Sakib Azgar
ME4565 CFD

02/05/24
Prof Allshouse

ME 4565: Introduction to Computational Fluid Dynamics

Spring 2024

Problem Set #~~01~~

Due ~~January 31st~~, 2~~024~~ (Submit to Canvas)
February 9th, 5:00 PM

Problem 1 - First Order ODE: For the equation

$$\frac{dx}{dt} = x + t$$

a. Derive the analytic solution using separation of variables. Use the initial condition that $x(0) = 1$. Note: This is an example of a non-homogeneous ODE. You will need to determine both the general and particular solution

(Hand derivation)

$\frac{dx}{dt} = x + t \rightarrow \frac{dx}{dt} + p(t)x = Q(t)$ ($p(t) = -1$, $Q(t) = t$) need integrating factor

$e^{\int p(t)dt} = e^{\int -1\,dt} = e^{-t}$

$\left(\frac{dx}{dt} - x = t\right) e^{-t}$

u	dv
+ t	e^{-t}
− 1	$-e^{-t}$
+ 0	e^{-t}

$e^{-t}\frac{dx}{dt} - e^{-t}x = te^{-t}$

$\int (e^{-t}x)' = \int te^{-t}$

$e^{-t}x = -te^{-t} - e^{-t} + C$

$e^{-t}x = -e^{-t}(t+1) + C$

$x(t) = -t - 1 + \frac{C}{e^{-t}} \rightarrow x(t) = -t - 1 + Ce^{t}$

$1 = -0 - 1 + Ce^{0}$

$1 = -1 + C \quad C = 2$

$$\boxed{x(t) = -(t+1) + 2e^{t}}$$

b. Derive the explicit and implicit Euler method equations.

(Hand derivation)

Explicit

$x_{n+1} = x_n + \Delta t \cdot f(t_n, x_n)$

$x_{n+1} = x_n + \Delta t (x_n + t_n)$

$x_{n+1} = x_n + \Delta t x_n + \Delta t t_n$

$$\boxed{x_{n+1} = x_n(1 + \Delta t) + t_n \Delta t}$$

Implicit

$x_{n+1} = x_n + \Delta t \cdot f(t_{n+1}, x_{n+1})$

$x_{n+1} = x_n + \Delta t \cdot (x_{n+1} + t_{n+1})$

$x_{n+1} = x_n + \Delta t x_{n+1} + \Delta t t_{n+1}$

$x_{n+1} - \Delta t x_{n+1} = x_n + \Delta t t_{n+1}$

$x_{n+1}(1 - \Delta t) = x_n + \Delta t (t_{n+1})$

$$\boxed{x_{n+1} = \frac{x_n + \Delta t (t_{n+1})}{1 - \Delta t}}$$

Problem 1c) Write a function that performs the explicit Euler method to output the approximate the value of x(t=1), where the time step Δt is the input to the function.

Problem 1C Function Script (explicit_euler):

```
% Homework 4 Problem 1 Part C
% Define a function to perform the explicit Euler method
function HW4P1C = explicit_euler(delta_t)

    % Initialize time and x variables
    t = 0; % Start time is set to 0
    x = 1; % Initial condition x(0) = 1 as given

    % Iterate until t reaches or exceeds 1
    while t < 1
        t_next = t + delta_t; % Calculate tentative next time to check for overshoot
        if t_next > 1 % Check if the next time step goes past t=1
            delta_t = 1 - t;  % Adjust delta_t for the final step to end exactly at
t=1
        end

        % Update x using the explicit Euler formula: x_n+1 = x_n + delta_t * f(t_n,
x_n)
        x = x + delta_t * (x + t); % Here, f(t_n, x_n) = x_n + t_n
        t = t + delta_t; % Update the time by the adjusted delta_t
                         % Increment current time by the time step delta_t
    end
    HW4P1C = x;  % Output the approximate value of x at t=1
end
```

Problem 1d) Write a function that performs the implicit Euler method to output the approximate the value of x(t=1), where the time step Δt is the input to the function.

Problem 1D Function Script (implicit_euler):

```
% Homework 4 Problem 1 Part D
% Define a function to perform the implicit Euler method
function HW4P1D = implicit_euler(delta_t)

    % Initialize time and x variables
    t = 0; % Start time is set to 0
    x = 1; % Initial condition x(0) = 1 as given

    % Iterate until t reaches or exceeds 1
    while t < 1 - delta_t % iterate up to the step before t=1
        t_next = t + delta_t;   % Update time to the next step
        x = (x + delta_t * t_next) / (1 - delta_t); % Update x using the implicit
Euler formula
        t = t_next;   % Increment the current time by the time step delta_t
    end

    if t < 1     % Handle the last step to reach exactly t=1
```

```
        delta_t_last = 1 - t; % Calculate the remaining time step to reach exactly
t=1

        % Update x for the last time step using the implicit Euler formula
        x = (x + delta_t_last * (t + delta_t_last)) / (1 - delta_t_last);
    end

    % Output the approximate value of x at t=1
    HW4P1D = x;
end
```

Problem 1e) Write a script that calls your functions for Δt = 0.05 and 0.005. Calculate the error of the two methods for both time steps. Based on these errors what is the order of the truncation error and why?

```
>> HW4P1E
Explicit Euler Method Errors:
    0.1300
    0.0135

Implicit Euler Method Errors:
    0.1425
    0.0137

Estimated order of truncation error for Explicit Euler: 0.982558
Estimated order of truncation error for Implicit Euler: 1.018421
>>
```

Problem 1E Script (HW4P1E):

```
% Homework 4 Problem 1 Part E
% Define the analytical solution
analytical_solution = @(t) -(t + 1) + 2 * exp(t);
time_steps = [0.05, 0.005]; % Time steps to use

% Initialize error storage
errors_explicit = zeros(length(time_steps), 1);
errors_implicit = zeros(length(time_steps), 1);

% Loop through each time step
for i = 1:length(time_steps)
    delta_t = time_steps(i);

    % Get the numerical solutions from both methods
    x_explicit = explicit_euler(delta_t);
    x_implicit = implicit_euler(delta_t);

    % Get the analytical solution at t=1
    x_analytical = analytical_solution(1);

    % Calculate the errors for both methods
    errors_explicit(i) = abs(x_explicit - x_analytical);
    errors_implicit(i) = abs(x_implicit - x_analytical);
```

```
end

% Display the errors on Command Windows
disp('Explicit Euler Method Errors:');
disp(errors_explicit);
disp('Implicit Euler Method Errors:');
disp(errors_implicit);

% Assuming that the error scales with delta_t to the power of p,
% we can estimate p using the ratio of errors for different delta_ts
p_explicit = log(errors_explicit(1) / errors_explicit(2)) / log(time_steps(1) /
time_steps(2));
p_implicit = log(errors_implicit(1) / errors_implicit(2)) / log(time_steps(1) /
time_steps(2));

% Display the estimated order of the truncation error on Command Window
fprintf('Estimated order of truncation error for Explicit Euler: %f\n', p_explicit);
fprintf('Estimated order of truncation error for Implicit Euler: %f\n', p_implicit);
```

Comments:

The computed errors from the explicit and implicit Euler methods for two different step sizes indicate that both methods exhibit first-order truncation error. Specifically, the explicit Euler method error decreased from 0.1300 to 0.0135, while the implicit Euler method error decreased from 0.1425 to 0.0137 as the step size was reduced by a factor of 10. The estimated orders of truncation error are approximately 0.982558 for the explicit method and 1.018421 for the implicit method, both values being close to 1. This near-unitary ratio signifies that the error is roughly proportional to the step size, consistent with the first-order accuracy typical of Euler methods. Deviations from the exact value of 1 can be attributed to the inherent numerical behavior of the solution, such as stability characteristics, rounding errors, and the specific way the solution changes near t=1.

Problem 2 - Second-Order Nonlinear ODE: For the differential equation

$$\frac{d^2x}{dt^2} = x - 2x^3$$

a. Derive the system of first order differential equations that you will use to numerically solve it. *(Hand derivation)*

$$\frac{d^2x}{dt^2} = x - 2x^3 \qquad y = \frac{dx}{dt} \qquad \frac{dy}{dt} = y' = \frac{d^2x}{dt^2} \rightsquigarrow \frac{dy}{dt} = x - 2x^3$$

$$\boxed{y = \frac{dx}{dt} \;\&\; \frac{dy}{dt} = x - 2x^3}$$

Problem 2b) Write scripts that numerically solve the ODE for t ∈ [0, 3] with initial conditions x(t = 0) = 1 and x'(t = 0) = 0, and a time step Δt = 0.1, using the explicit Euler Method and 2nd order Runge-Kutta method.

Derivations for part b

for part b Euler & Runge Kutta Methods

Euler

$$x_{n+1} = x_n + \Delta t\left(\frac{dx}{dt}\Big|_n\right) \qquad \frac{dv}{dt} = f(x,v) = x - 2x^3 \qquad g(v) = \frac{dx}{dt} = v$$

(1) $x_{n+1} = x_n + \Delta t\,(v_n)$

$$x_{n+1}' = x_n' + \Delta t\left(\frac{d^2x}{dt^2}\Big|_n\right)$$

$$v_{n+1} = v_n + \Delta t\left(\frac{dv}{dt}\Big|_n\right)$$

(2) $v_{n+1} = v_n + \Delta t\,(x_n - 2x_n^3)$

Runge Kutta

$$k_0 = f(x_n, v_n)$$

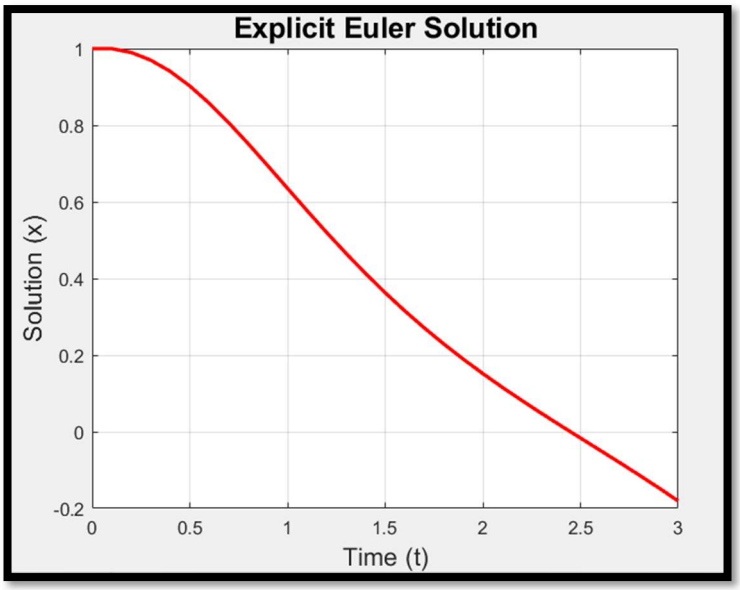
$$k_1 = g\left(x_n + \frac{\Delta t}{2}k_0,\ t_n + \frac{\Delta t}{2}\right)$$

$$k_2 = g\left(x_n + \frac{\Delta t}{2}k_1,\ t_n + \frac{\Delta t}{2}\right)$$

$$k_3 = g(x_n + \Delta t\,k_2,\ t_n + \Delta t)$$

↳ take next step:

$$x_{n+1} = x_n + \frac{\Delta t}{6}(k_0 + 2k_1 + 2k_2 + k_3)$$



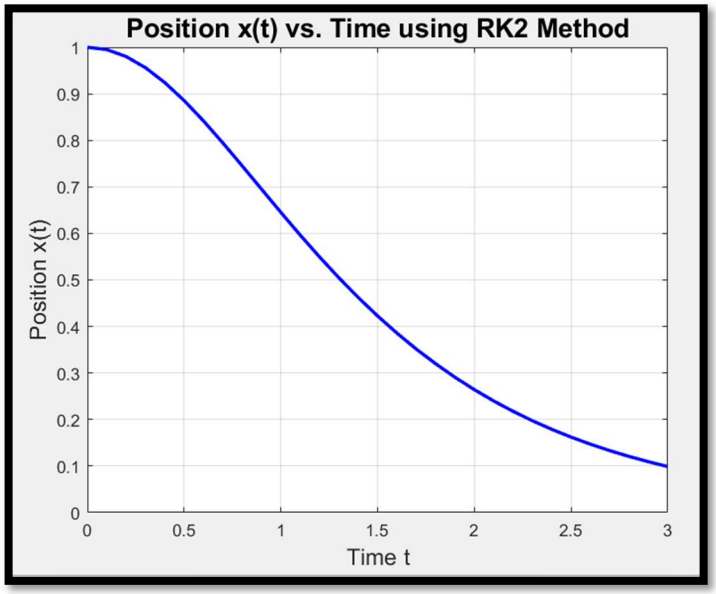


Figure 1: Explicit Euler Solution and X(t) vs Time RK2 Method Graphs. Graphs for Problem 2B

Problem 2B Script (HW4P2B1):

```
clear; close all; clc;

% Problem 2 Part B1 Explicit Euler Method Script

% Initial conditions
x0 = 1; % Initial condition for x at t=0
y0 = 0; % Initial condition for y or dx/dt at t = 0

% Time step
dt = 0.1; % Time step size
t_final = 3; % Final time value
t = 0:dt:t_final; % vector of time pts from 0 to final time with steps of dt

x = zeros(length(t), 1); % Preallocate a column vector for x values with the same
length as t
y = zeros(length(t), 1); % Preallocate a column vector for y values with the same
length as t
x(1) = x0; % Set the first element of the x vector to the initial condition x0
y(1) = y0; % Set the first element of the y vector to the initial condition y0

% Explicit Euler solver loop
for n = 1:(length(t)-1) % Iterate through every time step except for the final one.
    % Update the state using the explicit Euler method
    y(n+1) = y(n) + dt * (x(n) - 2 * x(n)^3);  % Calculate y at the next time step
    x(n+1) = x(n) + dt * y(n);                 % Calculate x at the next time step
based on y
end

% Plot the results
plot(t, x, 'r', 'LineWidth', 2); % Plot x values over time with red line
xlabel('Time (t)', 'FontSize', 14); % x-axis label
ylabel('Solution (x)', 'FontSize', 14); % y-axis label
title('Explicit Euler Solution', 'FontSize', 16); % title of graph
grid on; % have grid displayed
```

Problem 2B Script (HW4P2B2):

```
clear; close all; clc;

%% Set Time parameters
dt = 0.1; % Define the time step size
t  = 0:dt:3; % Define the time vector from 0 to 3 with steps of dt
nT = length(t); % Determine the number of time steps

%% Set Initial conditions
t_n = t(1); % Current time
x_n = 1; % Initial condition for x(t=0)
y_n = 0; % Initial condition for dx/dt(t=0), assuming y = dx/dt

% Initialize arrays to store the trajectory for plotting
x_trajectory = zeros(1, nT); % Array to store x values
x_trajectory(1) = x_n; % Set initial condition
```

```
y_trajectory = zeros(1, nT); % Array to store y values
y_trajectory(1) = y_n; % Set initial condition

%% Perform RK2 Implementation
for n = 1:nT-1
    % Calculate slopes at the current position
    k1x = y_n; % First slope for x, which is y because y = dx/dt
    k1y = x_n - 2*x_n^3; % First slope for y, which is the second derivative of x
(ODE)

    % Calculate the half step position
    x_1h = x_n + (dt/2)*k1x; % Half-step for x
    y_1h = y_n + (dt/2)*k1y; % Half-step for y
    t_h = t_n + dt/2; % Half-step for time

    % Calculate slopes at the half step position
    k2x = y_1h; % Second slope for x, using values at the half step
    k2y = x_1h - 2*x_1h^3; % Second slope for y, using values at the half step

    % Calculate the full step position
    x_np1 = x_n + dt*k2x; % Next position for x using the second slope
    y_np1 = y_n + dt*k2y; % Next position for y using the second slope
    t_np1 = t_n + dt; % Next time step

    % Store the computed values for plotting
    x_trajectory(n+1) = x_np1;
    y_trajectory(n+1) = y_np1;

    % Update position and time for the next iteration
    x_n = x_np1; % Update x to the new value
    y_n = y_np1; % Update y to the new value
    t_n = t_np1; % Update time to the new value
end

% Plot the x trajectory versus time
figure; % Create a new figure window
plot(t, x_trajectory, 'b-', 'LineWidth', 2); % Plot x versus t
xlabel('Time t', 'FontSize', 14); % Label the x-axis
ylabel('Position x(t)', 'FontSize', 14); % Label the y-axis
title('Position x(t) vs. Time using RK2 Method', 'FontSize', 16); % Title for the
plot
grid on; % Turn on the grid
```

Problem 2c) Why would it be hard to implement an implicit method to solve this equation (e.g. Implicit Euler)?

Comments:

Applying an implicit technique like the Implicit Euler to a second-order nonlinear differential equation characterized by the term −2x^3 is inherently complex. Each iteration requires the resolution of a nonlinear algebraic problem, demanding iterative solution methods that are

numerically demanding. Such methods hinge on accurate initial estimates and can be computationally heavy due to the need for repeated iterations and, in some cases, the calculation of derivatives arranged in a Jacobian matrix. The initial conversion of the second-order ODE to a set of first-order equations also amplifies the problem's dimensionality, adding to the method's intricacy. Consequently, the effort to use implicit methods for such nonlinear equations is substantially greater than for explicit methods.

Problem 2d) Derive the Crank-Nicolson equation you would use to solve this problem? Write a code to perform the iterative Crank-Nicolson approach. Take 10 iterations per time step.

Part D Derivation

$$x_{n+1} = x_n + \frac{\Delta t}{2}\left(\left.\frac{dx}{dt}\right|^n + \left.\frac{dx}{dt}\right|^{n+1}\right) \qquad \underbrace{x_{n+1}^{1}}_{\frac{dx}{dt}} = x_n^1 + \frac{\Delta t}{2}\left(\left.\frac{d^2x}{dt^2}\right|^n + \left.\frac{d^2x}{dt^2}\right|^{n+1}\right)$$

$$x_{n+1} = x_n + \frac{\Delta t}{2}(y_n + y_{n+1}) \qquad y_{n+1} = y_n + \frac{\Delta t}{2}\left(x_n - 2x_n^3 + x_{n+1} - 2x_{n+1}^3\right)$$

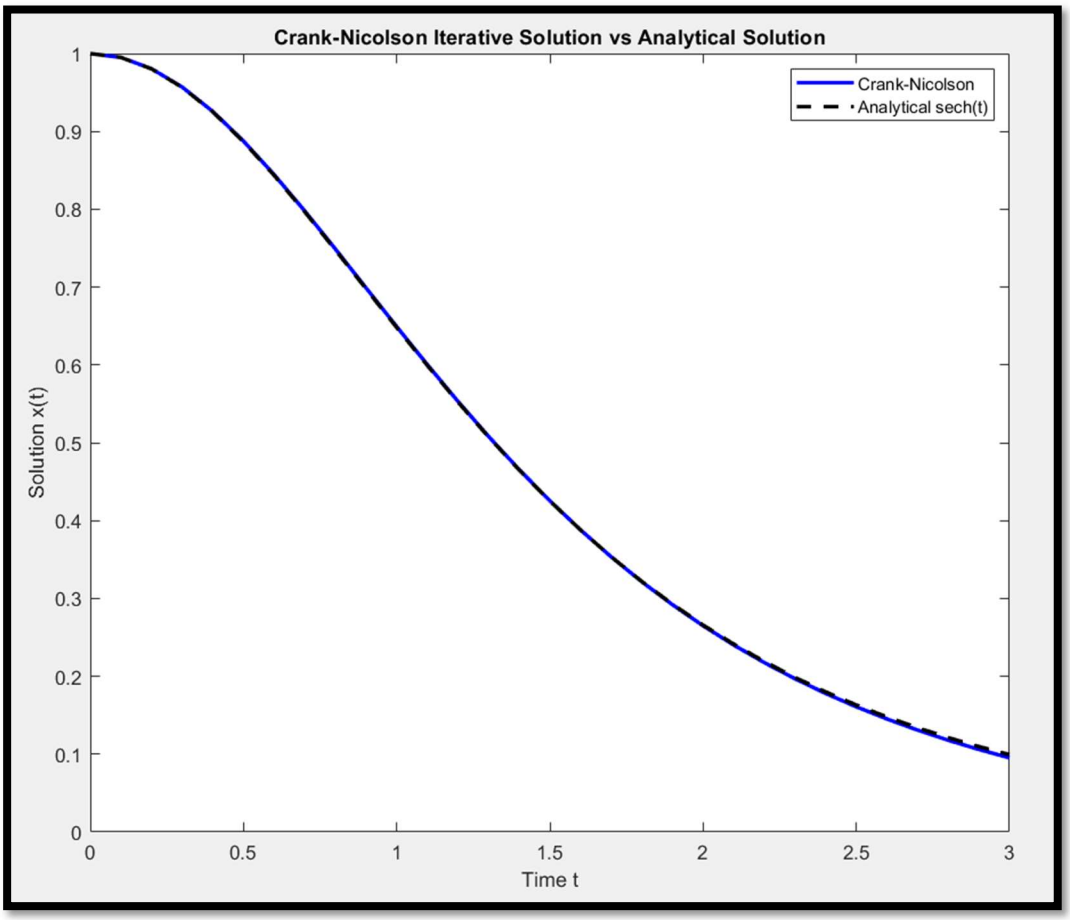


Figure 2: Crank Nicolson Solution vs Analytical Solution Plot for Problem 2D

Problem 2D Script (HW4P2D):

```
% Homework 4 Problem 2 Part D
clear; close all; clc;

% Set up the time parameters for the simulation
dt = 0.1; % Time step size
t  = 0:dt:3; % Time vector from 0 to 3 with increments of dt
nt = length(t); % Total number of time steps

% Number of iterations for the Crank-Nicolson solver within each time step
nIter = 10;

% Initialize solution vectors for x and y with zeros
x_cn = zeros(nt,1); % x at each time step
y_cn = zeros(nt,1); % y (dx/dt) at each time step

% Set initial conditions for x and y
x_cn(1) = 1; % x at t=0
y_cn(1) = 0; % y (dx/dt) at t=0

% Begin the time-stepping loop for Crank-Nicolson method
for n = 1:nt-1
    % Initialize guesses for x and y at the next time step as the current values
    x_np1_guess = x_cn(n); % Initial guess for x at t_(n+1)
    y_np1_guess = y_cn(n); % Initial guess for y at t_(n+1)

    % Perform the specified number of iterations to solve for x and y at the next
time step
    for j = 1:nIter
        % Calculate the function evaluations at the current and guessed next steps
        f_curr = x_cn(n) - 2 * x_cn(n)^3; % f(x_n, y_n)
        f_next = x_np1_guess - 2 * x_np1_guess^3; % f(x_(n+1), y_(n+1))

        % Apply the Crank-Nicolson formula for y
        y_np1_new = y_cn(n) + (dt/2) * (f_curr + f_next);

        % Apply the Crank-Nicolson formula for x
        x_np1_new = x_cn(n) + (dt/2) * (y_cn(n) + y_np1_new);

        % Update the guess for the next iteration
        x_np1_guess = x_np1_new;
        y_np1_guess = y_np1_new;
    end

    % Update the solution vectors with the values at the end of the iterations
    x_cn(n+1) = x_np1_guess;
    y_cn(n+1) = y_np1_guess;
end

% Plotting the Crank-Nicolson and analytical solutions for comparison
plot(t, x_cn, 'b-', 'LineWidth', 2); % Plot numerical solution from Crank-Nicolson
hold on; % Hold the plot for multiple series
```

```
% Compute and plot the analytical solution using the same time vector
x_analytic = sech(t);
plot(t, x_analytic, 'k--', 'LineWidth', 2); % Plot analytical solution

% Add labels, title, and legend to the plot
xlabel('Time t'); % Label for the x-axis
ylabel('Solution x(t)'); % Label for the y-axis
title('Crank-Nicolson Iterative Solution vs Analytical Solution'); % Plot title
legend('Crank-Nicolson', 'Analytical sech(t)'); % Legend for the plot
hold off; % Release the plot hold
```

Problem 2e) Plot the results of your scripts from (b1-2) along with the analytic solution x(t) = sech(t).

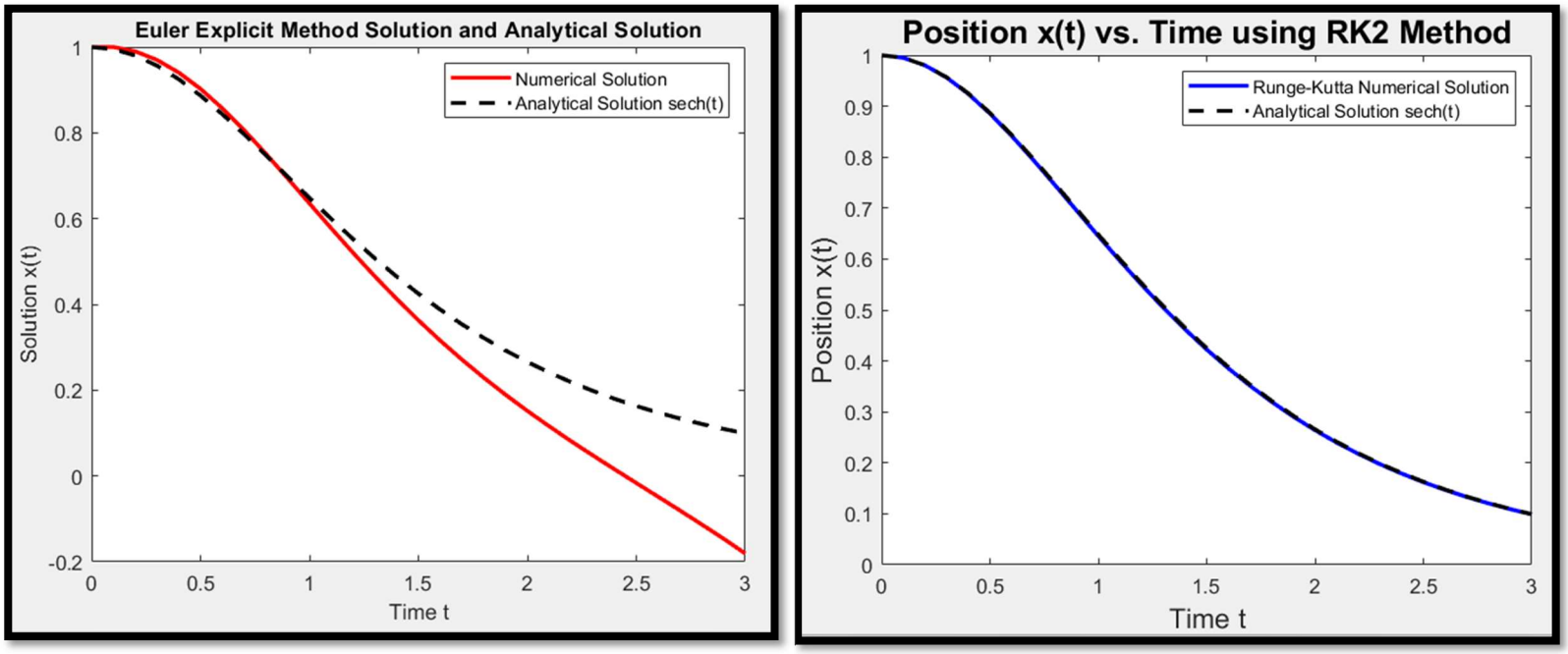


Figure 3: Two plots. Euler Explict Method Solution and Runge-Kutta Method Solution being compared to Analytical Solution sech(t)

Problem 2E Script (HW4P2E1):

```
clear; close all; clc;

% Problem 2 Part E
% From Problem 2 Part B1 UPDATED Explicit Euler Method Script

% Initial conditions
x0 = 1; % Initial condition for x at t=0
y0 = 0; % Initial condition for y or dx/dt at t = 0

% Time step
dt = 0.1; % Time step size
```

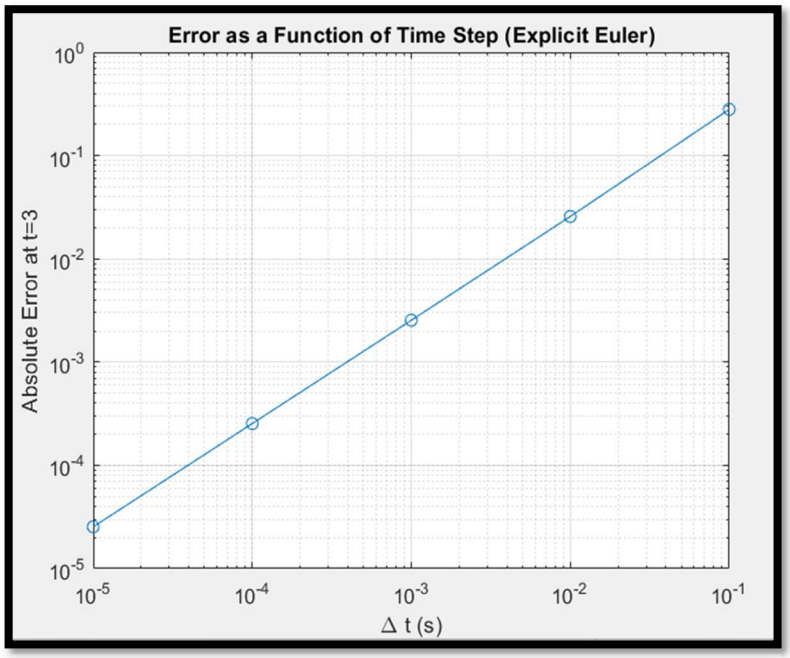
```
t_final = 3; % Final time value
t = 0:dt:t_final; % vector of time pts from 0 to final time with steps of dt

x = zeros(length(t), 1); % Preallocate a column vector for x values with the same
length as t
y = zeros(length(t), 1); % Preallocate a column vector for y values with the same
length as t
x(1) = x0; % Set the first element of the x vector to the initial condition x0
y(1) = y0; % Set the first element of the y vector to the initial condition y0

% Explicit Euler solver loop
for n = 1:(length(t)-1) % Iterate through every time step except for the final one.
    % Update the state using the explicit Euler method
    y(n+1) = y(n) + dt * (x(n) - 2 * x(n)^3);  % Calculate y at the next time step
    x(n+1) = x(n) + dt * y(n);                 % Calculate x at the next time step
based on y
end

% Compute the analytical solution using the same time vector
x_analytic = sech(t);

% Plot the numerical solution
plot(t, x, 'r', 'LineWidth', 2); % Replace 'x' with x_trajectory for RK2 method
hold on;

% Plot the analytical solution
plot(t, x_analytic, 'k--', 'LineWidth', 2);

% Add labels, title, and legend to the plot
xlabel('Time t');
ylabel('Solution x(t)');
title('Euler Explicit Method Solution and Analytical Solution');
legend('Numerical Solution', 'Analytical Solution sech(t)');
hold off; % Release the plot hold
```

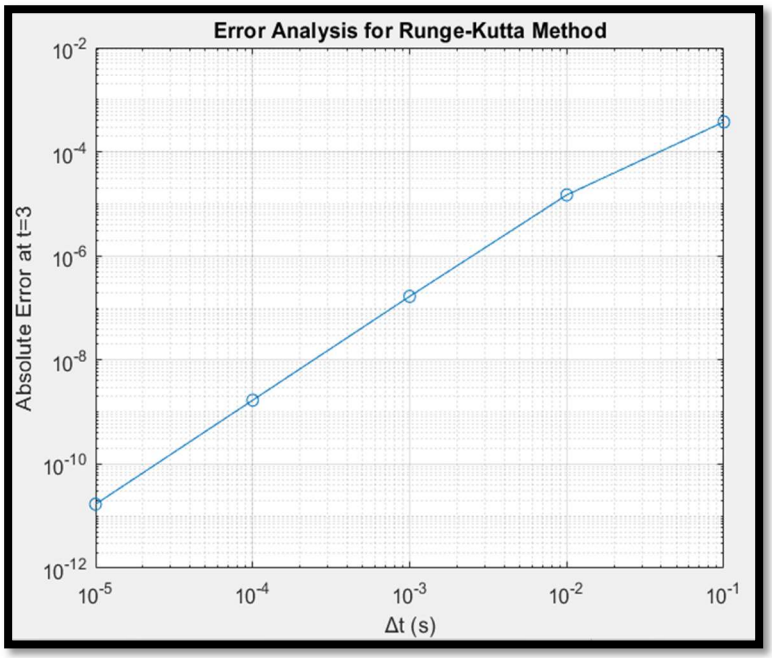
Problem 2E Script (HW4P2E2):

```
clear; close all; clc;

% Problem 2 Part E
% From Problem 2 Part B2 UPDATED 2nd Order Runge-Kutta Method Script

%% Set Time parameters
dt = 0.1; % Define the time step size
t  = 0:dt:3; % Define the time vector from 0 to 3 with steps of dt
nT = length(t); % Determine the number of time steps

%% Set Initial conditions
t_n = t(1); % Current time
x_n = 1; % Initial condition for x(t=0)
y_n = 0; % Initial condition for dx/dt(t=0), assuming y = dx/dt

% Initialize arrays to store the trajectory for plotting
x_trajectory = zeros(1, nT); % Array to store x values
```

```
y_trajectory = zeros(1, nT); % Array to store y values
y_trajectory(1) = y_n; % Set initial condition

%% Perform RK2 Implementation
for n = 1:nT-1
    % Calculate slopes at the current position
    k1x = y_n; % First slope for x, which is y because y = dx/dt
    k1y = x_n - 2*x_n^3; % First slope for y, which is the second derivative of x
(ODE)

    % Calculate the half step position
    x_1h = x_n + (dt/2)*k1x; % Half-step for x
    y_1h = y_n + (dt/2)*k1y; % Half-step for y
    t_h = t_n + dt/2; % Half-step for time

    % Calculate slopes at the half step position
    k2x = y_1h; % Second slope for x, using values at the half step
    k2y = x_1h - 2*x_1h^3; % Second slope for y, using values at the half step

    % Calculate the full step position
    x_np1 = x_n + dt*k2x; % Next position for x using the second slope
    y_np1 = y_n + dt*k2y; % Next position for y using the second slope
    t_np1 = t_n + dt; % Next time step

    % Store the computed values for plotting
    x_trajectory(n+1) = x_np1;
    y_trajectory(n+1) = y_np1;

    % Update position and time for the next iteration
    x_n = x_np1; % Update x to the new value
    y_n = y_np1; % Update y to the new value
    t_n = t_np1; % Update time to the new value
end


% Plot the RK2 numerical solution trajectory versus time
figure; % Create a new figure window
plot(t, x_trajectory, 'b-', 'LineWidth', 2); % Plot RK2 x versus t
hold on; % Hold the figure for the next plot

% Compute the analytical solution and plot it
x_analytic = sech(t); % Analytical solution using the same time vector
plot(t, x_analytic, 'k--', 'LineWidth', 2); % Plot analytical solution

% Add labels, title, and legend to the plot
xlabel('Time t', 'FontSize', 14); % Label the x-axis
ylabel('Position x(t)', 'FontSize', 14); % Label the y-axis
title('Position x(t) vs. Time using RK2 Method', 'FontSize', 16); % Title for the
plot
legend('Runge-Kutta Numerical Solution', 'Analytical Solution sech(t)'); % Add legend
hold off; % Release the plot hold
```

Problem 2f) Rerun your scripts from (b1-2) and (d) using the time steps Δt = 0.1, 0.01, 10−3, 10−4, and 10−5, and calculate the absolute error with respect to the analytic solution at t = 3. Plot the absolute error as a function of time step on a log-log plot. Based on your results, what are the order of global errors for each of your methods? Explain your reasoning.





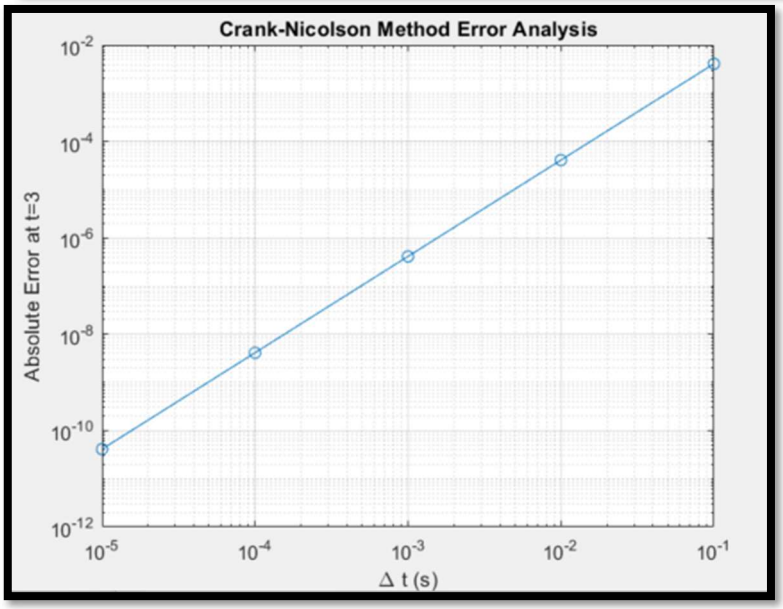
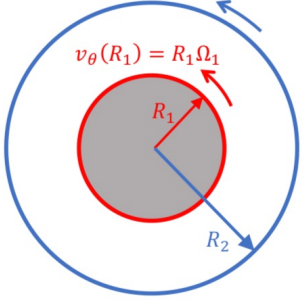


Figure 4: Three plots. Euler Explict Method Solution, Runge-Kutta Method, and Crank-Nicolson Method Log Plots when against analytical solution.

Comments:

In numerical analysis, the global error of a method is assessed by how it scales with the step size. For the Explicit Euler method, the error was found to be 2.53*10^-5. indicative of its first-order nature where error is directly proportional to the step size. The fourth order Runge-Kutta method, designed for greater precision, yielded a significantly lower error of 1.6938×10^-11,

Problem 3 - Taylor-Couette Flow: Shooting Method



The Taylor-Couette flow consists of a system where a viscous fluid is confined in the gap between two rotating concentric cylinders. In a general formulation of this problem, the inner cylinder has radius R_1 and is rotating at an angular velocity Ω_1, while the outer cylinder has radius R_2 and is rotating at Ω_2. The fluid has a dynamic viscosity μ, and both wall boundary conditions are no-slip, making the velocity of the fluid at each boundary match the rotation velocity. The relevant momentum equation is

$$\rho\left(\frac{\partial v_\theta}{\partial t}+v_r\frac{\partial v_\theta}{\partial r}+\frac{v_\theta}{r}\frac{\partial v_\theta}{\partial \theta}+\frac{v_r v_\theta}{r}+v_z\frac{\partial v_\theta}{\partial z}\right)=-\frac{1}{r}\frac{\partial p}{\partial \theta}+\mu\left[\frac{\partial}{\partial r}\left(\frac{1}{r}\frac{\partial}{\partial r}(rv_\theta)\right)+\frac{1}{r^2}\frac{\partial^2 v_\theta}{\partial \theta^2}+\frac{2}{r^2}\frac{\partial v_r}{\partial \theta}+\frac{\partial^2 v_\theta}{\partial z^2}\right]$$

a. The simplest solution for this flow assumes steady-state, axisymmetry, and that the fluid develops no velocity in the radial or axial directions, velocity does not vary in the axial direction, and azimuthal pressure gradients do not develop. Under those assumptions, simplify the Navier-Stokes equations to get an ODE for $v_\theta(r)$ and solve it analytically for $r \in [R_1, R_2]$, accounting for the boundary conditions at $r = R_1$ and $r = R_2$.

Hand Derivation

$$\rho\left(\frac{\partial V_\theta}{\partial t}+V_r\frac{\partial V_\theta}{\partial r}+\frac{V_\theta}{r}\frac{\partial V_\theta}{\partial \theta}+\frac{V_r V_\theta}{r}+V_z\frac{\partial V_\theta}{\partial z}\right)=-\frac{1}{r}\frac{\partial p}{\partial \theta}+\mu\left[\frac{\partial}{\partial r}\left(\frac{1}{r}\frac{\partial}{\partial r}(rV_\theta)\right)+\frac{1}{r^2}\frac{\partial^2 V_\theta}{\partial \theta^2}+\frac{2}{r^2}\frac{\partial V_r}{\partial \theta}+\frac{\partial^2 V_\theta}{\partial z^2}\right]$$

- Steady State
- No velocity in radial or axial directions $\rightarrow V_r = 0,\ V_z = 0$
- axisymmetric $\rightarrow$ flow properties do not change around axis of rotation. no variation in θ so no derivatives for θ
- Velocity no variation in z-direction $\rightarrow$ no derivatives
- no pressure gradient

$$0=\mu\left(\frac{\partial}{\partial r}\left(\frac{1}{r}\frac{\partial (rV_\theta)}{\partial r}\right)\right)$$

- Other information $\rightarrow$ rotational symmetry, steady state, no slip boundary

Boundary Conditions (both no slip) $\rightarrow$ fluid has 0 velocity relative to boundary $\rightarrow$ implies fluid at surface of boundary (surface of cylinders) is @ rest w respect to surface.

1) @ $r = R_1$; fluid velocity at inner cylinder matches rotational velocity of inner cylinder $\rightarrow V_\theta(R_1) = R_1\Omega_1$

2) @ $r = R_2$; fluid velocity at outer cylinder matches rotational velocity of outer cylinder $\rightarrow V_\theta(R_2) = R_2\Omega_2$

$$\mu\left(\frac{d}{dr}\left(\frac{1}{r}\frac{d(rV_\theta)}{dr}\right)\right)=0$$

$$\mu\left(\frac{1}{r}\frac{d}{dr}\left(r\frac{dV_\theta}{dr}+V_\theta\right)\right)=0 \quad \text{divide } \mu$$

$$\frac{d}{dr}\left(\frac{1}{r}\frac{\partial}{\partial r}(rV_\theta)\right)$$

$$\frac{\partial}{\partial r}(rV_\theta) = r\frac{dV_\theta}{dr}+V_\theta$$

$$\frac{d}{dr}\left(\frac{1}{r}\left(r\frac{dV_\theta}{dr}+V_\theta\right)\right)=0 \quad \cdot r$$

$$\frac{d}{dr}\left(r\frac{dV_\theta}{dr}+V_\theta\right)=0$$

$$\int\left(\frac{d}{dr}\left(r\frac{dV_\theta}{dr}+\frac{dV_\theta}{dr}\right)dr=\int 0\,dr\right.$$

$$r\frac{dV_\theta}{dr}+V_\theta = C_1$$

$$\frac{dV_\theta}{dr}+\frac{V_\theta}{r}=\frac{C_1}{r}$$

$$\int\frac{dV_\theta}{dr}dr=\int\frac{C_1}{r}-\frac{V_\theta}{r}\,dr$$

$$V_\theta(r) = C_1\ln|r| - V_\theta\ln|r| + C_2$$

$$V_\theta(r) = C_1 \ln|r| - V_\theta \ln|r| + C_2$$

$$V_\theta(R_1) = R_1\Omega_1 \quad \& \quad V_\theta(R_2) = R_2\Omega_2$$

$$R_1\Omega_1 = C_1 \ln(R_1) - V_\theta \ln(R_1) + C_2$$

$$R_2\Omega_2 = C_1 \ln(R_2) - V_\theta \ln(R_2) + C_2$$

$$R_1\Omega_1 + V_\theta \ln(R_1) = C_1 \ln(R_1) + C_2$$

$$R_2\Omega_2 + V_\theta \ln(R_2) = C_1 \ln(R_2) + C_2$$

$$C_2 = R_2\Omega_2 + V_\theta \ln(R_2) - C_1 \ln(R_2)$$

$$C_2 = R_2\Omega_2 + V_\theta \ln(R_2) - \left(\frac{R_1\Omega_1 - R_2\Omega_2}{\ln\left(\frac{R_1}{R_2}\right)} + V_\theta\right)\ln(R_2)$$

$$R_1\Omega_1 + V_\theta \ln(R_1) = C_1 \ln(R_1) + C_2$$

$$-R_2\Omega_2 - V_\theta \ln(R_2) = -C_1 \ln(R_2) - C_2$$

$$R_1\Omega_1 - R_2\Omega_2 + V_\theta \ln(R_1) - V_\theta \ln(R_2) = C_1\left[\ln\left(\frac{R_1}{R_2}\right)\right]$$

$$R_1\Omega_1 - R_2\Omega_2 + V_\theta\left[\ln\left(\frac{R_1}{R_2}\right)\right] = C_1\left[\ln\left(\frac{R_1}{R_2}\right)\right]$$

$$C_1 = \frac{R_1\Omega_1 - R_2\Omega_2 + V_\theta \ln\left(\frac{R_1}{R_2}\right)}{\ln(R_1/R_2)}$$

$$C_1 = \frac{R_1\Omega_1 - R_2\Omega_2}{\ln\left(\frac{R_1}{R_2}\right)} + V_\theta$$

$$V_\theta(r) = C_1 \ln|r| - V_\theta \ln|r| + C_2$$

$$V_\theta(r) = \left[\frac{R_1\Omega_1 - R_2\Omega_2}{\ln\left(\frac{R_1}{R_2}\right)} + V_\theta\right]\ln|r| - V_\theta \ln|r| + R_2\Omega_2 + V_\theta \ln(R_2) - \left(\frac{R_1\Omega_1 - R_2\Omega_2}{\ln\left(\frac{R_1}{R_2}\right)} + V_\theta\right)\ln(R_2)$$

$$V_\theta(r) = \left[\frac{R_1\Omega_1 - R_2\Omega_2}{\ln\left(\frac{R_1}{R_2}\right)}\right]\ln|r| + R_2\Omega_2 - \left[\frac{R_1\Omega_1 - R_2\Omega_2}{\ln\left(\frac{R_1}{R_2}\right)}\right]\ln|R_2|$$

$$\boxed{V_\theta(r) = \frac{R_1\Omega_1 - R_2\Omega_2}{\ln\left(\frac{R_1}{R_2}\right)}\left[\ln\left|\frac{r}{R_2}\right|\right] + R_2\Omega_2}$$

b. The simplified Navier-Stokes equation from **a** can equivalently be written as

$$\frac{1}{r}\frac{dv_\theta}{dr} - \frac{v_\theta}{r^2} + \frac{d^2v_\theta}{dr^2} = 0.$$

Rewrite this second order ODE as a system of two first order ODEs.

Hand Derivation

$$u(r) = \frac{dV_\theta}{dr}$$

$$\frac{du}{dr} = \frac{d^2V_\theta}{dr^2}$$

$$\frac{1}{r}\frac{dV_\theta}{dr} - \frac{V_\theta}{r^2} + \frac{d^2V_\theta}{dr^2} = 0$$

$$\frac{1}{r}u - \frac{V_\theta}{r^2} + \frac{du}{dr} = 0$$

$$\frac{du}{dr} = \frac{V_\theta}{r^2} - \frac{1}{r}u$$

System of two first order ODEs

$$\frac{dV_\theta}{dr} = u \quad \& \quad \frac{du}{dr} = \frac{V_\theta}{r^2} - \frac{1}{r}u$$

aligning with the expectation of a steeper slope and thus a higher order of accuracy on a log-log scale. Surprisingly, the Crank-Nicolson method, typically second-order, produced an error of 4.1162*10^-11, which is very close to that of the Runge-Kutta method, suggesting an unusually high accuracy for this case that may be attributed to specific attributes of the problem or the numerical implementation. Therefore, the order is explicit euler, runge-kutta, and then crank-Nicolson method.

Problem 2F Script (HW4P2F1):

```
clear; close all; clc;

% From Problem 2 Part B and E UPDATED Explicit Euler Script
% Array of time steps

time_steps = [0.1, 0.01, 1e-3, 1e-4, 1e-5];

% Initialize array to store errors
errors = zeros(size(time_steps));

% Analytical solution at t=3
x_analytic_at_3 = sech(3);

% Loop over each time step
for i = 1:length(time_steps)
    dt = time_steps(i);
    t = 0:dt:3; % Update time vector based on current time step
    nT = length(t); % Update number of time steps

    % Initialize x and y for this dt
    x = zeros(nT, 1);
    y = zeros(nT, 1);
    x(1) = 1; % initial condition for x
    y(1) = 0; % initial condition for y (dx/dt at t=0)

    % Explicit Euler solver loop for the current dt
    for n = 1:(nT-1)
        y(n+1) = y(n) + dt * (x(n) - 2 * x(n)^3);
        x(n+1) = x(n) + dt * y(n);
    end

    % Compute the absolute error at t=3
    errors(i) = abs(x(end) - x_analytic_at_3); %  2.5390e-05

end

% Plot the absolute errors against time steps on a log-log plot
figure;
loglog(time_steps, errors, '-o'); % log log plot for errors
xlabel('\Delta t (s)'); % x-axis label
ylabel('Absolute Error at t=3'); % y-axis label
title('Error as a Function of Time Step (Explicit Euler)'); % title label
grid on;
```

Problem 2F Script (HW4P2F2):

```
clear; close all; clc;

% Define time steps to analyze
time_steps = [0.1, 0.01, 1e-3, 1e-4, 1e-5];
errors = zeros(size(time_steps)); % Initialize error storage

% Loop over each time step
for i = 1:length(time_steps)
    dt = time_steps(i);
    t = 0:dt:3; % Update time vector based on current dt
    nT = length(t); % Update number of time steps

    % Initialize x and y for RK2 method
    x = zeros(1, nT);
    y = zeros(1, nT);
    x(1) = 1; % initial condition for x
    y(1) = 0; % initial condition for y (dx/dt at t=0)

    % Perform RK2 Implementation
    for n = 1:nT-1
        % Calculate slopes at the current position
        k1x = y(n);
        k1y = x(n) - 2*x(n)^3;

        % Calculate the half step position
        x_1h = x(n) + (dt/2)*k1x;
        y_1h = y(n) + (dt/2)*k1y;

        % Calculate slopes at the half step position
        k2x = y_1h;
        k2y = x_1h - 2*x_1h^3;

        % Calculate the full step position
        x(n+1) = x(n) + dt*k2x;
        y(n+1) = y(n) + dt*k2y;
    end

    % After RK2 calculation, compute the absolute error at t=3
    errors(i) = abs(x(end) - sech(3)); % 1.6938e-11

end

% Plotting the errors on a log-log plot
figure;
loglog(time_steps, errors, '-o'); % declaration of log log plot
xlabel('Δt (s)'); % x-axis label
ylabel('Absolute Error at t=3'); % y-axis label
title('Error Analysis for Runge-Kutta Method'); % title label
grid on;
```

Problem 2F Script (HW4P2F3):

```
clear; close all; clc;

% Problem 2 Part F
% From Problem 2 Part D UPDATED Crank-Nicolson Script

% Time steps to analyze
time_steps = [0.1, 0.01, 1e-3, 1e-4, 1e-5];
errors = zeros(size(time_steps)); % Initialize error storage

% Analytical solution at t=3
x_analytic_at_3 = sech(3);

% Number of iterations for the Crank-Nicolson solver within each time step
nIter = 10;

% Loop over each time step
for i = 1:length(time_steps)
    dt = time_steps(i);
    t = 0:dt:3; % Update time vector
    nt = length(t); % Update number of time steps

    % Initialize solution vectors for x and y with zeros
    x_cn = zeros(nt,1); % x at each time step
    y_cn = zeros(nt,1); % y (dx/dt) at each time step

    % Set initial conditions for x and y
    x_cn(1) = 1; % x at t=0
    y_cn(1) = 0; % y (dx/dt) at t=0

    % Begin the time-stepping loop for Crank-Nicolson method
    for n = 1:nt-1
        % Initialize guesses for x and y at the next time step as the current values
        x_np1_guess = x_cn(n); % Initial guess for x at t_(n+1)
        y_np1_guess = y_cn(n); % Initial guess for y at t_(n+1)

        % Perform the specified number of iterations to solve for x and y at the next
time step
        for j = 1:nIter
            % Calculate the function evaluations at the current and guessed next
steps
            f_curr = x_cn(n) - 2 * x_cn(n)^3; % f(x_n, y_n)
            f_next = x_np1_guess - 2 * x_np1_guess^3; % f(x_(n+1), y_(n+1))

            % Apply the Crank-Nicolson formula for y
            y_np1_new = y_cn(n) + (dt/2) * (f_curr + f_next);

            % Apply the Crank-Nicolson formula for x
            x_np1_new = x_cn(n) + (dt/2) * (y_cn(n) + y_np1_new);

            % Update the guess for the next iteration
            x_np1_guess = x_np1_new;
            y_np1_guess = y_np1_new;
        end

        % Update the solution vectors with the values at the end of the iterations
```

```
        x_cn(n+1) = x_np1_guess;
        y_cn(n+1) = y_np1_guess;
    end

    % Calculate absolute error at t=3
    errors(i) = abs(x_cn(end) - x_analytic_at_3); % 4.1162e-11

end

% Plot errors on a log-log plot
figure;
loglog(time_steps, errors, '-o');
xlabel('\Delta t (s)');
ylabel('Absolute Error at t=3');
title('Crank-Nicolson Method Error Analysis');
grid on;
```

This problem is intrinsically a boundary value problem, because the function v_θ is known both at R_1 and R_2. The numerical time-marching schemes we have studied so far are designed to solve initial value problems, for which the so-called initial-conditions are both given at a single point (either $r = R_1$ or $r = R_2$). An approach to solve this problem using our typical marching schemes is referred to as the shooting method, and implementing this method is our final goal in this problem.

Let's start, however, by simplifying the problem to an initial value problem for which we know information at $r = R_1$ only, and let's *neglect the condition at R_2 for now.* For what follows, use as initial conditions:

$$v_\theta(r = R_1) = R_1\Omega_1 = v_1, \quad \frac{dv_\theta}{dr}(r = R_1) = \tau_1/\mu = v_1',$$

where τ_1 corresponds to the viscous stress at the inner wall, and v_1 and v_1' are auxiliary constants that we introduce to simplify our codes.

Problem 3c) Create a function to solve the initial value problem from b using the Runge-Kutta 2-step (tc rk2.m) method. The function, e.g. vtheta = tc rk2(r,v1,v1p), should take as inputs the array r setting the discretized spatial interval and Δr, the initial conditions v1 and v'1, and output the approximated velocity profile vθ(r).

Problem 3C Function Script (tc_rk2):

```
% Homework 4 Problem 3 Part C
% Define a function to solve the initial value problem using the Runge-Kutta 2-step
method
function v_theta = tc_rk2(r, v1, v1p)
    % Input:
    % r - array setting the discretized spatial interval and step size Delta r
    % v1 - initial condition for v0 at r = R1
    % v1p - initial condition for the derivative of v0 at r = R1, denoted as u

    % Output:
    % v_theta - approximated velocity profile vθ(r)

    % Calculate the step size by subtracting the second element of r from the first
    dr = r(2) - r(1);

    % Initialize the array for v_theta, which will hold the solution v0 at each step
    v_theta = zeros(size(r));
    % Initialize the array for u, which is the derivative of v0 with respect to r
    u = zeros(size(r));

    % Set the first element of v_theta to the initial condition v1
    v_theta(1) = v1;
    % Set the first element of u to the initial condition v1p
    u(1) = v1p;

    % Loop over the array r, stopping one element before the end
    for i = 1:length(r)-1

        k1v0 = u(i);          % Compute the slope k1 for v0 at the current step
```

```
        k1u = v_theta(i)/(r(i)^2) - u(i)/r(i);  % Compute the slope k1 for u at the
current step based on the ODE
        v0_half = v_theta(i) + 0.5 * dr * k1v0; % Estimate v0 at the halfway point of
the current step
        u_half = u(i) + 0.5 * dr * k1u;   % Estimate u at the halfway point of the
current step
        r_half = r(i) + 0.5 * dr;         % Calculate the radius at the halfway point
of the current step

        % Compute the slope k2 for v0 at the halfway point
        k2v0 = u_half;
        % Compute the slope k2 for u at the halfway point based on the ODE
        k2u = v0_half/(r_half^2) - u_half/r_half;

        % Use the slope k2 to estimate the next value of v0
        v_theta(i+1) = v_theta(i) + dr * k2v0;
        % Use the slope k2 to estimate the next value of u
        u(i+1) = u(i) + dr * k2u;
    end
end
```

Problem 3D) Write a script that calls your function and plots the velocity profile obtained as a function of r. Use R1 = 2, R2 = 6 and Δr = 0.01 to define r = R1 : Δr : R2, and use initial conditions v1 = 2 (corresponding to Ω1 = 1), and v'_1 = 0 (arbitrary choice, for now). What is the value of Ω2 for the obtained solution?

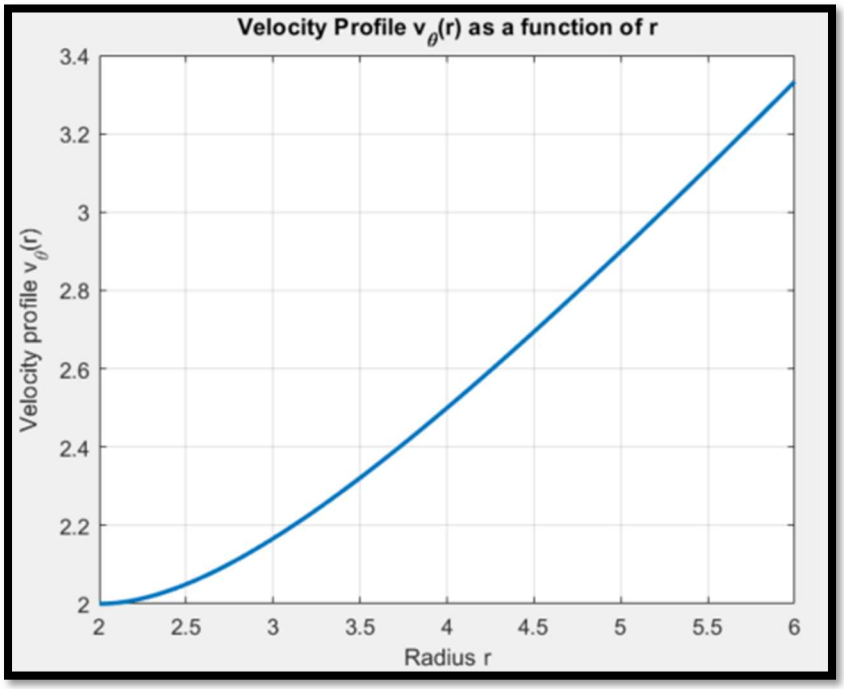


Figure 5: Velocity Profile V_theta (r) as a function of radius going in concave up trend.

Comment: The value of Ω_2 value is 3.333 based on the command terminal output.

The provided graph shows a velocity profile $V_\theta(r)$ that increases with the radius r from R1 to R2, beginning at a velocity of 2 with a horizontal slope at R1. This upward concave shape indicates that not only does the fluid's velocity increase as it moves from the inner to the outer boundary, but the rate of this increase also accelerates with radius. This behavior is a result of the specific conditions and equations governing the flow, likely influenced by the radial distribution of viscous stresses within the fluid. If Ω2 is taken to be proportional to the velocity at R2, the graph suggests that its value would be approximately 3.4, based on the end value of the velocity profile at the outer boundary.

Problem 3D Function Script (HW4P3D):

```
% Homework 4 Problem 3 Part C
% Define a function to solve the initial value problem using the Runge-Kutta 2-step
method
function v_theta = tc_rk2(r, v1, v1p)
    % r - array setting for interval and step size Delta r
    % v1 - initial condition for v0 at r = R1
    % v1p - initial condition for the derivative of v0 at r = R1, denoted as u
    % v_theta - approximated velocity profile vθ(r) -> for output

    % Calculate the step size by subtracting the second element of r from the first
    dr = r(2) - r(1);

    % Initialize the array for v_theta, which will hold the solution v0 at each step
    v_theta = zeros(size(r));
    % Initialize the array for u, which is the derivative of v0 with respect to r
    u = zeros(size(r));

    % Set the first element of v_theta to the initial condition v1
    v_theta(1) = v1;
    % Set the first element of u to the initial condition v1p
    u(1) = v1p;

    % Loop over the array r, stopping one element before the end
    for i = 1:length(r)-1

        k1v0 = u(i);         % Compute the slope k1 for v0 at the current step
        k1u = v_theta(i)/(r(i)^2) - u(i)/r(i);  % Compute the slope k1 for u at the
current step based on the ODE
        v0_half = v_theta(i) + 0.5 * dr * k1v0; % Estimate v0 at the halfway point of
the current step
        u_half = u(i) + 0.5 * dr * k1u;   % Estimate u at the halfway point of the
current step
        r_half = r(i) + 0.5 * dr;         % Calculate the radius at the halfway point
of the current step

        % Compute the slope k2 for v0 at the halfway point
        k2v0 = u_half;
        % Compute the slope k2 for u at the halfway point based on the ODE
```

```
        k2u = v0_half/(r_half^2) - u_half/r_half;

        % Use the slope k2 to estimate the next value of v0
        v_theta(i+1) = v_theta(i) + dr * k2v0;
        % Use the slope k2 to estimate the next value of u
        u(i+1) = u(i) + dr * k2u;
    end
end
```

In the following items, we will apply the predictor-corrector method to try and match the correct solution to the boundary value problem. This method consists of varying the (unknown) value of v′ 1 in order to find the specific value leading to a solution such that vθ(r = R2) = R2Ω2.

Problem 3E) For Ω1 = 1 and Ω2 = 1/3, write a script that varies the parameter v′1 and for each of value calls your RK2 function to compute the solution and read the value corresponding to vθ(r = R2). Use a binary-search (also known as bisection) method to vary v′1 to get Ω2 = vθ(R2)/R2 = 1/3 ± 0.0001. Report the lower and upper bounds for v′1 that you used on your binary-search, as well as the value of v′1 leading to the solution that satisfies both boundary conditions. Plot your numerical result and compare it to the analytic solution (from a).

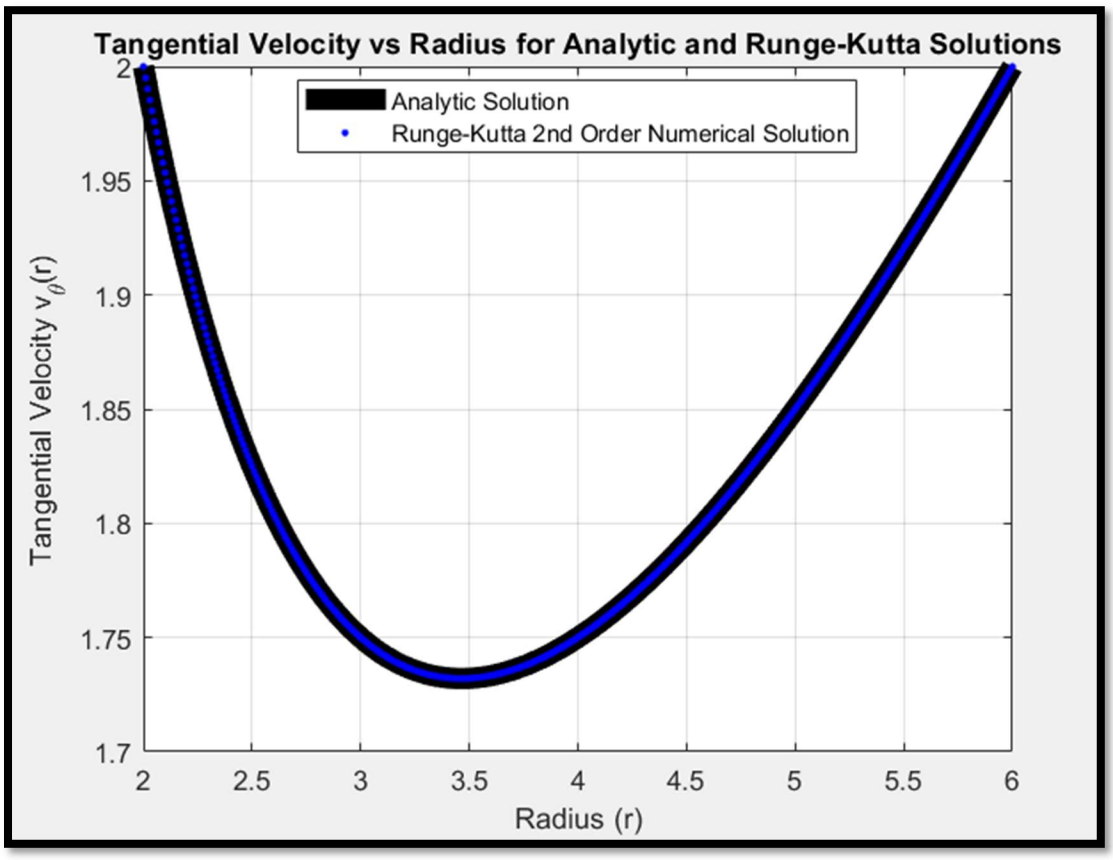


Figure 6: Numerical Solution and Analytical Solution Velocity vs r graph

After 100 iterations: Leading solution lower bound of v1': -0.499998, Leading solution upper bound of v1': -0.499998. The value of v1’ leading to the solution is approximately: -0.499998 Solution converges.

Problem 3E Function Script (HW4P3E):

```
% Homework 4 Problem 3 Part E
clear; close all; clc;

% Define all parameters
R1 = 2; % Initial radius
R2 = 6; % Final radius
Omega1 = 1; % Initial angular velocity
Omega2 = 1/3; % Target angular velocity at R2
dr = 0.01; % Step size for the radius
r = R1:dr:R2;
OmegaGuess = -1; % Setting an initial value to be replaced once we iterate

% Boundary initial conditions
v1p_lower = -1;
v1p_upper = 1;

% Run the loop for exactly 100 iterations
for iter = 1:100
    % Velocity profile and bisecting function
    v1 = 2; % Guess for v1 initial condition
    v1p = (v1p_upper + v1p_lower)/2; % Compute midpoint for v1p search
    vtheta = tc_rk2(r, v1, v1p); % Compute the numerical solution from function.
    OmegaGuess = vtheta(end)/R2; % Calculate Omega at the final radius of R2

    % Adjusting bounds based on the comparison of OmegaGuess and Omega2
    if OmegaGuess < Omega2
        v1p_lower = v1p; % Update lower bound for v1p
    else
        v1p_upper = v1p; % Update upper bound for v1p
    end
end

% The value of v1’ leading to the solution
v1p_solution = (v1p_lower + v1p_upper) / 2;

% After 100 iterations, display the solution bounds of v1'
fprintf('After 100 iterations:\n');
fprintf('Leading solution lower bound of v1'': %.6f\n', v1p_lower);
fprintf('Leading solution upper bound of v1'': %.6f\n', v1p_upper);
fprintf('The value of v1’ leading to the solution is approximately: %.6f\n',
v1p_solution);

% Plotting Graph
vtheta_analytic = (1/(R2^2-R1^2))*(R2^2*Omega2*(r-(R1^2./r)) - R1^2*Omega1*(r-
(R2^2./r)));
plot(r, vtheta_analytic, 'k', 'LineWidth', 2);
hold on;
plot(r, vtheta, 'b.', 'MarkerSize', 10); % Plot numerical solution with blue dots
```

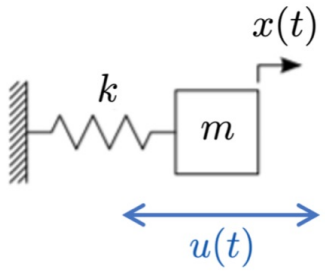
```
title('Tangential Velocity vs Radius for Analytic and Runge-Kutta Solutions');
xlabel('Radius (r)');
ylabel('Tangential Velocity v_{\theta}(r)');
legend({'Analytic Solution', 'Runge-Kutta 2nd Order Numerical Solution'}, 'Location',
'best');
grid on; % Added grid for better visualization
hold off;
```

Problem 4 - Block immersed in oscillatory flow: A block of mass m, connected to a spring of constant k, is immersed in a flow for which the background velocity oscillates in time according to $u(t) = \sin(2\pi t)$. The equation of motion for the block is

analytic

$$m\ddot{x} + kx = F_d,$$

where $x(t)$ is its position in time, dots represent time derivatives, and F_d is the drag force exerted by the fluid on the block. The initial conditions for the system are $x(t=0) = 0$ and $\dot{x}(t=0) = 0$.



a. Transform this second order ODE into a system of first order ODEs. Write down the equations to time-advance this linear system using the Crank-Nicolson method. For this part, we want to use the method without the iterative approach, so isolate all $n+1$ terms.

Hand derivation

$x(t=0)=0$

$\dot{x}(t=0)=0$

$m\ddot{x} + kx = F_d$

$F_d = C_d \underbrace{u(t)}_{\sin 2\pi t}$ (drag)

$\dot{x}(t) = v(t)$

$\ddot{x}(t) = \dot{v}(t)$

$\frac{m\ddot{x}}{m} + \frac{kx}{m} = \frac{F_d}{m}$

$\ddot{x} + \frac{k}{m}x = \frac{F_d}{m}$

$\ddot{x} = \frac{F_d}{m} - \frac{kx}{m}$

$\dot{v}(t) = \frac{F_d}{m} - \frac{kx}{m}$

$\dot{v}(t) = \frac{C_d \sin 2\pi t}{m} - \frac{k}{m}x(t)$

Crank - Nicolson Method

$x_{n+1} = x_n + \frac{\Delta t}{2}\left(\frac{dx}{dt}\Big|^n + \frac{dx}{dt}\Big|^{n+1}\right)$

$\boxed{x_{n+1} = x_n + \frac{\Delta t}{2}\left(v_n + v_{n+1}\right)}$

$v_{n+1} = \dot{x}_n + \frac{\Delta t}{2}\left(\frac{d^2x}{dt^2}\Big|^n + \frac{d^2x}{dt^2}\Big|^{n+1}\right)$

$v_{n+1} = v_n + \frac{\Delta t}{2}\left(\dot{v}_n + \dot{v}_{n+1}\right)$

$v_{n+1} = v_n + \frac{\Delta t}{2}\left(\frac{C_d(\sin 2\pi t_n)}{m} - \frac{k}{m}x_n + \frac{C_d(\sin 2\pi t_{n+1})}{m} - \frac{k}{m}x_{n+1}\right)$

$v_{n+1} = v_n + \frac{\Delta t}{2}\left[\frac{C_d(\sin 2\pi t_n)}{m} + \frac{C_d(\sin 2\pi t_{n+1})}{m} - \frac{k}{m}\left(x_n + x_{n+1}\right)\right]$

$v_{n+1} = v_n + \frac{C_d \Delta t}{2m}\left[\sin(2\pi t_n) + \sin(2\pi t_{n+1})\right] - \frac{k\Delta t}{2m}\left(x_n + x_{n+1}\right)$

$v_{n+1} = v_n + \frac{C_d \Delta t}{2m}\left[\sin(2\pi t_n) + \sin(2\pi t_{n+1})\right] - \frac{k\Delta t}{2m}\left(\frac{2x_n}{x_{n+1}+x_n} + \frac{v_n \Delta t}{2} + \frac{v_{n+1}\Delta t}{2}\right)$

$v_{n+1} = v_n + \frac{C_d \Delta t}{2m}\left[\sin(2\pi t_n) + \sin(2\pi t_{n+1})\right] - \frac{k x_n \Delta t}{m} - \frac{v_n k \Delta t^2}{4m} - \frac{k v_{n+1}\Delta t^2}{4m}$

$v_{n+1} + \frac{k v_{n+1}\Delta t^2}{4m} = v_n + \frac{C_d \Delta t}{2m}\left[\sin(2\pi t_n) + \sin(2\pi t_{n+1})\right] - x_n \frac{k\Delta t}{m} - \frac{v_n k \Delta t^2}{4m}$

$v_{n+1}\left(1 + \frac{k\Delta t^2}{4m}\right) = v_n\left(1 - \frac{k\Delta t^2}{4m}\right) + \frac{\Delta t}{2m} C_d\left[\sin(2\pi t_n) + \sin(2\pi t_{n+1})\right] - \frac{\Delta t k}{m} x_n$

$$\boxed{v_{n+1} = \frac{v_n\left(1 - \frac{k\Delta t^2}{4m}\right) + \frac{\Delta t}{2m} C_d\left[\sin 2\pi t_n + \sin(2\pi t_{n+1})\right] - \frac{\Delta t k}{m} x_n}{1 + \frac{k\Delta t^2}{4m}}}$$

In reality, the drag force does not depend exclusively on the background flow, but rather on the relative velocity between the flow and the block, $u(t) - \dot{x}$. For the second part of the problem, we will account for this relative velocity, by setting $F_d(t, \dot{x}) = C_d\,(u(t) - \dot{x})\,|u(t) - \dot{x}|$. The main difference from a mathematical standpoint is that the ODE is now nonlinear in $\dot{x}$, and the Crank-Nicolson implementation requires an iterative approach.

b. Derive the modified system of ODEs and the equations to implement the iterative Crank-Nicolson method for the new system.

Hand derivation

$u(t) = \sin(2\pi t)$

$$F_d(t, \dot{x}) = C_d\,(u(t) - \dot{x})\,|u(t) - \dot{x}| \qquad m\ddot{x} + kx = F_d$$

$$\ddot{x} = \frac{F_d}{m} - \frac{kx}{m}$$

recall $\dot{x}(t) = v(t)$, $\ddot{x}(t) = \dot{v}(t)$

$$\dot{v}(t) = \frac{1}{m}(F_d - kx)$$

$$\dot{v}(t) = \frac{1}{m}\left[C_d(\sin(2\pi t) - v)\,|\sin(2\pi t) - v\,| - kx\right]$$

$$x_{n+1} = x_n + \frac{\Delta t}{2}(v_n + v_{n+1})$$

$$v_{n+1} = v_n + \frac{\Delta t}{2}(\dot{v}_n + \dot{v}_{n+1})$$

$$v_{n+1} = v_n + \frac{\Delta t}{2m}\left[C_d(\sin(2\pi t_n) - v_n)\,|\sin(2\pi t_n) - v_n| - kx_n\right] + \frac{\Delta t}{2m}\left[C_d(\sin(2\pi t_{n+1}) - v_{n+1})\,|\sin(2\pi t_{n+1}) - v_{n+1}| - kx_{n+1}\right]$$

Iterative Method

→ Guess for v_{n+1} ; $v_n = v_{n+1}$

→ Calculate x_{n+1} based on the current guess of $v_{n+1} = v_n$.

→ Update guess for v_{n+1} and repeat.

For absolute values, take a look at:

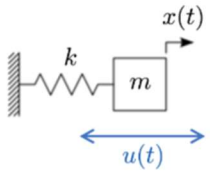
$$|u(t) - v(t)| = u(t) - v(t) \quad \text{if } u(t) \geq v(t)$$

$$|u(t) - v(t)| = -(u(t) - v(t)) = v(t) - u(t) \quad \text{if } u(t) < v(t)$$

Problem 5 - Block immersed in oscillatory flow: A block of mass m, connected to a spring of constant k, is immersed in a flow for which the background velocity oscillates in time according to $u(t) = \sin(2\pi t)$. The equation of motion for the block is

$$m\ddot{x} + kx = F_d,$$

where $x(t)$ is its position in time, dots represent time derivatives, and F_d is the drag force exerted by the fluid on the block. The initial conditions for the system are $x(t = 0) = 0$ and $\dot{x}(t = 0) = 0$.



For the first part of the problem, we simplify the drag force contribution to depend only on the background flow $u(t)$ and not on the block's motion, with $F_d(t) = C_d\, u(t)\, |u(t)|$, where C_d is the drag coefficient.

Problem 5a) Write a function block_cn_linear(t,m,k,Cd) that implements the Crank-Nicolson method you derived on the in class exam. The function should take as inputs a uniformly discretized time vector t and the parameters m, k, and Cd. The outputs of the function should be the discretized approximations for the block's position x(t) and velocity ˙x(t), as well as the discretized background flow u(t).

Problem 5A Function Script (block_cn_linear)

```
% Problem 5 Part A

% Function block_cn_linear with inputs for time vector, mass, stiffness, and damping
coefficient

function [x, v, u] = block_cn_linear(t, m, k, Cd)

    % Initialize arrays for positions (x), velocities (v), and background flow (u)
    N = length(t); % Determine the # of time steps based on the length of the time
vector
    x = zeros(1, N); % Initialize position array with zeros
    v = zeros(1, N); % Initialize velocity array with zeros
    u = zeros(1, N); % Initialize background flow array with zeros

    % Calculate the time step size based on the time vector
    dt = t(2) - t(1); % Used from Problem set 3

    % Set initial conditions for position and velocity aka x'
    x(1) = 0; % Initial position
    v(1) = 0; % Initial velocity

    % Calculate the background flow u(t) for each time step
    for i = 1:N
        u(i) = sin(2 * pi * t(i)); % Using u(t) = sin(2*pi*t)
    end

    % Loop through each time step to calculate position and velocity
```

```
    for n = 1:N-1
        u_n = sin(2 * pi * t(n)) + sin(2 * pi * t(n+1)); % Calculate the combined
effect for the current
                                                          % and next time steps for
the driving force

        % Crank-Nicolson method to update position
        x(n+1) = x(n) + (dt / 2) * (v(n) + v(n+1)); % Derivation of x_n+1 from
Question 4 Part a

        % Crank-Nicolson method to update velocity
        v(n+1) = (v(n)*(1 - (k*dt^2)/(4*m)) + (dt/(2*m))*Cd*u_n - (dt*k/m)*x(n)) / (1
+ (k*dt^2)/(4*m));
    end
end

% Define the inputs as follows in the MATLAB command window in order to
% test function
% t = linspace(0, 10, 100); % Creates a vector of 100 time points from 0 to 10
% m = 1; % Mass of the block
% k = 20; % Spring constant
% Cd = 0.1; % Damping coefficient

% On command window, I have to type the function with these inputs
% [x, v, u] = block_cn_linear(t, m, k, Cd);
```

Problem 5b) Write a script p2b that calls your function with m = 1, k = 1, and Cd = 2.5, for t = 0 : 0.01 : 10. Create two figures. On one, plot the background flow velocity u(t) and the block velocity $\dot{x}$(t). On the other, plot the position of the block x(t). Comment on the physical meaning of what you observe and the relationship between the background flow and the block's position and velocity.

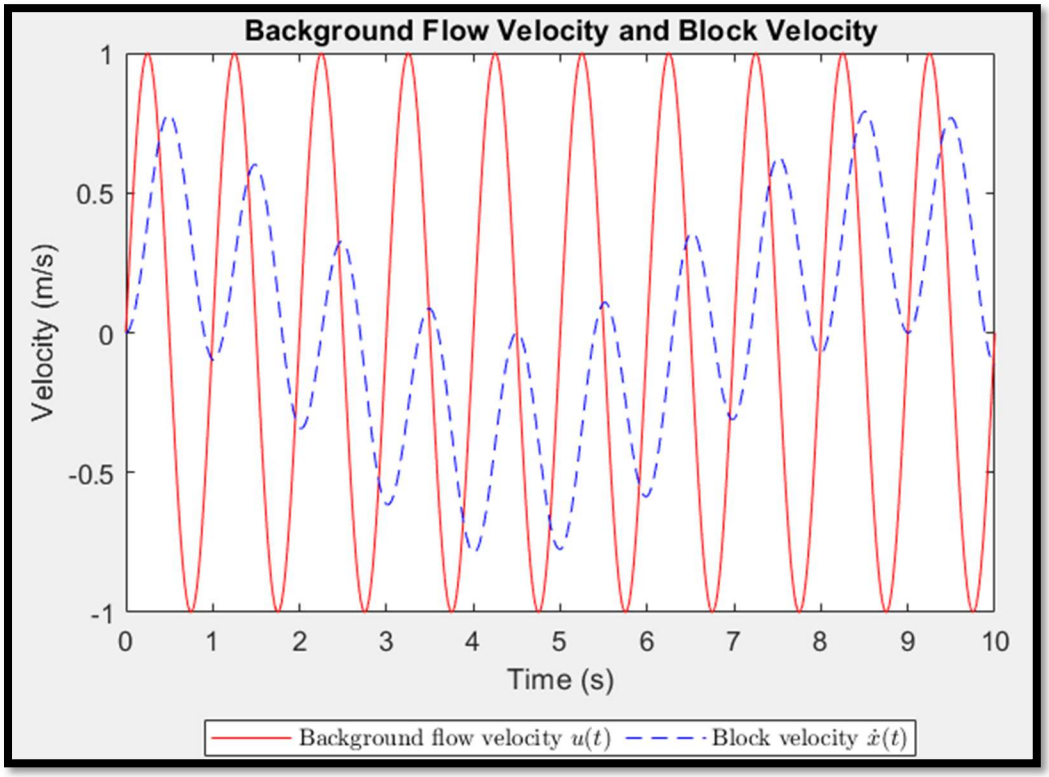


Figure 7: Flow Velocity and Block Velocity Comparisons as a function of time

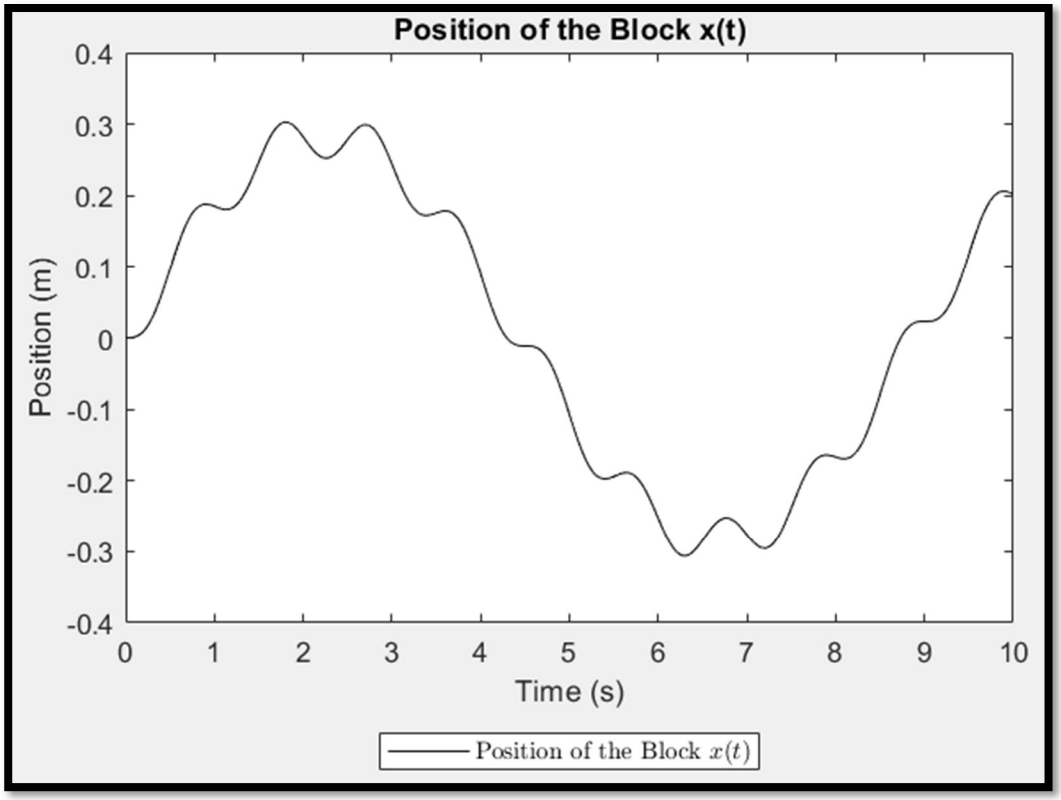


Figure 8: Position of Block as a function of time

Comments:

In the plots, we're looking at how a moving fluid (like water or air) affects a block's motion. The first plot shows how the fluid's speed changes in a regular up-and-down pattern, like ocean waves. At the same time, it shows how fast the block moves. When the block moves at the same speed as the fluid, it's easier for the block, like surfing a wave. But when they're moving in opposite directions, it's harder for the block to go anywhere.

The second plot is about where the block is at different times. It swings back and forth because of the fluid, a spring that pulls it back, and something slowing it down. The size of these swings and when they happen depend on how fast the fluid waves are and how much the block is "springy."

Based on the plots, I can infer that there is a mix of forces. The fluid pushes the block, the spring pulls it back, and something slows it down. Imagine trying to push a swing – you can make it go higher if you push it at the right time, but if you push when it's coming back at you, it's harder.

By looking at these plots, we can see when the block moves best with the fluid, and when it struggles. It helps us understand how the block responds to the fluid, the spring, and the slowdown, and how all these things work together.

Problem 5B Code Script (p2b):

```
% Problem 5 Part B

% Define the parameters
m = 1; % Setting a value for mass
k = 1; % Setting a value for spring coefficient
Cd = 2.5; % Setting value for drag
t = 0:0.01:10; % Creating a time vector 't' from 0 to 10 with a step of 0.01
[x, v, u] = block_cn_linear(t, m, k, Cd); % Call the function


% Create the first figure for background flow velocity u(t) and block velocity v(t)
figure;
plot(t, u, 'r-', t, v, 'b--'); % plotting parameters
xlabel('Time (s)'); % x-axis label
ylabel('Velocity (m/s)'); % y-axis label
% Making it latex to have v dot and u in greek letter. Moved legend to
% bottom of graph but in one line
legend('Background flow velocity $u(t)$', 'Block velocity $\dot{x}(t)$',
'Interpreter', 'latex', 'Location', 'southoutside', 'Orientation', 'horizontal');
title('Background Flow Velocity and Block Velocity'); % title label

% Create the second figure for the position of the block x(t)
figure;
plot(t, x, 'k-'); % plotting parameters
xlabel('Time (s)'); % x-axis label
ylabel('Position (m)');  % y-axis label
```

```
legend('Position of the Block $x(t)$', 'Interpreter', 'latex', 'Location',
'southoutside', 'Orientation', 'horizontal');
title('Position of the Block x(t)'); % title label
```

Prompt from Problem Set 5c and 5d: In reality, the drag force does not depend exclusively on the background flow, but rather on the relative velocity between the flow and the block, u(t) − ˙x. For the second part of the problem, we will account for this relative velocity, by setting Fd(t, ˙x) = Cd (u(t) − ˙x) |u(t) − ˙x|. The main difference from a mathematical standpoint is that the ODE is now nonlinear in ˙x, and the Crank-Nicolson implementation requires an iterative approach.

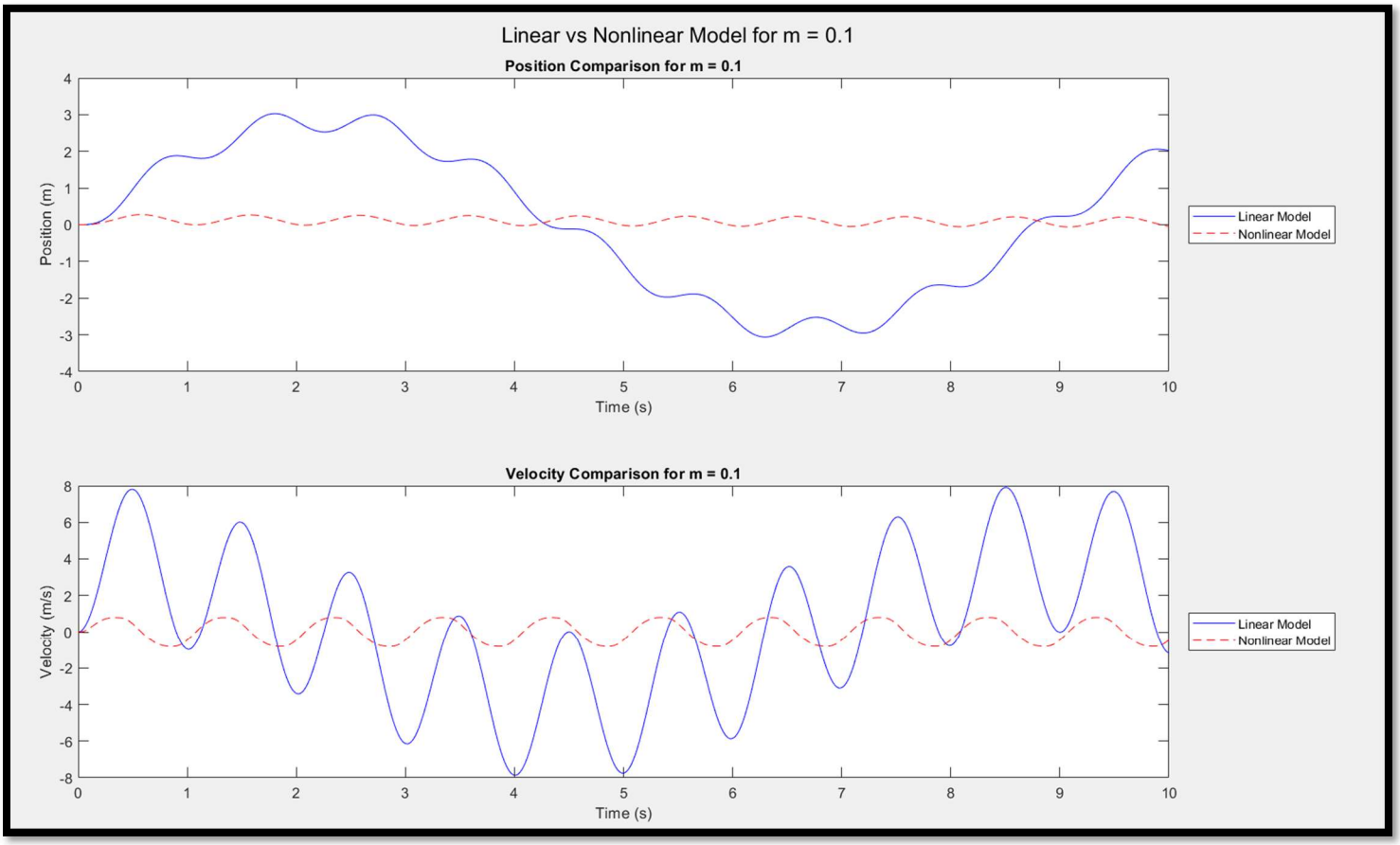
Problem 5c) Write a function block cn nonlinear(t,m,k,Cd) that implements the iterative Crank-Nicolson method for the system that accounts for the relative velocity. Perform 10 iterations within each time step of the method. The inputs and outputs should be the same as in a.

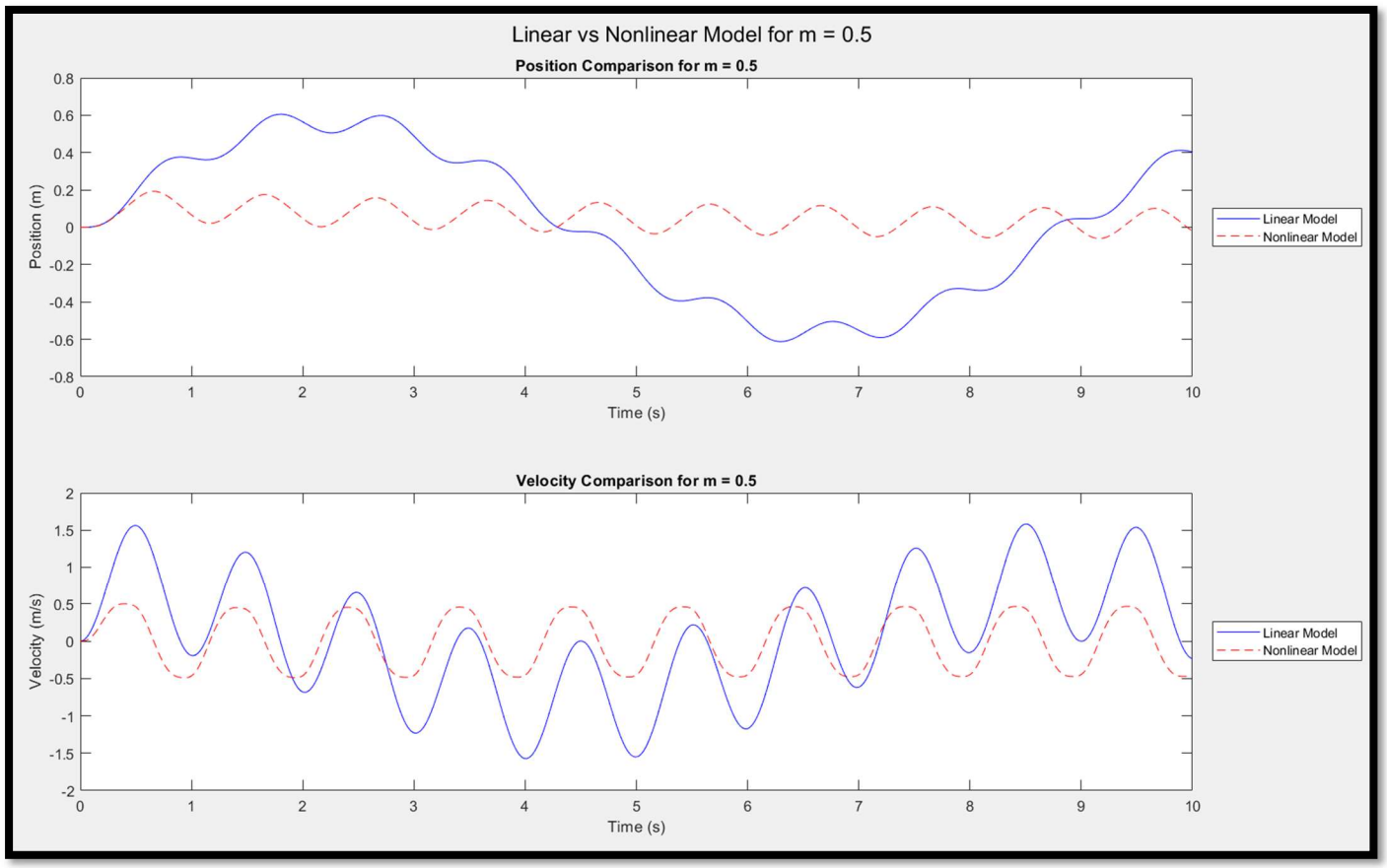
Problem 5c Code Script (block_cn_nonlinear):

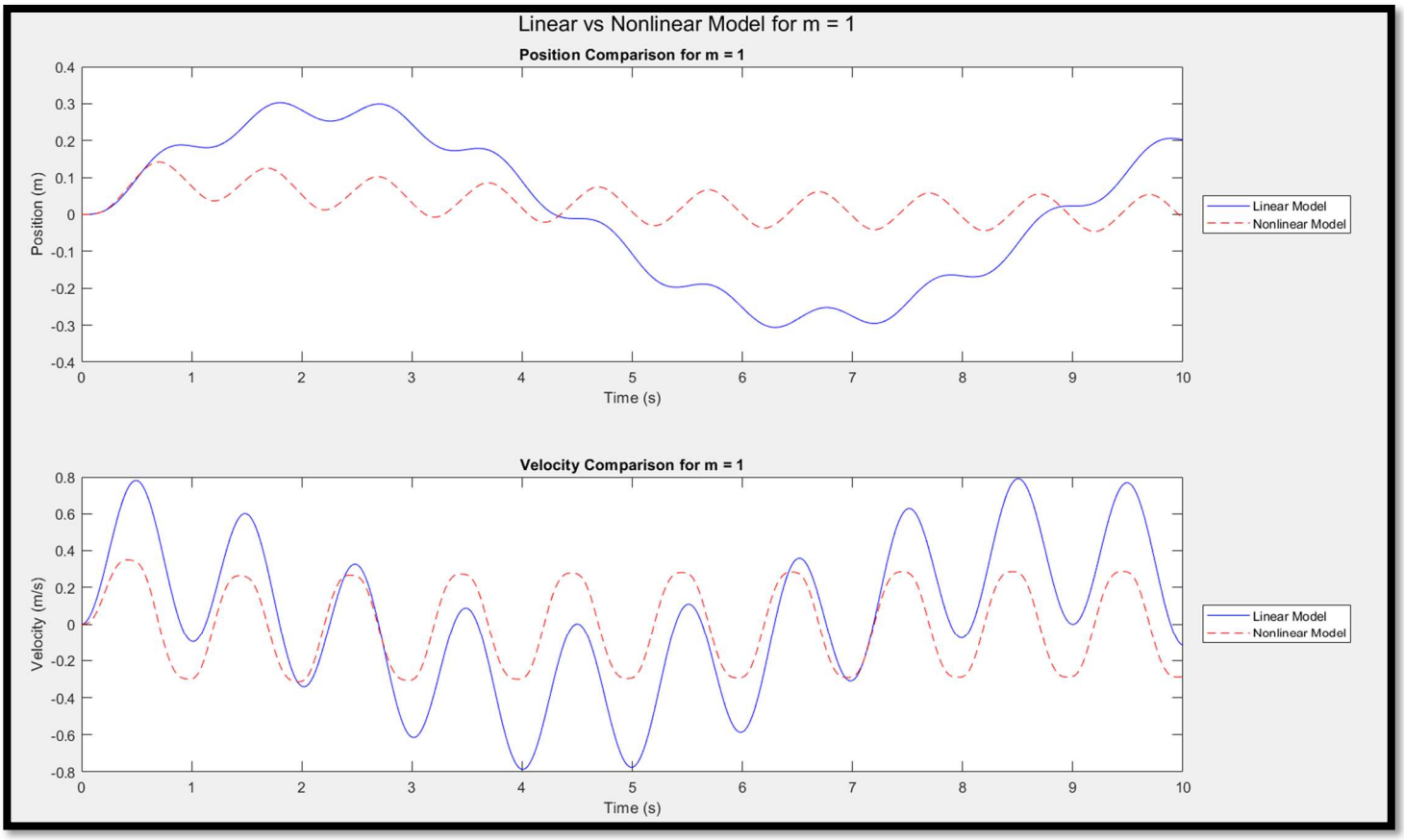
```
% Problem 5 Part C

% Function block_cn_nonlinear with inputs for time vector, mass, stiffness, and
damping coefficient
function [x, v, u] = block_cn_nonlinear(t, m, k, Cd)
    % Initialize arrays for positions (x), velocities (v), and background flow (u)
    N = length(t); % Determine the # of time steps based on the length of the time
vector
    x = zeros(1, N); % Initialize position array with zeros
    v = zeros(1, N); % Initialize velocity array with zeros
    u = zeros(1, N); % Initialize background flow array with zeros for sin(2*pi*t)

    % Calculate the time step size based on the time vector
    dt = t(2) - t(1);

    % Calculate the background flow u(t) for each time step
    for i = 1:N
        u(i) = sin(2 * pi * t(i)); % given in the problem
    end

    % Loop through each time step to calculate position and velocity
    for n = 1:N-1
        % Initial guess for v_{n+1} (could be v_n)
        v_guess = v(n);

        % Perform 10 iterations within each time step to solve for v_{n+1}
        for iter = 1:10
            % Calculate drag forces at time steps n and n+1
            Fd_n = Cd * (u(n) - v(n)) * abs(u(n) - v(n));
            Fd_np_1 = Cd * (u(n+1) - v_guess) * abs(u(n+1) - v_guess);

            % Implicit Crank-Nicolson formula for v_{n+1}
            v_new = v(n) + (dt/(2*m)) * (Fd_n - k*x(n)) + (dt/(2*m)) * (Fd_np_1 -
k*x(n));
```

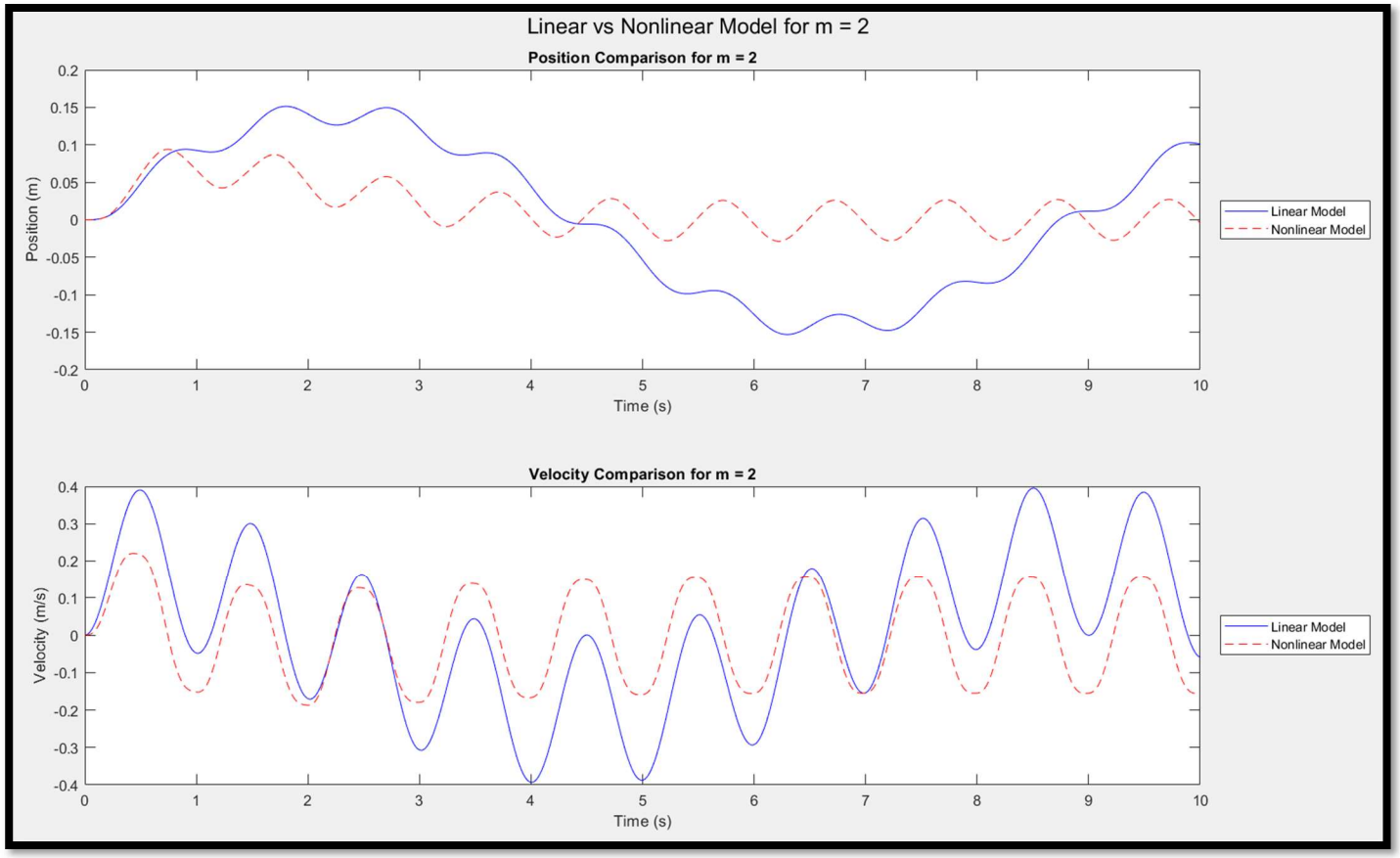
```
            % Update guess for v_{n+1}
            if abs(v_new - v_guess) < 1e-5 % Convergence criterion
                break;
            end
            v_guess = v_new; % make the v_guess the new value
        end

        % Update velocity and position for the next time step
        v(n+1) = v_guess; % based on part b derivation
        x(n+1) = x(n) + dt/2 * (v(n) + v(n+1)); % based on part b derivation
    end
end
```

Problem 5d) Now let's compare the outputs of both of your functions as we vary the mass of the block. Write a script p2d that calls both of your functions with m = 0.1, 0.5, 1, 2, 10 within a for-loop, while keeping k/m = 1 and Cd = 2.5, for t = 0 : 0.01 : 10. For each value of m, generate pairs of figures similar to the ones you produced in b, with the linear and nonlinear model solutions superposed and labelled. Based on your plots, when does the linear drag model provide comparable results to the nonlinear model: is it when the block is relatively lighter or heavier? Why is that?




Linear vs Nonlinear Model for m = 0.5
Position Comparison for m = 0.5
Position (m)
Time (s)
Linear Model
Nonlinear Model
Velocity Comparison for m = 0.5
Velocity (m/s)
Time (s)
Linear Model
Nonlinear Model


Linear vs Nonlinear Model for m = 1
Position Comparison for m = 1
Position (m)
Time (s)
Linear Model
Nonlinear Model
Velocity Comparison for m = 1
Velocity (m/s)
Time (s)
Linear Model
Nonlinear Model



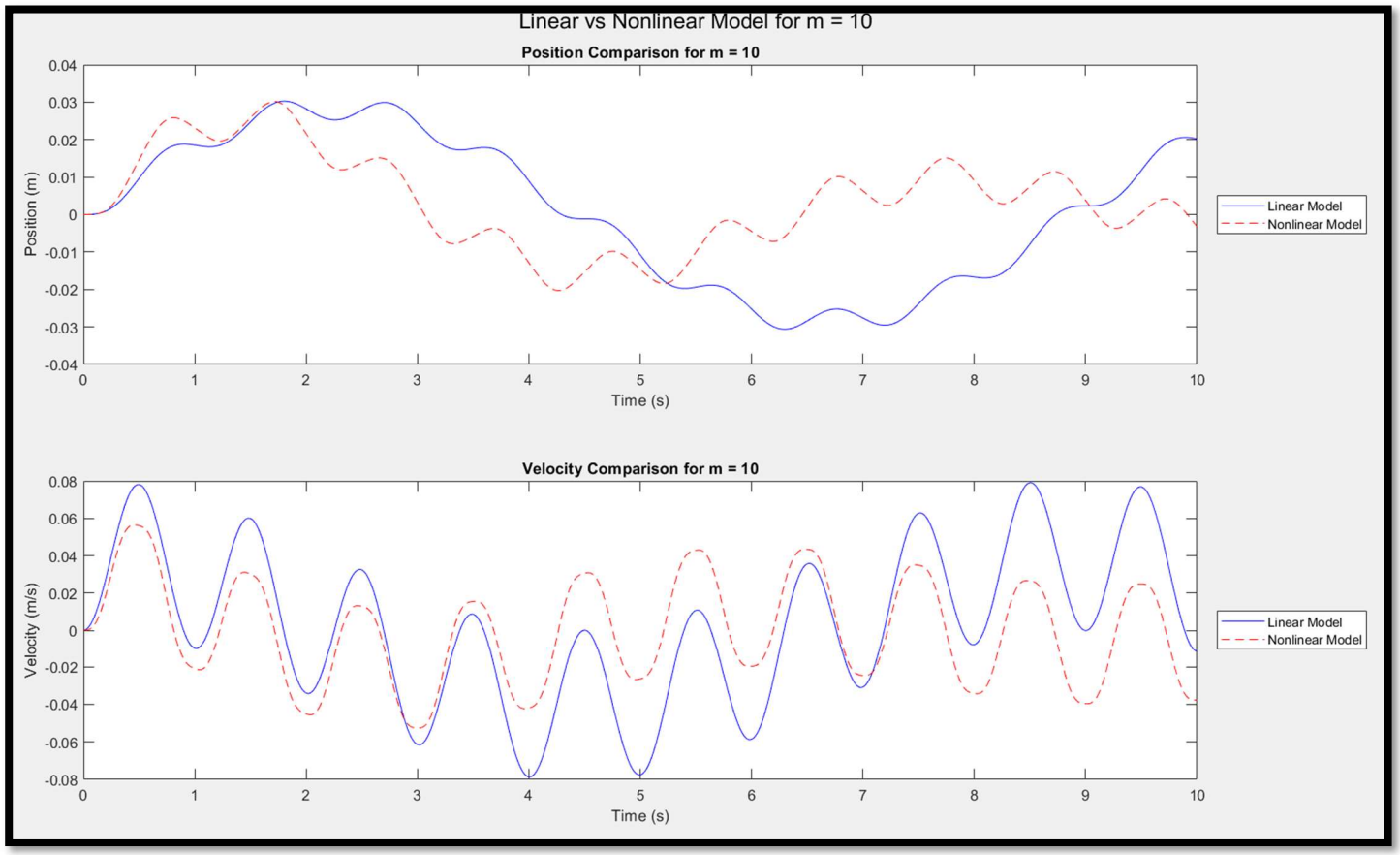


Figure 9: 5 Images of Graphs showing Position and Velocity Comparison of m = 0.1, 0.5, 1, 2, and 10 respectively. Linear and Nonlinear graphs present

Comments:

When comparing the linear and nonlinear drag models as block mass varies, the linear model tends to align more closely with the nonlinear model for heavier blocks. The reason lies in the mass's influence on motion: heavier blocks possess greater inertia, making them less responsive to the same force magnitude. Consequently, velocity changes induced by drag forces are subtler, and the discrepancies between a constant coefficient (as used in the linear model) and a velocity-dependent one (in the nonlinear model) are minimized. Hence, the simplicity of the linear drag force is a better approximation when the block has more mass.

Conversely, for lighter blocks, their lower inertia means they are more susceptible to acceleration changes due to drag, which is significantly affected by the relative velocity between the flow and the block. With lighter masses, this relative velocity can fluctuate more rapidly, highlighting the nonlinear drag's dependency on these changes. As such, the linear model's assumption of constant drag becomes less representative of the actual forces at play, leading to a greater divergence from the nonlinear model's predictions.

Problem 5D Code Script (p2d):

```
% Define the time vector for the simulation
t = 0:0.01:10; % Time vector from 0 to 10 with a step of 0.01

% Constants for the simulation
k_over_m = 1; % Ratio of spring constant to mass, ensuring k/m remains constant
Cd = 2.5; % Damping coefficient
masses = [0.1, 0.5, 1, 2, 10]; % Array of different mass values to simulate

% Loop over each mass value to perform simulations
for m = masses
    % Calculate the spring constant k for the current mass to maintain constant k/m
ratio
    k = k_over_m * m;

    % Call the linear model function for the current mass
    [x_linear, v_linear, u_linear] = block_cn_linear(t, m, k, Cd);

    % Call the nonlinear model function for the current mass
    [x_nonlinear, v_nonlinear, u_nonlinear] = block_cn_nonlinear(t, m, k, Cd);

    figure; % Create a new figure for plotting

    % Create a subplot for position comparison
    ax1 = subplot(2,1,1); % Create a subplot for position (first of two vertical
plots)
    plot(t, x_linear, 'b-', t, x_nonlinear, 'r--'); % Plot both linear and nonlinear
positions
    title(['Position Comparison for m = ', num2str(m)]); % Title for position plot
    xlabel('Time (s)'); % Label for the x-axis
    ylabel('Position (m)'); % Label for the y-axis
```

```
    legend('Linear Model', 'Nonlinear Model', 'Location', 'eastoutside'); % Legend to
the right of the plot

    % Create a subplot for velocity comparison
    ax2 = subplot(2,1,2); % Create a subplot for velocity (second of two vertical
plots)
    plot(t, v_linear, 'b-', t, v_nonlinear, 'r--'); % Plot both linear and nonlinear
velocities
    title(['Velocity Comparison for m = ', num2str(m)]); % Title for velocity plot
    xlabel('Time (s)'); % Label for x-axis
    ylabel('Velocity (m/s)'); % Label for y-axis
    legend('Linear Model', 'Nonlinear Model', 'Location', 'eastoutside'); % Legend to
the right of the plot

    % Adjust layout with a super title for figure
    sgtitle(['Linear vs Nonlinear Model for m = ' num2str(m)]); % Super title for
overall comparison
end
```