جامعـة نيويورك أبـوظـبي

NYU | ABU DHABI

ENGR-UH 2017

Numerical Methods

Final Exam

Fall 2024

Yamsine Elsisi ~ yme2013

N number = N16931391
For the following exercises d = 1

Exercise 1

Problem Explanation:

We have a spherical tank and we want to find the depth h at which it holds a volume V=40 m^3. The radius R depends on a parameter d. We used d = 1 to get R = 5.2 . The equation relating depth h to volume V is solved using three numerical methods: Bisection, Secant, and Newton-Raphson.

Methodology:

1. **Bisection Method:** Starts with an interval that brackets the root and repeatedly halves it until the solution is close enough.
2. **Secant Method:** Uses two initial guesses and repeatedly forms secant lines to find a better approximation.
3. **Newton-Raphson Method:** Uses a single initial guess and the function's derivative to quickly converge to the root.

Code:

```
% Exercise 1

% Given parameter d = 1

d = 1;

% Parameters
R = 5 + 0.2 * d;  % Radius of the tank
V = 40;           % volume (m^3)

% Function h_func
h_func = @(h) ((pi*(h^2))/3)*(3*R - h) - V;

% Convergence tolerance
e = 1e-6;

%% (a) Bisection Method

fprintf('Bisection Method:\n');

a0 = 0.1;   % Initial lower bound
b0 = 10.0;  % Initial upper bound
error_bisec = abs(b0 - a0);  % Initial error estimate
```

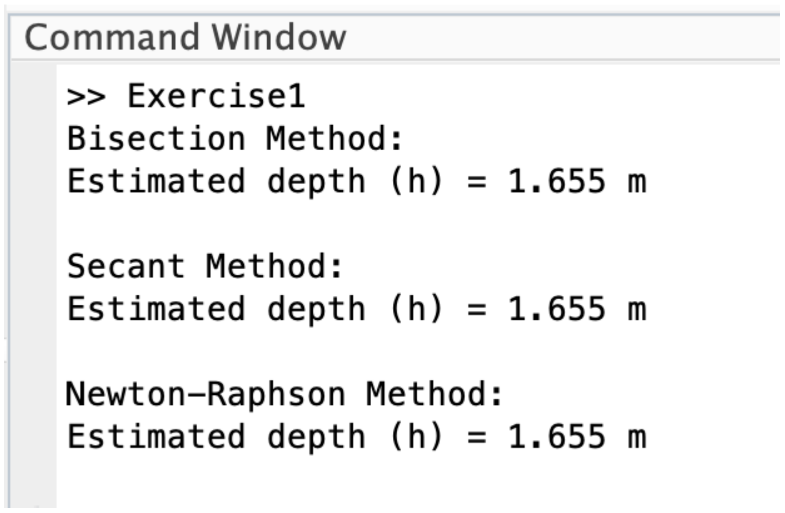
```
% Bisection loop
while error_bisec > e
    h_mid = (a0 + b0) / 2;
    if h_func(a0)*h_func(h_mid) < 0
        b0 = h_mid;
    else
        a0 = h_mid;
    end

    error_bisec = abs(b0 - a0);   % Update error estimate
end

h_bisec = (a0 + b0) / 2;

% Display results
fprintf('Estimated depth (h) = %.3f m\n\n', h_bisec);


%% (b) Secant Method

fprintf('Secant Method:\n');

% Initial conditions
h_1 = 0.5; % h_-1
h_0 = 1.0; % h_0
e = 1e-6;

error_secant = Inf;

while error_secant > e
    f_1 = h_func(h_1);
    f_0   = h_func(h_0);
    h_next = h_0 - f_0 * ((h_0 - h_1) / (f_0 - f_1));
    error_secant = abs(h_next - h_0);

    % Update for next iteration
    h_1 = h_0;
    h_0 = h_next;
end

h_secant = h_0;

fprintf('Estimated depth (h) = %.3f m\n\n', h_secant);

%% (b) Newton-Raphson Method
```

```
fprintf('Newton-Raphson Method:\n');

% Initial guess
h_newton = 1.0;
e = 1e-6;
error_newton = Inf;

% Derivative of h_func(h)
h_prime = @(h) pi*(2*R*h - h^2);

while error_newton > e
    h_next = h_newton - h_func(h_newton)/h_prime(h_newton);
    error_newton = abs(h_next - h_newton);
    h_newton = h_next;
end

fprintf('Estimated depth (h) = %.3f m\n\n', h_newton);
```

Discussion of Code:

- We define h_func(h) for the tank volume equation.
- For the bisection method, we pick an interval [0.1, 10] and narrow it down until the error is less than 1e-6.
- For the secant method, we start from two points (0.5 and 1.0) and iterate until convergence.
- For Newton-Raphson, we start from h=1.0 and iterate using the function derivative.

Results:



```
>> Exercise1
Bisection Method:
Estimated depth (h) = 1.655 m

Secant Method:
Estimated depth (h) = 1.655 m

Newton-Raphson Method:
Estimated depth (h) = 1.655 m
```

Results to two significant figures:-

- Bisection Method: h ≈ 1.7 m
- Secant Method: h ≈ 1.7 m
- Newton-Raphson Method: h ≈ 1.7 m

Disussion of Results:

All three methods yielded approximately the same result. This consistency suggests that the solution is correct and stable. The depth of 1.655 m is the height of water needed in the spherical tank to achieve the desired volume of 40 m^3. The agreement between the methods also shows that the chosen initial guesses and intervals were appropriate.

Exercise 2

Problem Explanation:

We want to compute the magnitude of the magnetic field |B| at a point (x,y,z) due to a circular coil. The integral form of the Biot-Savart law is given, and we use three different numerical integration methods: Trapezoidal rule, Simpson's 1/3 rule, and Gauss-Legendre quadrature.

Methodology:

1. **Trapezoidal Rule:** Approximates the integral by dividing the interval into segments and using trapezoids.
2. **Simpson's Rule:** Uses a parabolic approximation for higher accuracy with fewer segments.
3. **Gauss-Legendre Quadrature:** Uses optimized nodes and weights for very accurate results with fewer points.

Code:

```
%% Exercise 2

d = 1;

% Parameters
m0 = 1;
I = 1;
R = 1;

% Coordinates of Point R
x = 0.5;
y = 0.2 + 0.3*d;
z = 0.5;

% Constant factor in the integral
```

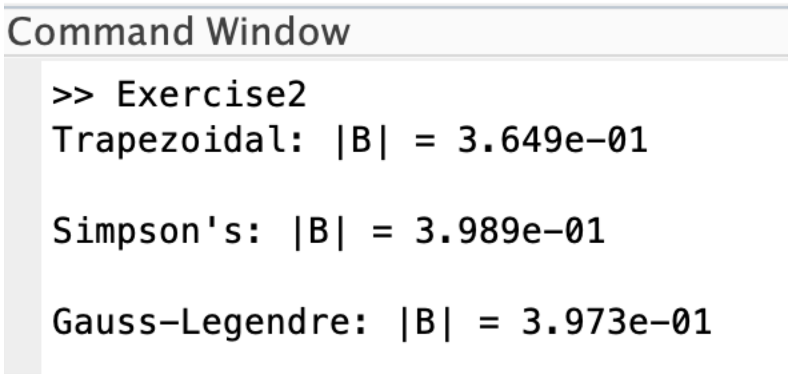
```
const_factor = (m0 * I) / (4 * pi);

% Limits of integration
a = 0;
b = 2*pi;

integrand = @(t) [ ...
    (R*z*cos(t)), ...
    (R*z*sin(t)), ...
    (R^2 - R*(x*cos(t) + y*sin(t))) ] ./ ...
    ( ((x - R*cos(t)).^2 + (y - R*sin(t)).^2 + z^2).^(3/2) );

%% (a) Trapezoidal Rule

n_trap = 15;
t_trap = linspace(a, b, n_trap+1);
f_trap = zeros(n_trap+1, 3);
for i = 1:n_trap+1
    f_trap(i,:) = integrand(t_trap(i));
end

h_trap = (b - a) / n_trap;
Bx_trap = (h_trap/2) * ( f_trap(1,1) + 2*sum(f_trap(2:end-1,1)) + f_trap(end,1) );
By_trap = (h_trap/2) * ( f_trap(1,2) + 2*sum(f_trap(2:end-1,2)) + f_trap(end,2) );
Bz_trap = (h_trap/2) * ( f_trap(1,3) + 2*sum(f_trap(2:end-1,3)) + f_trap(end,3) );

B_trap = const_factor * [Bx_trap, By_trap, Bz_trap];
B_trap_mag = sqrt(sum(B_trap.^2));

fprintf('Trapezoidal: |B| = %.3e\n\n', B_trap_mag);

%% (b) Simpson’s 1/3 Rule

n_simp = 9;
t_simp = linspace(a, b, n_simp);
f_simp = zeros(n_simp,3);
for i = 1:n_simp
    f_simp(i,:) = integrand(t_simp(i));
end

h_simp = (b - a) / (n_simp - 1);

% Indices for Simpson's rule:

odd_ind = 2:2:n_simp-1;   % 2,4,6,8
even_ind = 3:2:n_simp-2; % 3,5,7
```

```
Bx_simp = (h_simp/3) * ( f_simp(1,1) + f_simp(end,1) + 4*sum(f_simp(odd_ind,1)) +
2*sum(f_simp(even_ind,1)) );
By_simp = (h_simp/3) * ( f_simp(1,2) + f_simp(end,2) + 4*sum(f_simp(odd_ind,2)) +
2*sum(f_simp(even_ind,2)) );
Bz_simp = (h_simp/3) * ( f_simp(1,3) + f_simp(end,3) + 4*sum(f_simp(odd_ind,3)) +
2*sum(f_simp(even_ind,3)) );

B_simp = const_factor * [Bx_simp, By_simp, Bz_simp];
B_simp_mag = sqrt(sum(B_simp.^2));

fprintf('Simpson''s: |B| = %.3e\n\n', B_simp_mag);

%% (c) Gauss-Legendre Quadrature

% Gauss-Legendre points and weights for n=5
gauss_points = [-0.9061798459, -0.5384693101, 0, 0.5384693101, 0.9061798459];
gauss_weights = [0.2369268851, 0.4786286705, 0.5688888889, 0.4786286705, 0.2369268851];

% Transforming to interval [0, 2π]
t_points = 0.5 * (gauss_points + 1) * 2 * pi;
weights = gauss_weights * pi;

% Function for the Biot-Savart integral
biot_savart_integrand = @(t) sqrt( ...
    (R * z .* cos(t)).^2 + (R * z .* sin(t)).^2 + ...
    (R^2 - R .* (x .* cos(t) + y .* sin(t))).^2 ...
) ./ ((x - R .* cos(t)).^2 + (y - R .* sin(t)).^2 + z^2).^(3/2);

% Compute the integral
integrand_values = biot_savart_integrand(t_points);
B_gauss_mag = (m0 * I / (4 * pi)) * sum(weights .* integrand_values);

fprintf('Gauss-Legendre: |B| = %.3e\n\n', B_gauss_mag);
```

Discussion of Code:

- We define an integrand for the Biot-Savart integral.
- For trapezoidal and Simpson's, we discretize the interval [0, 2π] and sum up the approximations.
- For Gauss-Legendre, we use predefined nodes and weights for n=5, transform them to our interval, and sum up the weighted integrand values.

Results:



```
Command Window
>> Exercise2
Trapezoidal: |B| = 3.649e-01

Simpson's: |B| = 3.989e-01

Gauss-Legendre: |B| = 3.973e-01
```

Results to two significant figures:-

- Trapezoidal: |B| = 3.6×10^{-1}
- Simpson’s: |B| = 4.0×10^{-1}
- Gauss-Legendre: |B| = 4.0×10^{-1}

Discussion of results:

The trapezoidal rule result, approximately 0.365, is somewhat lower than the other two methods. Simpson’s and Gauss-Legendre results, about 0.399 and 0.397 respectively, are much closer to each other. This suggests that Simpson’s and Gauss-Legendre methods provide more accurate estimates with the chosen number of segments or points.

Exercise 3

Problem Explanation:

We have a system of two nonlinear equations representing the steady-state concentrations of bacteria (X) and substrate (S). We want to solve them using Newton-Raphson to find steady-state values. We also need to compute the determinant of the Jacobian on the first iteration and provide the solution after 5 iterations.

Methodology:

- Newton-Raphson is used here. We compute the system of equations and the Jacobian each iteration.
- Update (X, S) until we have done 5 iterations.

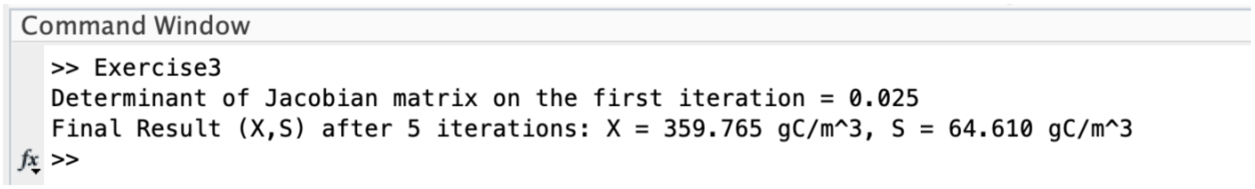
Code:

```
%% Exercise 3

% Given parameters
d = 1;
kg_max = 0.3 + 0.1 * d;
Y = 0.5;
Ks = 120;
kd = 0.01;
kr = 0.01;
Tw = 20;
S_in = 1000;

% Initial guesses for X and S
X = 100;
S = 0;

% Convergence parameters
iterations = 5;

for i = 1:iterations
    % Compute function values and Jacobian
    [F, J] = sysEqu(X, S, kg_max, Y, Ks, kd, kr, Tw, S_in);

    if i == 1
        % (a) The determinant of on the first iteration
        detJ = det(J);
        fprintf('Determinant of Jacobian matrix on the first iteration = %.3f\n', detJ);
    end

    delta = -J \ F;

    % Update X and S
    X = X + delta(1);
    S = S + delta(2);
end

%(b) Final result after 5 iterations
fprintf('Final Result (X,S) after 5 iterations: X = %.3f gC/m^3, S = %.3f gC/m^3\n', X, S);

%% Function to compute the system of equations and Jacobian
function [F, J] = sysEqu(X, S, kg_max, Y, Ks, kd, kr, tau_w, Sin)

    % Computing function values
    f1 = (kg_max * Y * S / (Ks + S)) * X - (kd + kr + 1/tau_w) * X;
    f2 = -(kg_max * S / (Ks + S)) * X + kd * X + (1/tau_w) * (Sin - S);
    F = [f1; f2];

    % Computing partial derivatives
    dfdX1 = (kg_max * Y * S / (Ks + S)) - (kd + kr + 1/tau_w);
    dfdS1 = (kg_max * Y * X * Ks) / ((Ks + S)^2);
    dfdX2 = -(kg_max * S / (Ks + S)) + kd;
    dfdS2 = -(kg_max * X * Ks) / ((Ks + S)^2) - (1/tau_w);
    J = [dfdX1, dfdS1; dfdX2, dfdS2];
end
```

Discussion of Code:

- The function sysEqu calculates F and J at the current (X, S).
- Newton-Raphson update formula is applied 5 times.
- We print the Jacobian determinant after the first iteration and the final (X, S) after the 5th iteration.

Results:



```
Command Window
>> Exercise3
Determinant of Jacobian matrix on the first iteration = 0.025
Final Result (X,S) after 5 iterations: X = 359.765 gC/m^3, S = 64.610 gC/m^3
>>
```

Results to two significant figures:-

- Determinant = 0.03
- (X,S) = 360 gC/m^3 , 65 gC/m^3

Discussion of Results:

The nonzero determinant (0.025) at the start means the system of equations is not stuck or singular right away, allowing the Newton-Raphson method to proceed smoothly. After five iterations, we reached steady values for X and S. This tells us the system settles into a steady state where the bacteria and substrate concentrations no longer change.

Exercise 4

Problem Explanation:

We have a mass-spring-damper system with a nonlinear damping (a * |v| * v). We solve the motion using Euler's method and the Runge-Kutta 4th order method and compare results and final displacement at a given time. We also plot displacement and velocity over time.

Methodology:

- Euler's method: A simple, first-order method stepping through time.
- RK4: A more accurate method that uses four intermediate steps per interval.

Code:

```
%% Exercise 4

% Parameters
m = 1;
a = 5;
k = 6;
d = 1;
F0 = 2.1 + 0.8 * d;
w = 0.5;
h = 0.125;
t_upper = 15;

% Define the force function as a nested function for clarity
force = @(t, v, u) (F0 * sin(w * t) - a * abs(v) * v - k * u) / m;

% Time vector
t = 0:h:t_upper;
n = length(t);

% Initial conditions
initial_u = 1; % Initial displacement at t=0
initial_v = 0; % Initial velocity at t=0

%% Euler's Method
[x_euler, v_euler] = eulerMethod(force, t, h, initial_u, initial_v);

% Display Euler Results
disp('Euler Results:');
fprintf('Displacement at t = %.1f: %.3f m\n', t_upper, x_euler(end));

%% Runge-Kutta (RK4) Method
[x_rk, v_rk] = rungeKuttaMethod(force, t, h, initial_u, initial_v);
```

```matlab
% Display RK4 Results
disp('Runge-Kutta Results:');
fprintf('Displacement at t = %.1f: %.3f m\n', t_upper, x_rk(end));

%% Plot Results
figure;

% Euler's Method subplot
subplot(2, 1, 1);
plot(t, x_euler, 'b-', 'LineWidth', 1.25, 'DisplayName', 'Displacement');
hold on;
plot(t, v_euler, 'm--', 'LineWidth', 1.25, 'DisplayName', 'Velocity');
xlabel('Time (sec)');
ylabel('Displacement (m) / Velocity (m/s)');
title('Euler''s Method');
legend('Location', 'best');
grid on;

% Runge-Kutta Method subplot
subplot(2, 1, 2);
plot(t, x_rk, 'b-', 'LineWidth', 1.25, 'DisplayName', 'Displacement');
hold on;
plot(t, v_rk, 'm--', 'LineWidth', 1.25, 'DisplayName', 'Velocity');
xlabel('Time (sec)');
ylabel('Displacement (m) / Velocity (m/s)');
title('Fourth-order Runge-Kutta Method');
legend('Location', 'best');
grid on;

%% Functions

function [x_vals, v_vals] = eulerMethod(forceFunc, t, h, u0, v0)
    % Preallocate arrays
    x_vals = zeros(size(t));
    v_vals = zeros(size(t));

    % Initial conditions
    x_vals(1) = u0;
    v_vals(1) = v0;

    for i = 1:length(t)-1
        du_dt = v_vals(i);
        dv_dt = forceFunc(t(i), v_vals(i), x_vals(i));

        x_vals(i+1) = x_vals(i) + h * du_dt;
        v_vals(i+1) = v_vals(i) + h * dv_dt;
    end
end

function [x_vals, v_vals] = rungeKuttaMethod(forceFunc, t, h, u0, v0)
    % Preallocate arrays
    x_vals = zeros(size(t));
    v_vals = zeros(size(t));

    % Initial conditions
    x_vals(1) = u0;
    v_vals(1) = v0;

    for i = 1:length(t)-1
        % k1 terms
        k1u = v_vals(i);
        k1v = forceFunc(t(i), v_vals(i), x_vals(i));

        % k2 terms
        k2u = v_vals(i) + 0.5 * h * k1v;
        k2v = forceFunc(t(i) + 0.5 * h, v_vals(i) + 0.5*h*k1v, x_vals(i) + 0.5*h*k1u);

        % k3 terms
        k3u = v_vals(i) + 0.5 * h * k2v;
        k3v = forceFunc(t(i) + 0.5 * h, v_vals(i) + 0.5*h*k2v, x_vals(i) + 0.5*h*k2u);

        % k4 terms
        k4u = v_vals(i) + h * k3v;
        k4v = forceFunc(t(i) + h, v_vals(i) + h*k3v, x_vals(i) + h*k3u);

        x_vals(i+1) = x_vals(i) + (h/6)*(k1u + 2*k2u + 2*k3u + k4u);
        v_vals(i+1) = v_vals(i) + (h/6)*(k1v + 2*k2v + 2*k3v + k4v);
    end
end
```

Discussion of Code:

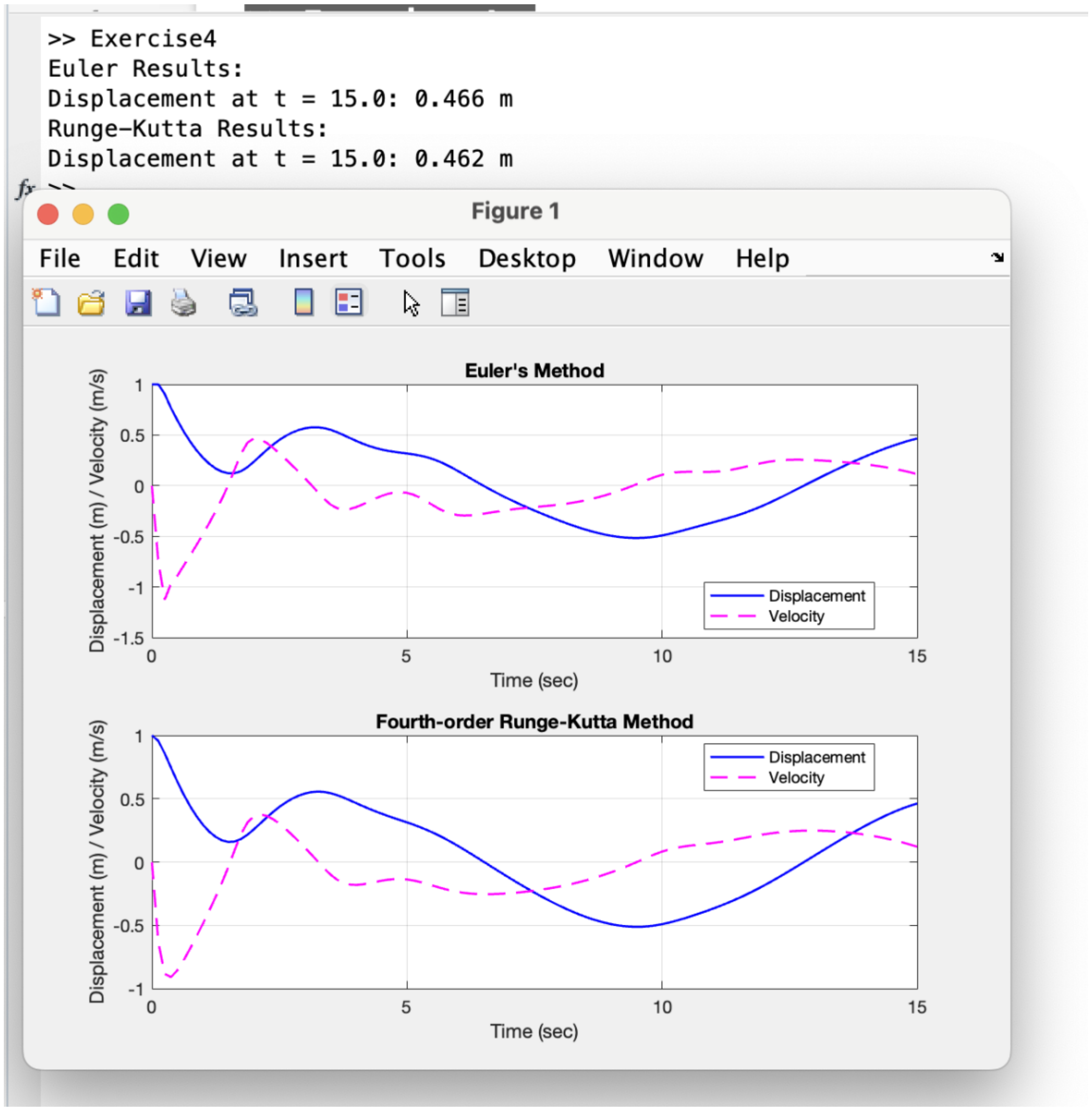
- We implement Euler and RK4 in separate functions.
- The force function defines the acceleration at any given time and state.
- We then run both methods and print the final displacement at t = 15s.
- Two subplots show displacement and velocity for both methods.

Results and discussion:

Results to two significant figures:-

- Euler displacement = 0.47 m
- Runge-Kutta displacement = 0.46 m

Both methods produce similar results, but the Runge-Kutta method is generally more accurate than Euler's, especially over longer times. The small difference in final displacement (0.466 m vs. 0.462 m) shows that Euler's method introduces slightly more numerical error. The plotted results clearly show that while both methods capture the general motion, Runge-Kutta provides a smoother and more reliable approximation of the system's true behavior.



```
>> Exercise4
Euler Results:
Displacement at t = 15.0: 0.466 m
Runge-Kutta Results:
Displacement at t = 15.0: 0.462 m
>>
```

Exercise 5

Problem Explanation:

We are solving a Couette flow problem, where fluid velocity between two plates is described by a differential equation. We use a finite difference approach to approximate the velocity at discrete points.

Methodology:

- Set up a grid of points from y=0 to y=L.
- Formulate a linear system from the discretized differential equation.
- Solve the system to get the velocity at each grid point.
- Compare with the exact solution and print the results.

Code:

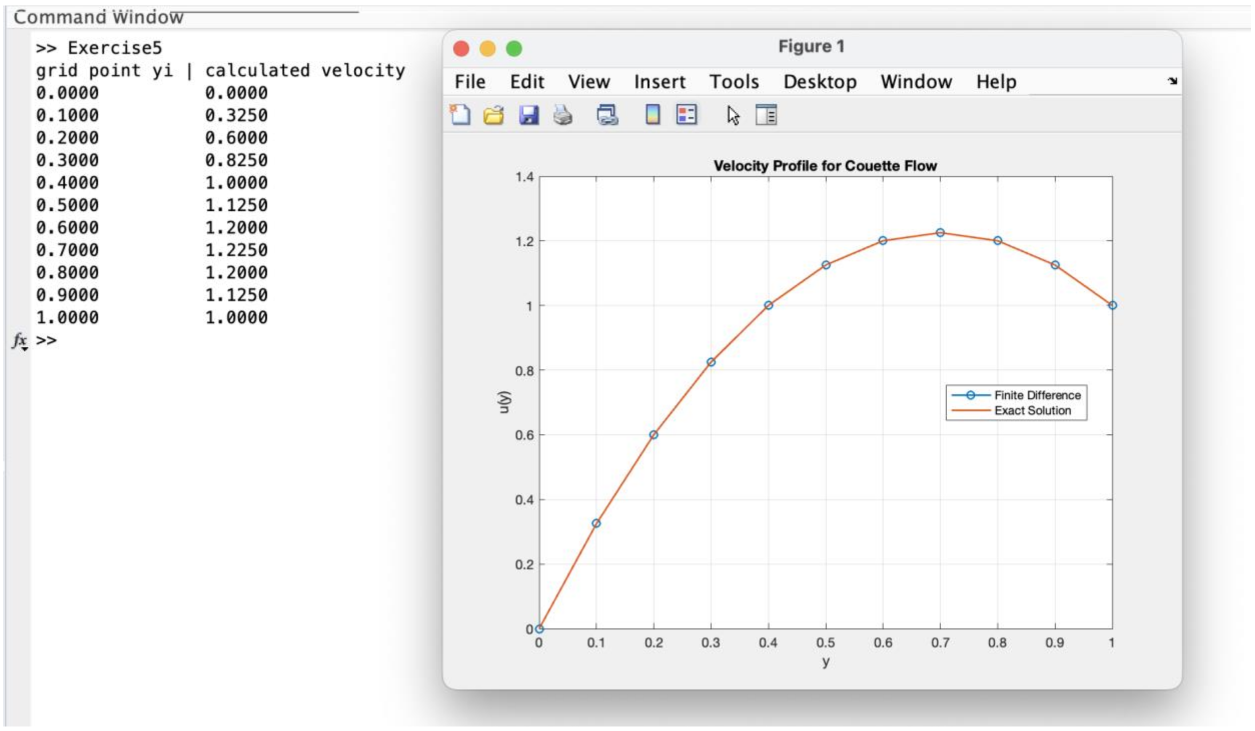
```
% Exercise 5

% Parameters
G = 1.0;
d = 1;
m = 0.1 * (1 + d);
L = 1;
U = 1;

n = 10;        % Number of segments
h = L / n;     % Step size
y = linspace(0, L, n+1); % Grid points

% coefficient matrix
A = zeros(n-1, n-1);
b = -G / m * ones(n-1, 1);

% Fill the tridiagonal matrix
for i = 1:n-1
    if i > 1
        A(i, i-1) = 1 / h^2;
    end
    A(i, i) = -2 / h^2;
    if i < n-1
        A(i, i+1) = 1 / h^2;
    end
end
```

```
b(end) = b(end) - U / h^2;

% Solve the linear system
u_fd = A \ b;
u_fd = [0; u_fd; U]; % boundary values at y=0 and y=L

% Exact solution
u_exact = (G / (2 * m)) * y .* (L - y) + (U / L) * y;

% Plot the results
figure;
plot(y, u_fd, 'o-', 'LineWidth', 1.25, 'DisplayName', 'Finite Difference');
hold on;
plot(y, u_exact, '-', 'LineWidth', 1.25, 'DisplayName', 'Exact Solution');
xlabel('y');
ylabel('u(y)');
title('Velocity Profile for Couette Flow');
legend('Location', 'best');
grid on;

% Print results in requested format
fprintf('grid point yi | calculated velocity\n');
for i = 1:length(y)
    fprintf('%.4f          %.4f\n', y(i), u_fd(i));
end
```

Discussion of Code:

- We create a tridiagonal matrix A and vector b.
- Solve A u = b for velocity u.
- Compare and plot both the finite difference and exact solutions.
- Print the results in a table-like format.

Results:



```
>> Exercise5
grid point yi | calculated velocity
0.0000          0.0000
0.1000          0.3250
0.2000          0.6000
0.3000          0.8250
0.4000          1.0000
0.5000          1.1250
0.6000          1.2000
0.7000          1.2250
0.8000          1.2000
0.9000          1.1250
1.0000          1.0000
>>
```

Results to two significant figures:-

y (m)	**u(y) (m/s)**
0.00	0.00
0.10	0.33
0.20	0.60
0.30	0.83
0.40	1.0
0.50	1.1
0.60	1.2

y (m)	u(y) (m/s)
0.70	1.2
0.80	1.2
0.90	1.1
1.0	1.0

Discussion of Results:

The results show how the fluid's velocity changes from one plate to the other in a Couette flow setup. At the lower plate (y=0), the velocity is zero since the fluid adheres to the plate surface (no-slip condition). As we move away from the lower plate, the velocity increases, reaching a peak at around the mid-region between the plates. After reaching its maximum, the velocity starts decreasing again as we approach the upper plate, which has a fixed velocity (1 m/s in this case).

The finite difference results closely follow the exact solution, confirming that our numerical method works well for this problem. The small differences are due to discretization and rounding.

Exercise 6

Problem Explanation:

We find the peak altitude of a projectile under drag using the Newton-Raphson method. We know velocity and its derivative, and we solve for the time when the vertical velocity is zero (peak point).

Methodology:

- Define velocity(t) and its derivative.
- Use Newton-Raphson to find t_peak where velocity(t_peak)=0.
- Compute altitude at t_peak.
- Plot altitude vs. time and mark the peak.

Code:

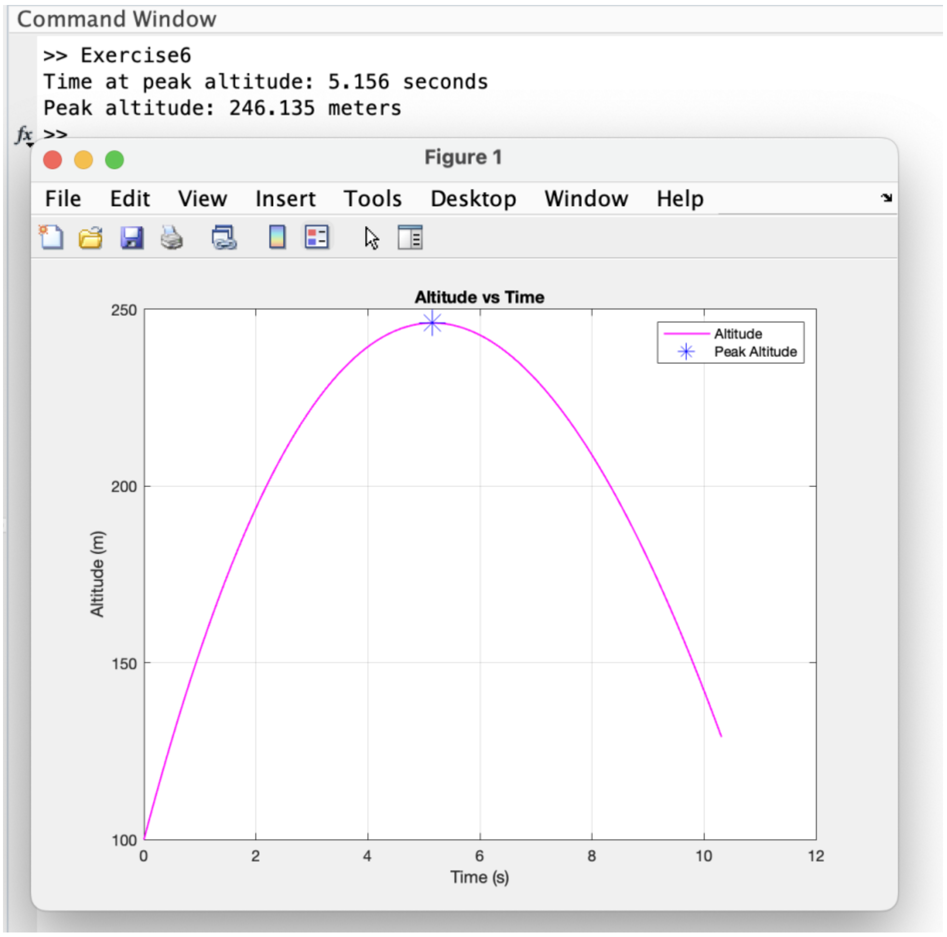
```
% Exercise 6

% Parameters
z0 = 100;
v0 = 60;
g = 9.81;
m = 90;
d = 1;
c = 5 + 0.8 * d;

% Newton-Raphson Parameters
e = 1e-6; % Convergence tolerance
i_max = 5; % Maximum number of iterations
t_guess = 0; % Initial guess for peak time

% Define Velocity and its Derivative
velocity = @(t) (v0 + (m * g) / c) * exp(-c * t / m) - (m * g) / c;
velocity_derivative = @(t) -(c / m) * (v0 + (m * g) / c) * exp(-c * t / m);

% Newton-Raphson Method to Find Peak Time
t_peak = t_guess;
for iter = 1:i_max
    t_new = t_peak - velocity(t_peak) / velocity_derivative(t_peak); % Update time
    if abs(t_new - t_peak) < e
        break;
    end
    t_peak = t_new; % Update the estimate
end

% Compute Peak Altitude
z_peak = z0 + (m / c) * (v0 + (m * g) / c) * (1 - exp(-c * t_peak / m)) - (m * g / c) * t_peak;

% Display Results
fprintf('Time at peak altitude: %.3f seconds\n', t_peak);
fprintf('Peak altitude: %.3f meters\n', z_peak);

% Altitude as a Function of Time
t_vals = linspace(0, 2 * t_peak, 100); % Time range for plotting
altitude = @(t) z0 + (m / c) * (v0 + (m * g) / c) * (1 - exp(-c * t / m)) - (m * g / c) * t;
z_vals = altitude(t_vals);

% Plot Altitude vs Time
figure;
plot(t_vals, z_vals, 'm-', 'LineWidth', 1, 'DisplayName', 'Altitude');
```

```
hold on;
plot(t_peak, z_peak, 'b*', 'MarkerSize', 15, 'DisplayName', 'Peak Altitude');
xlabel('Time (s)');
ylabel('Altitude (m)');
title('Altitude vs Time');
grid on;
legend show;
```

Discussion of Code:

- Newton-Raphson is applied to the velocity function to find when it crosses zero.
- Once t_peak is found, we plug it into the altitude equation.
- The plot shows the altitude curve and the peak point.

Results:



```
>> Exercise6
Time at peak altitude: 5.156 seconds
Peak altitude: 246.135 meters
>>
```

Results to two significant figures:-

- Time at peak alitiude = 5.2 s
- Peak altitude = 250 m

Discussion of Results:

The Newton-Raphson method gave us a peak altitude time of about 5.156 seconds, at which point the projectile reached roughly 246.135 meters. This is expected, because, as time passes, the projectile slows down due to gravity and drag until its vertical velocity hits zero at the peak. After this point, it starts falling back down.
The smooth curve in the plot, and the clearly defined maximum point, show that our approach accurately captured the projectile's motion under the given conditions. The presence of drag and gravity is evident, as the peak altitude occurs at a finite time and height, rather than continuing indefinitely.