

Initial Setup for Symbolic Computation (Commented Out)

- This part of the code is commented out, but it sets up symbolic variables for a four-bar mechanism, defining the angular positions `th2(t)`, `th3(t)`, and `th4(t)`, along with the lengths of the arms `l2`, `l3`, `l4`, `lde`, and `lae`.
- It defines two constraint equations (`f1` and `f2`) for the mechanism and calculates their first (`fv1`, `fv2`) and second derivatives (`fa1`, `fa2`) with respect to time. These derivatives represent velocity (`fv1`, `fv2`) and acceleration (`fa1`, `fa2`) equations.

matlab

```
%syms th2(t) th3(t) th4(t) l2 l3 l4 lde lae
%f1=l2*cos(th2)+l3*cos(th3)+l4*cos(th4)-lde==0;
%f2=l2*sin(th2)+l3*sin(th3)+l4*sin(th4)+lae==0;
%fv1=diff2(f1,t);
%fv2=diff2(f2,t);
%fa1=diff2(diff2(f1,t),t);
%fa2=diff2(diff2(f2,t),t);
```

Physical Parameters

```matlab
L2 = 0.3;  % m, length of arm 2
L3 = 0.45; % m, length of arm 3
L4 = 0.7;  % m, length of arm 4
LDE = 0.75; % m
LAE = 0.35; % m
```

- Sets the physical lengths of the mechanism's components.

Simulation Parameters

```matlab
Nmax = 100; % Maximum number of iterations
x = [20*pi/180, 320*pi/180]; % Initial guess for th3, th4 (in radians)
xe = 0.0001 * abs(x); % Slightly relaxed error tolerance
dth = 5 * pi / 180; % Step size for th2
th2 = 0:dth:2*pi; % Range of th2 (in radians)
w2 = -0.2944 * ones(size(th2)); % Constant angular velocity w2
acc2 = zeros(size(th2)); % Constant angular acceleration acc2
```

- Sets up parameters for the numerical iteration: maximum iterations, initial guesses for angles `th3` and `th4`, error tolerance, step size, and the range for angle `th2`. It also defines a constant angular velocity (`w2`) and acceleration (`acc2`) for `th2`.

Preallocation of Result Arrays

```matlab
th3 = zeros(size(th2));
th4 = zeros(size(th2));
w3 = zeros(size(th2));
w4 = zeros(size(th2));
acc3 = zeros(size(th2));
acc4 = zeros(size(th2));
```

- Preallocates arrays to store the calculated values for angles, angular velocities, and accelerations of `th3` and `th4`.

Main Loop

```matlab
for k = 1:length(th2)
    ...
end
```

- Iterates over each value of `th2` to calculate the corresponding states of `th3`, `th4`, their velocities, and accelerations.

Inside the Main Loop

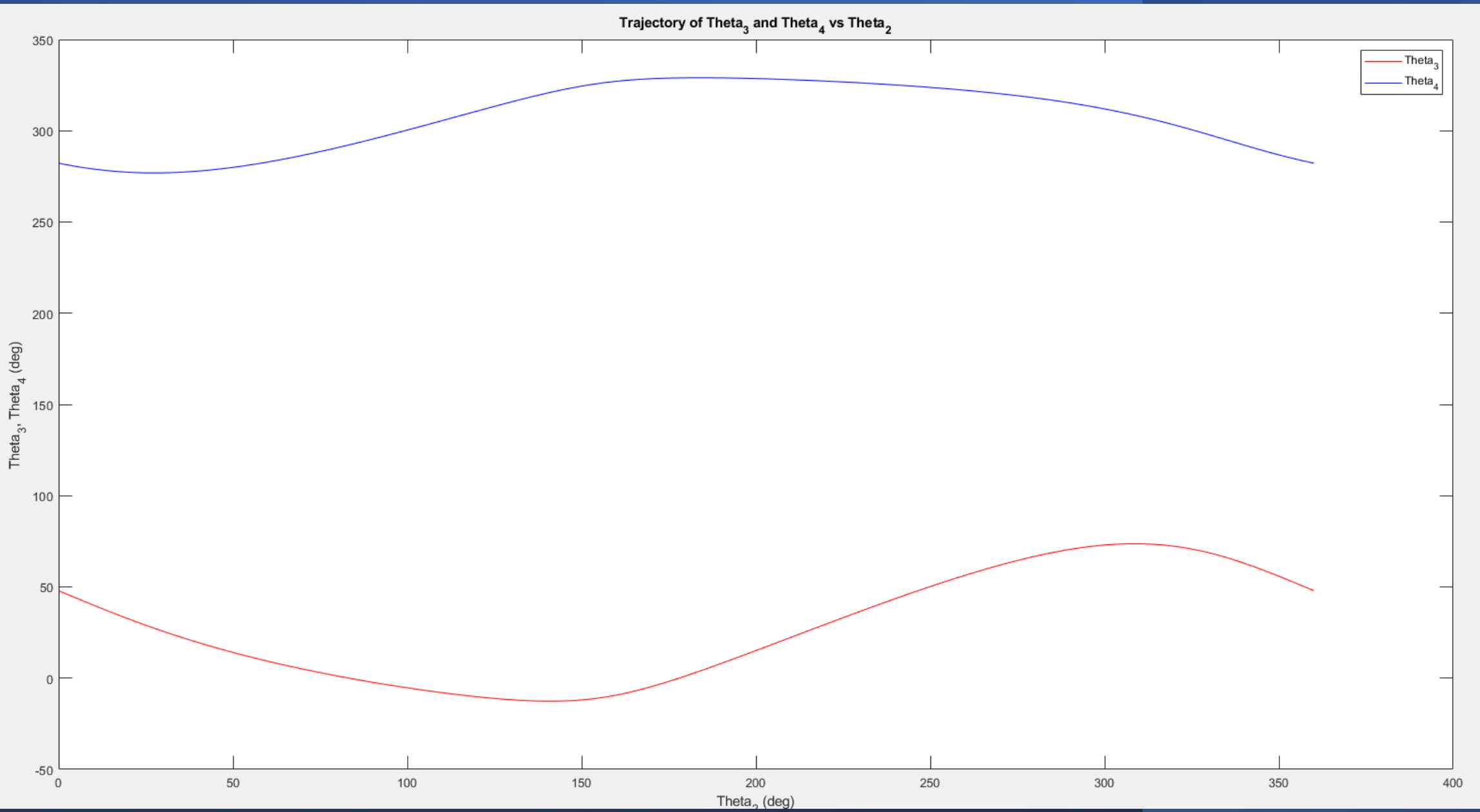
Position Analysis

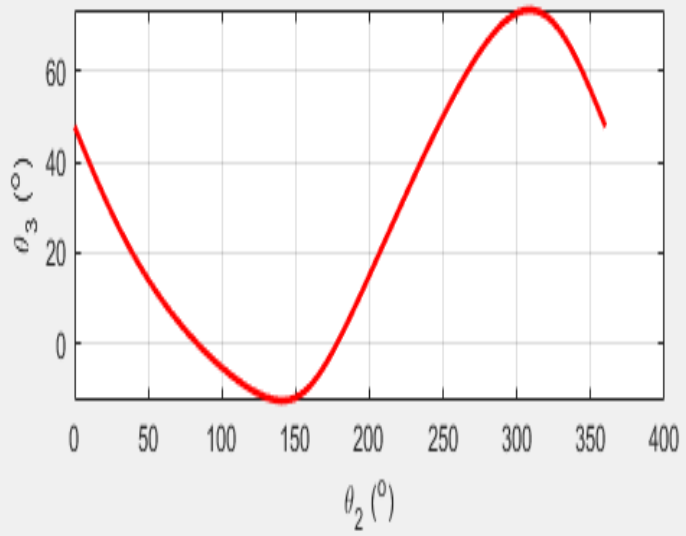
```matlab
for n = 1:Nmax
    ...
end
```

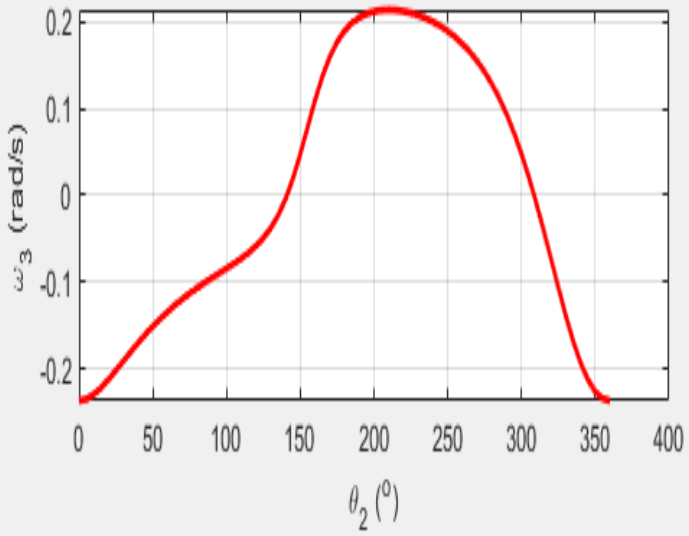
- Performs an iterative solver (Newton-Raphson method) to find the angles `th3` and `th4` that satisfy the mechanism's constraints for each `th2`. It uses a Jacobian matrix `J` and a function vector `f` representing the constraints.

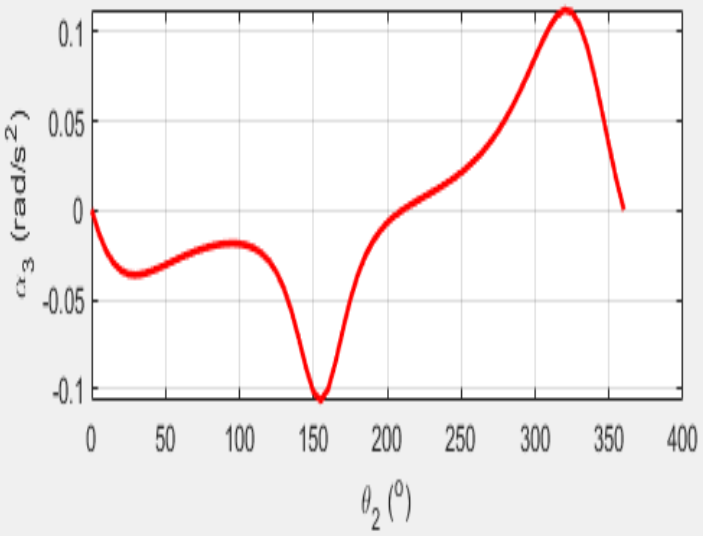
Velocity and Acceleration Analysis

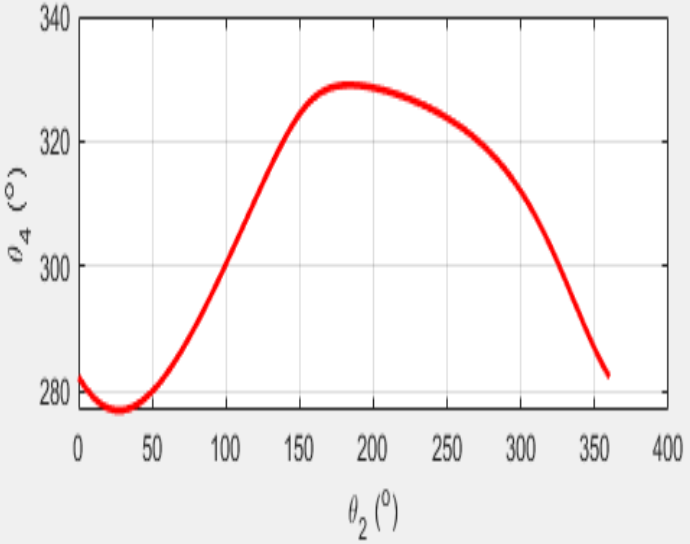
```matlab
fv = [-L2*sin(th2(k))*w2(k); L2*cos(th2(k))*w2(k)];
vel = J\fv;
w3(k) = vel(1);
w4(k) = vel(2);

fa = [-L2*cos(th2(k))*(w2(k))^2 - L3*cos(th3(k))*(w3(k))^2 - L4*cos(th4
      L2*cos(th2(k))*acc2(k) - L3*sin(th3(k))*(w3(k))^
```

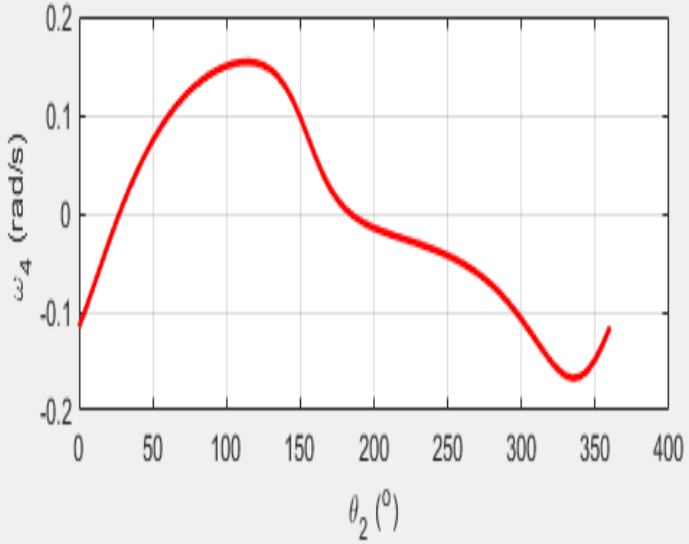

Trajectory of Theta3 and Theta4 vs Theta2
Theta3
Theta4
Theta3, Theta4 (deg)
Theta2 (deg)
350
300
250
200
150
100
50
0
-50
0
50
100
150
200
250
300
350
400

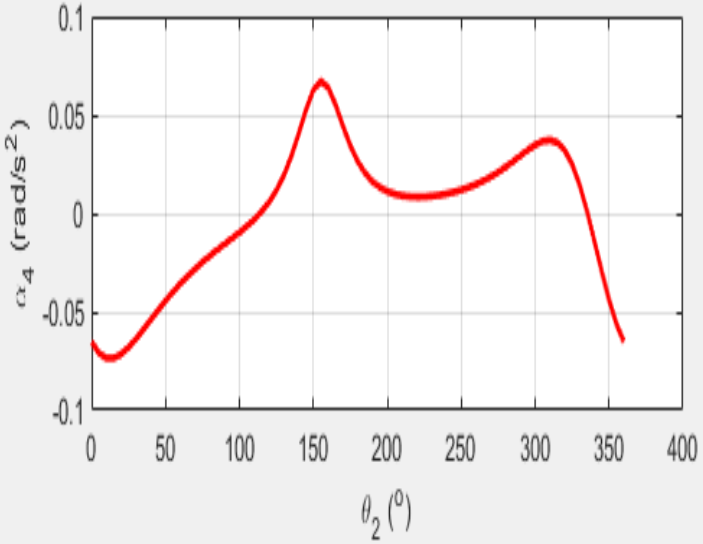

θ_3 (°)
θ_2 (°)


ω_3 (rad/s)
θ_2 (°)

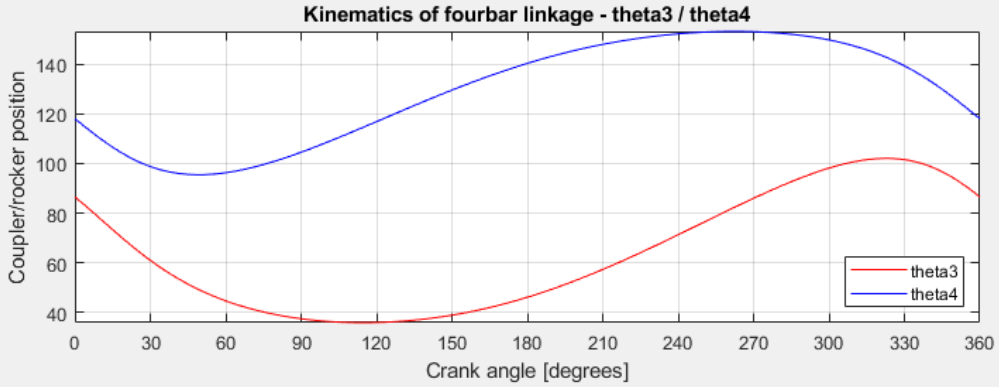

α_3 (rad/s^2)
θ_2 (°)

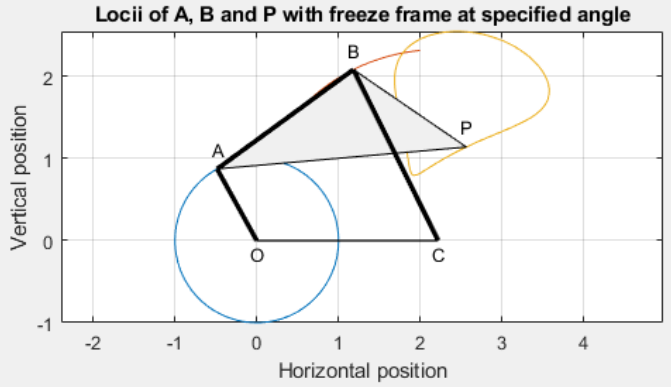

θ_4 (°)
θ_2 (°)

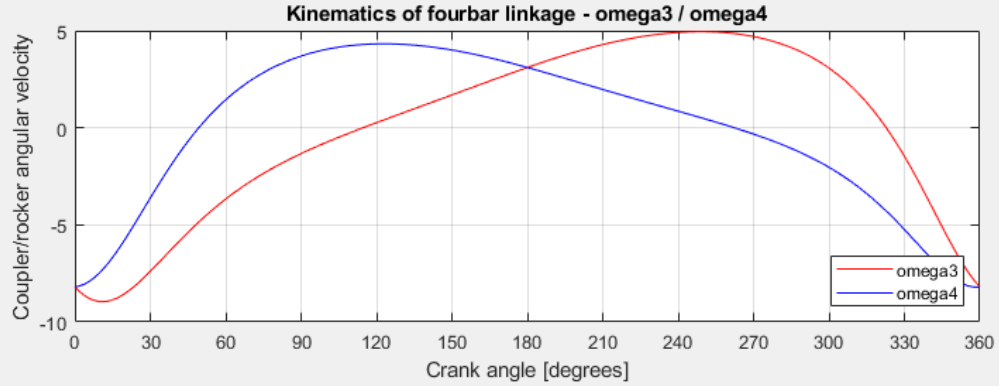

ω_4 (rad/s)
θ_2 (°)

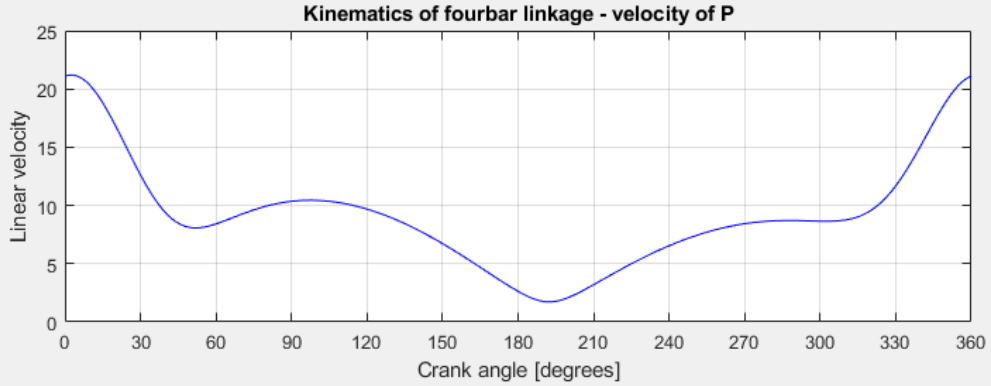

α_4 (rad/s^2)
θ_2 (°)

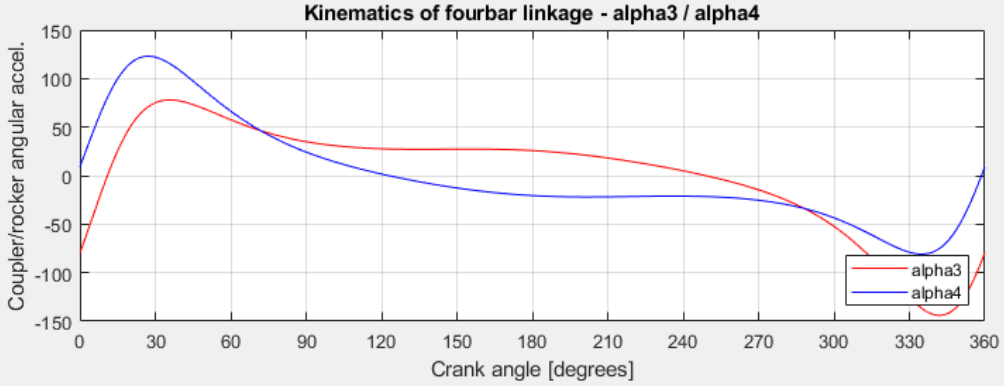
crankrocker_completekinematic.m

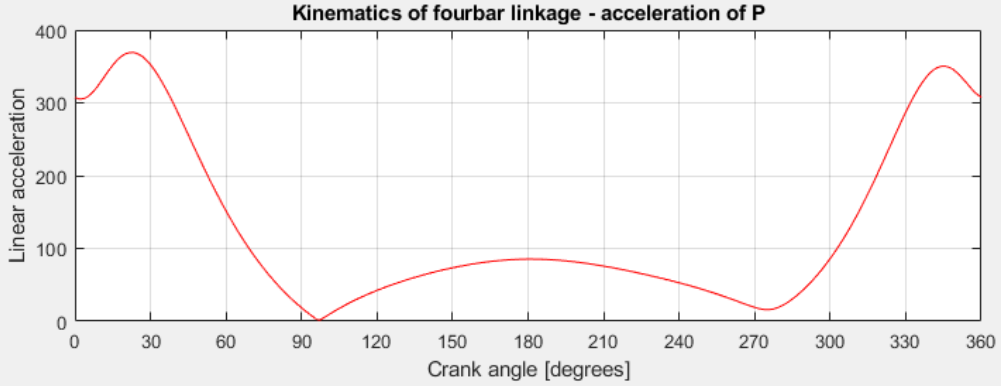












crankconnectingrod_completekinematic.m

Animation