

Task 1

please convert this to pdf for submission

Table of Contents

(i) (e) $y = \sin(x)$ plot [1].....	1
(ii) (e) $y = \sin_scale(x,sf)$ plot [1].....	1
(ii) (e) $y = \sin$ array plot [3].....	1
Numerical Differentiation $y=\cos(x)$ plot [3].....	2
Numerical Intergration (a) $\int \sin(x) dx$ [3].....	2
Numerical Intergration (b) compare [3].....	3
Comp Trap (c) $1/x$ for 5,50,100 [4].....	3
Comp Trap (d) compare [1].....	4
Comp Trap (e) to tolerance [3].....	4
Simpsons (b) $1/x$ for 5,50,100 [4].....	5
Simpsons (c) compare [1].....	7
Simpsons (d) to tolerance [3].....	7

(i) (e) $y = \sin(x)$ plot [1]

```
N = 100;  
x = linspace(-pi,pi,N)';  
y = sin(x); figure(1), plot(x,y)
```

(ii) (e) $y = \sin_scale(x,sf)$ plot [1]

```
sf = 0.1;  
y = sin_scale(x,sf);  
figure(2),  
plot(x,y)
```

```
function y = sin_scale(x,sf)  
%UNTITLED Summary of this function goes here  
% Detailed explanation goes here  
y = sin(x./sf);  
end
```

(ii) (e) $y = \sin$ array plot [3]

```
sf = [0.1 0.5 2];  
N_sf = numel(sf);  
sin_array = zeros(N,N_sf);  
  
for ndx = 1:N_sf  
    sin_array(:,ndx) = sin_scale(x,sf(ndx));  
end  
figure(3),plot(x,sin_array), xlim([-pi,pi])  
title(['limits = [-pi pi]; Scale factors = ',num2str(sf)])
```

Numerical Differentiation $y=\cos(x)$ plot [3]

```
% MATLAB script for numerical differentiation using FDT

% (a) Define input vector x
N = 100; % Number of points
x = linspace(-pi, pi, N)'; % Define x vector

% (b) Calculate y = cos(x)
y = cos(x);

% (d) Estimate the derivative using the defined function
dydx_fdt = forward_difference(x, y);

% Print the estimated derivative to the screen
disp('Estimated Cosine Derivative using Forward Difference Technique:');
disp(dydx_fdt);

% (e) Plot the FDT derivative against the intrinsic MATLAB function '-sin'
figure;
plot(x, dydx_fdt, 'b-', 'LineWidth', 1.5); % Plot FDT derivative
hold on;
plot(x, -sin(x), 'r--', 'LineWidth', 1.5); % Plot true derivative using -sin
hold off;

% Add labels, legend, and title
xlabel('x');
ylabel('Derivative');
title('Numerical Differentiation Using Forward Difference Technique');
legend('FDT Derivative', '-sin(x)', 'Location', 'Best');
grid on;

% Function definition must appear at the end of the script
function dydx = forward_difference(x, y)
    % Input: x (vector of x values), y (vector of y values)
    % Output: dydx (forward difference approximation of derivative)

    N = length(x); % Number of points
    h = x(2) - x(1); % Step size (assumed constant)
    dydx = zeros(N, 1); % Preallocate derivative array

    % Forward difference method v
    for i = 1:N-1
        dydx(i) = (y(i+1) - y(i)) / h;
    end
    dydx(N) = NaN; % Last point cannot be computed using forward difference
end
```

Numerical Intergration (a) $\int \sin(x) dx$ [3]

```
% Numerical integration Main Script: Numerical Integration using Composite Trapezium Rule
```

```
% Part 1: Calculate integral for x*sin(x)
f = @(x) x .* sin(x); % Function handle for x*sin(x)
a = 0; % Lower limit
b = pi; % Upper limit
n = 100; % Number of intervals

% Calculate the integral using composite trapezium rule
numerical_integral = composite_trapezium(f, a, b, n);

% Compute the analytic solution
syms x;
analytic_integral = int(x * sin(x), x, 0, pi); % Symbolic integral
analytic_value = double(analytic_integral); % Convert to numeric
```

Numerical Intergration (b) compare [3]

```
% Calculate percentage error
percentage_error = abs((analytic_value - numerical_integral) / analytic_value) * 100;

disp(['Numerical integral: ', num2str(numerical_integral)]);
disp(['Analytic integral: ', num2str(analytic_value)]);
disp(['Percentage error: ', num2str(percentage_error), '%']);
```

Comp Trap (c) 1/x for 5,50,100 [4]

```
% Part 1: Calculate integral for x*sin(x)
f = @(x) x .* sin(x); % Function handle for x*sin(x)
a = 0; % Lower limit
b = pi; % Upper limit
n = 100; % Number of intervals

% Calculate the integral using composite trapezium rule
numerical_integral = composite_trapezium(f, a, b, n);

% Compute the analytic solution
syms x;
analytic_integral = int(x * sin(x), x, 0, pi); % Symbolic integral
analytic_value = double(analytic_integral); % Convert to numeric

% Calculate percentage error
percentage_error = abs((analytic_value - numerical_integral) / analytic_value) * 100;
```

```

disp(['Numerical integral: ', num2str(numerical_integral)]);
disp(['Analytic integral: ', num2str(analytic_value)]);
disp(['Percentage error: ', num2str(percentage_error), '%']);

% Part 2: Evaluate 1/x for different intervals
f = @(x) 1 ./ x; % Function handle for 1/x
a = 1; % Lower limit
b = 2; % Upper limit
n_values = [5, 50, 100]; % Number of intervals

% Compute the integrals for different n values
results = zeros(length(n_values), 2); % Store results [n, integral]
for i = 1:length(n_values)
    n = n_values(i);
    results(i, 1) = n; % Store n
    results(i, 2) = composite_trapezium(f, a, b, n); % Numerical integral
end

% Compute the analytic solution
analytic_value = log(2);

% Display the results
disp('n\tNumerical Integral\tPercentage Error');
for i = 1:length(n_values)
    percentage_error = abs((analytic_value - results(i, 2)) / analytic_value) * 100;
    fprintf('%d\t%.6f\t\t%.6f%%\n', results(i, 1), results(i, 2), percentage_error);
end

```

Comp Trap (d) compare [1]

```

% Display the table for results
T = table(results(:, 1), results(:, 2), abs((analytic_value - results(:, 2)) /
analytic_value * 100), ...
    'VariableNames', {'Intervals', 'NumericalIntegral', 'PercentageError'});
disp(T);

```

Comp Trap (e) to tolerance [3]

```

% Display the table for results
T = table(results(:, 1), results(:, 2), abs((analytic_value - results(:, 2)) /
analytic_value * 100), ...
    'VariableNames', {'Intervals', 'NumericalIntegral', 'PercentageError'});
disp(T);

% Part 3: Determine number of intervals needed for 6-decimal accuracy
tolerance = 1e-6; % 6 decimal places
n_needed = find_required_intervals(f, a, b, log(2), tolerance);

disp(['Number of intervals needed for 6-decimal accuracy: ', num2str(n_needed)]);

```

```

% Function to determine number of intervals needed for 6-decimal accuracy
function n_needed = find_required_intervals(f, a, b, analytic_value, tolerance)
    n = 1; % Start with one interval
    error = Inf; % Initialise error
    while error > tolerance
        n = n * 2; % Double the number of intervals
        numerical_integral = composite_trapezium(f, a, b, n);
        error = abs((analytic_value - numerical_integral)); % Absolute error
    end
    n_needed = n;
end

```

Simpsons (b) $1/x$ for 5,50,100 [4]

```

% Define the function for 1/x
f = @(x) 1 ./ x; % Function handle for 1/x
a = 1; % Lower limit
b = 2; % Upper limit
n_values = [5, 50, 100]; % Number of intervals

% Compute the integrals for different n values
results = zeros(length(n_values), 2); % Store results [n, integral]
analytic_value = log(2); % Analytical solution

for i = 1:length(n_values)
    n = n_values(i);
    if mod(n, 2) ~= 0
        n = n + 1; % Ensure n is even for Simpson's rule
        warning('Number of intervals increased to %d to satisfy Simpson''s rule.',
n);
    end
    results(i, 1) = n; % Store n
    results(i, 2) = simpsons_rule(f, a, b, n); % Numerical integral
end

% Display the results
disp('n\tNumerical Integral\tPercentage Error');
for i = 1:length(n_values)
    percentage_error = abs((analytic_value - results(i, 2)) / analytic_value) * 100;
    fprintf('%d\t%.6f\t\t%.6f%%\n', results(i, 1), results(i, 2), percentage_error);
end

% Display the table for results
T = table(results(:, 1), results(:, 2), abs((analytic_value - results(:, 2)) /
analytic_value * 100), ...
    'VariableNames', {'Intervals', 'NumericalIntegral', 'PercentageError'});
disp(T);

% Part 4: Determine number of intervals needed for 6-decimal accuracy
tolerance = 1e-6; % 6 decimal places

```

```

% Function to determine the required number of intervals
n_needed = find_required_intervals_simpsons(f, a, b, analytic_value, tolerance);

disp(['Number of intervals needed for 6-decimal accuracy: ', num2str(n_needed)]);

% Helper Function to determine required intervals
function n_needed = find_required_intervals_simpsons(f, a, b, analytic_value,
tolerance)
    n = 2; % Start with two intervals (must be even for Simpson's rule)
    error = Inf; % Initialize error
    while error > tolerance

        function integral = simpsons_rule(f, a, b, n)

```

Function definition is misplaced or improperly nested.

```

% simpsons_rule evaluates the integral using Simpson's rule
% Input:
%   f - function handle of the integrand
%   a, b - integration limits
%   n - number of intervals (must be even)
% Output:
%   integral - numerical approximation of the integral

% Validate input
if ~isscalar(n) || n <= 0 || mod(n, 1) ~= 0
    error('Number of intervals (n) must be a positive integer.');
```

end

```

if mod(n, 2) ~= 0
    error('Number of intervals (n) must be even for Simpson's rule.');
```

end

```

% Step size
delta_x = (b - a) / n;

% Compute the x values
x = linspace(a, b, n + 1);

% Compute the f(x) values
fx = arrayfun(f, x);

% Simpson's rule formula
integral = (delta_x / 3) * (fx(1) + 4 * sum(fx(2:2:end-1)) + 2 *
sum(fx(3:2:end-2)) + fx(end));
end

```

Simpsons (c) compare [1]

```
% Display the table for results
T = table(results(:, 1), results(:, 2), abs((analytic_value - results(:, 2)) /
analytic_value * 100), ...
    'VariableNames', {'Intervals', 'NumericalIntegral', 'PercentageError'});
disp(T);
```

Simpsons (d) to tolerance [3]

```
% Function to determine the required number of intervals
n_needed = find_required_intervals_simpsons(f, a, b, analytic_value, tolerance);

disp(['Number of intervals needed for 6-decimal accuracy: ', num2str(n_needed)]);
```