Problem Statement and Reference Solution Setup

The practical assignment focuses on the numerical solution of the initial value problem (Cauchy problem) for a first-order ordinary differential equation:

$$\frac{dy}{dt} = f(t, y) = -2ty, \quad y(0) = y_0,$$

on the interval $t \in [0, 3]$. This equation admits an analytical solution $y(t) = e^{-t^2}$, but for the purpose of methodological consistency with problems lacking closed-form solutions, a high-accuracy numerical reference is employed.

The block of code initializes the problem parameters and constructs a highly accurate reference solution using MATLAB's adaptive Runge–Kutta solver `ode45`. The solver is configured with stringent tolerance settings (`RelTol = 1e-10`, `AbsTol = 1e-12`) to ensure that the computed trajectory is sufficiently precise to serve as a surrogate for the exact solution in error analysis. The resulting discrete solution is then wrapped into a continuous interpolating function `y_reference(t)` via cubic spline interpolation, enabling evaluation at arbitrary time points—particularly at the uniform grid nodes used by the implemented numerical methods.

This reference solution plays a critical role in the subsequent stages of the assignment: it serves as the ground truth against which the global errors of the explicit Euler, improved Euler (explicit trapezoid), RK4, and other methods are quantified. By using a robust, high-order adaptive solver with tight tolerances, the reference minimizes the influence of its own numerical error, ensuring that the observed discrepancies primarily reflect the intrinsic accuracy of the tested methods rather than limitations of the benchmark itself.

```
clear; clc; close all;

%% 1. Problem Definition
f = @(t, y) -2 * t * y;   % dy/dt = -2*t*y
y0 = 1;                   % Initial condition y(0) = 1
t0 = 0;                   % Start time
t_end = 3;                % End time
h = 0.1;                 % Step size for main solution
N = (t_end - t0) / h;     % Number of steps
t_grid = linspace(t0, t_end, N+1);  % Time grid

%% 2. High-Accuracy Numerical Reference Solution using ode45
[t_ode45, y_ode45] = ode45(f, [t0, t_end], y0,
odeset('RelTol',1e-10,'AbsTol',1e-12));
y_reference = @(t) interp1(t_ode45, y_ode45, t, 'spline');
```

1. Euler Method: Formal Description

1.1. Cauchy Problem Statement

Let $f : \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$ be a continuous function satisfying the Lipschitz condition in its second argument over a domain containing the initial point (t_0, y_0). Consider the initial value problem for a system of ordinary differential equations:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}), \quad \mathbf{y}(t_0) = \mathbf{y}_0,$$

where t is the independent variable, $y(t)$ is the unknown function to be determined, and $f(t, y)$ is a given continuous function satisfying the Lipschitz condition with respect to y in some neighborhood of the initial point (t_0, y_0). This condition guarantees the existence and uniqueness of a local solution.

The goal of numerical solution is to construct an approximate value of the function $y(t)$ at discrete points within the interval $[t_0, T]$.

1.2. Derivation of the Iterative Formula

To construct a numerical method, we introduce a uniform grid over the interval $[t_0, T]$, defined by the step size $h = \frac{T - t_0}{N}$, where N is the number of subintervals. The grid nodes are given by:

$$t_n = t_0 + nh, \quad n = 0, 1, \dots, N.$$

Let y_n denote the approximate value of the solution at point t_n, i.e., $y_n \approx y(t_n)$. Using the first-order Taylor expansion, we obtain:

$$y(t_{n+1}) = y(t_n + h) = y(t_n) + h \cdot y'(t_n) + O(h^2).$$

Substituting the derivative expression $y'(t_n) = f(t_n, y(t_n))$, we arrive at the approximation:

$$y(t_{n+1}) \approx y(t_n) + h \cdot f(t_n, y(t_n)).$$

Replacing the exact values $ y(t_n) $ with their approximations y_n, we obtain the **iterative formula of Euler's method**:

$$y_{n+1} = y_n + h \cdot f(t_n, y_n), \quad n = 0, 1, \dots, N - 1,$$

with initial condition $y_0 = y(t_0)$.

1.3. Algorithmic Implementation

The Euler method constructs a sequence of approximations $\{\mathbf{y}_n\}_{n=0}^{N}$, where $\mathbf{y}_n \approx \mathbf{y}(t_n)$, according to the following recurrence relation:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h \cdot \mathbf{f}(t_n, \mathbf{y}_n), \quad n = 0, 1, \dots, N - 1,$$

with initial condition $\mathbf{y}_0 = \mathbf{y}(t_0)$.

This recurrence defines an explicit one-step numerical scheme, wherein each approximation $\mathbf{y}_{n+1}$ is computed directly from the previous state $\mathbf{y}_n$ and the evaluation of the vector field $\mathbf{f}$ at the point $(t_n, \mathbf{y}_n)$.

The algorithm terminates when $n = N$, yielding the discrete trajectory $\{(t_n, \mathbf{y}_n)\}_{n=0}^{N}$ as the numerical solution to the initial value problem.

In computational implementation, the method proceeds iteratively:

1. Initialize t_0 and $\mathbf{y}_0$ as given by the initial condition.

2. For each n from 0 to $N-1$:

   - Evaluate the right-hand side: $\mathbf{k}_1 = \mathbf{f}(t_n, \mathbf{y}_n)$.
   - Compute the next state: $\mathbf{y}_{n+1} = \mathbf{y}_n + h \cdot \mathbf{k}_1$.
   - Advance time: $t_{n+1} = t_n + h$.

3. Return the arrays $\{t_n\}$ and $\{\mathbf{y}_n\}$ as the numerical solution.

The method requires exactly N evaluations of the function $\mathbf{f}$ and N vector additions, making it computationally inexpensive per step. However, due to its first-order global convergence, achieving high accuracy necessitates small step sizes, which may lead to increased cumulative computational cost and potential accumulation of rounding errors.

This algorithm serves as the foundational scheme for more advanced numerical integrators and provides a clear illustration of the principles underlying explicit time-stepping methods for ordinary differential equations.

1.4. Accuracy and Convergence Analysis

The local truncation error of Euler's method — that is, the error introduced at one step assuming the previous value y_n is exact — is of order $O(h^2)$. This follows directly from the Taylor expansion.

The global error, accumulated over the entire integration interval, is defined as:

$$E = \max_{0 \le n \le N} |y(t_n) - y_n|.$$

For Euler's method, the global error is of order $O(h)$, meaning the method has **first-order convergence**. Thus, reducing the step size by a factor of k reduces the global error approximately by the same factor k. This makes Euler's method simple but relatively slow to converge compared to more advanced schemes such as Runge-Kutta methods.

1.5. Stability

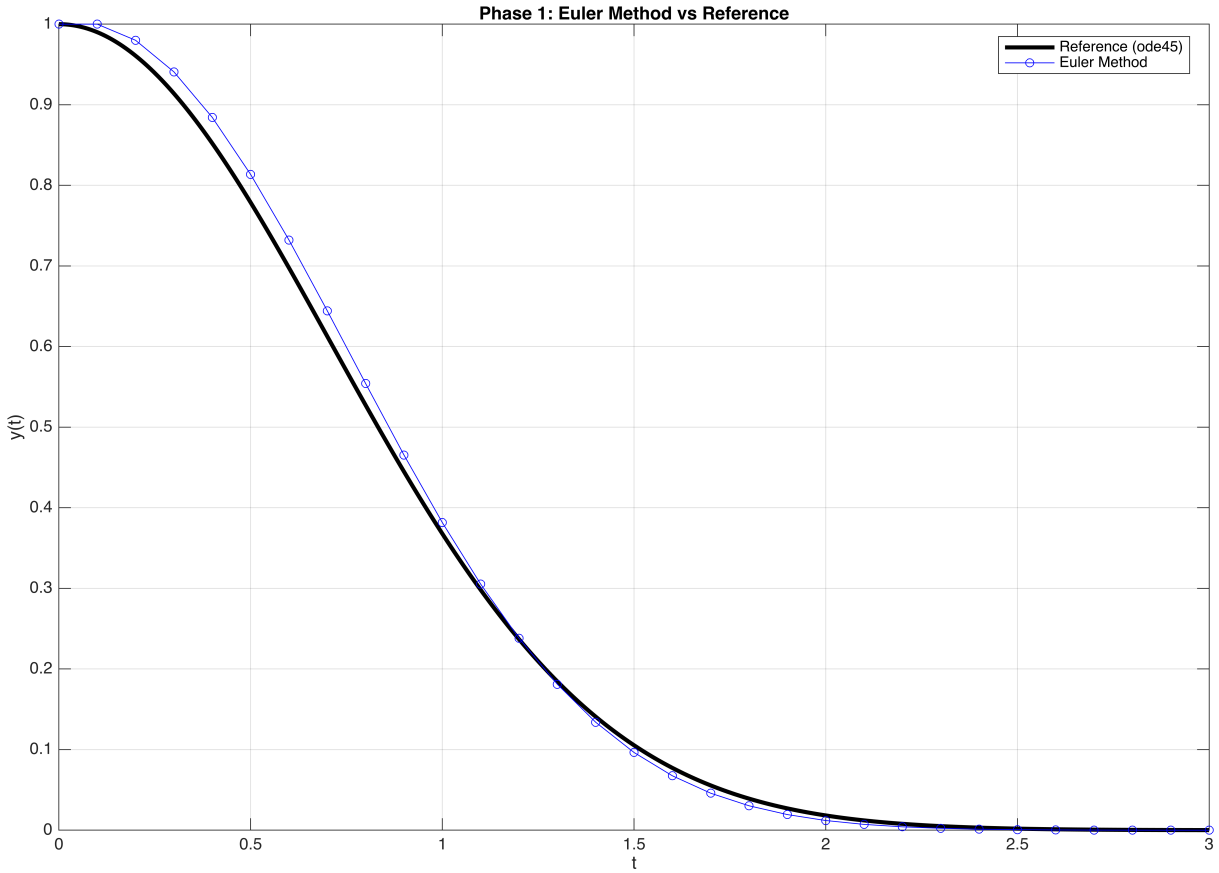
The stability of Euler's method depends on the nature of the equation being solved and the step size h. For the linear equation $y' = \lambda y$, where $\lambda \in \mathbb{C}$ and $\mathrm{Re}(\lambda) < 0$, the method is stable if and only if: $|1 + h\lambda| \le 1$.

This condition defines the stability region of the method in the complex plane — a unit circle centered at $(-1, 0)$. For equations with large negative eigenvalues (stiff systems), Euler's method requires extremely small step sizes to maintain stability, making it inefficient in such cases.

```
%% PHASE 1: EULER METHOD


%% 3. Euler Method Implementation
y_euler = zeros(1, N+1);
y_euler(1) = y0;

for n = 1:N
    y_euler(n+1) = y_euler(n) + h * f(t_grid(n), y_euler(n));
end

%% Error vs Reference
y_exact = y_reference(t_grid);
err_euler = norm(y_euler - y_exact);
fprintf('Global Error (Euler vs Reference): %.2e\n', err_euler);
```

```
Global Error (Euler vs Reference): 8.71e-02
```

```
%% Plot: Reference + Euler
figure('Position', [100, 100, 900, 600]);
t_plot = linspace(t0, t_end, 1000);
y_plot = y_reference(t_plot);
plot(t_plot, y_plot, 'k-', 'LineWidth', 2.5, 'DisplayName', 'Reference
(ode45)');
hold on;
plot(t_grid, y_euler, 'bo-', 'MarkerSize', 5, 'DisplayName', 'Euler
Method');
xlabel('t');
ylabel('y(t)');
title('Phase 1: Euler Method vs Reference');
legend('Location', 'best');
grid on;
hold off;
```




```
%% Convergence Analysis for Euler ONLY
h_values = [0.1, 0.05, 0.025, 0.0125, 0.00625];
errors_euler = zeros(size(h_values));

for idx = 1:length(h_values)
    h_temp = h_values(idx);
    N_temp = (t_end - t0) / h_temp;
    t_temp = linspace(t0, t_end, N_temp+1);

    % Euler method - вычисляем только его
    y_temp_euler = zeros(1, N_temp+1);
    y_temp_euler(1) = y0;
    for n = 1:N_temp
        y_temp_euler(n+1) = y_temp_euler(n) + h_temp * f(t_temp(n),
y_temp_euler(n));
    end

    y_exact_temp = y_reference(t_temp);
    errors_euler(idx) = norm(y_temp_euler - y_exact_temp);
end
```

```
%% Sort for convergence plot (local to this phase)
[sorted_h_phase1, idx_phase1] = sort(h_values);
sorted_errors_euler_phase1 = errors_euler(idx_phase1);

%% Print convergence table for Euler
fprintf('\n--- EULER CONVERGENCE: Step Size vs Error ---\n');
```

```
--- EULER CONVERGENCE: Step Size vs Error ---
```

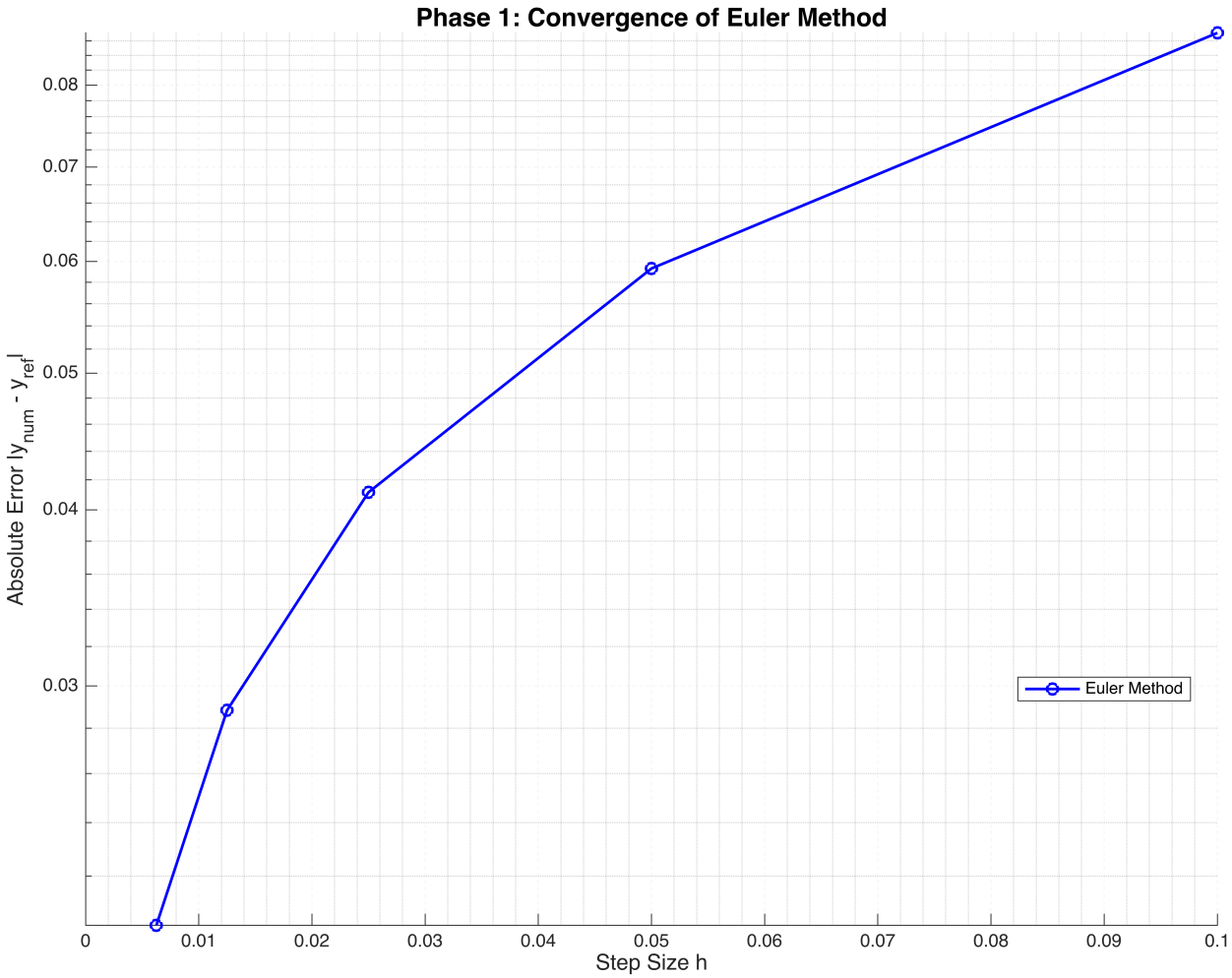
```
fprintf('%8s %12s\n', 'h', 'Euler Error');
```

```
       h  Euler Error
```

```
for i = 1:length(h_values)
    fprintf('%8.5f %12.2e\n', sorted_h_phase1(i),
sorted_errors_euler_phase1(i));
end
```

```
 0.00625     2.03e-02
 0.01250     2.88e-02
 0.02500     4.12e-02
 0.05000     5.93e-02
 0.10000     8.71e-02
```

```

%% Convergence Plot for Euler
figure('Position', [100, 100, 800, 600]);
semilogy(sorted_h_phase1, sorted_errors_euler_phase1, 'b-o', 'LineWidth',
1.5, 'MarkerSize', 6, 'DisplayName', 'Euler Method');
xlabel('Step Size h', 'FontSize', 12);
ylabel('Absolute Error |y_{num} - y_{ref}|', 'FontSize', 12);
title('Phase 1: Convergence of Euler Method', 'FontSize', 14, 'FontWeight',
'bold');
grid on; grid minor;
set(gca, 'GridLineStyle', '--', 'GridColor', [0.7, 0.7, 0.7]);
box off;
legend('Location', 'best');
```




```
%% Estimate Euler convergence order
h_sub = sorted_h_phase1(3:end);
err_sub = sorted_errors_euler_phase1(3:end);
if ~isempty(h_sub) && ~isempty(err_sub)
    p_euler = log(err_sub(1)/err_sub(end)) / log(h_sub(1)/h_sub(end));
    fprintf('Estimated convergence order (Euler): %.3f (expected:
~1.0)\n\n', p_euler);
else
    fprintf('Not enough points to estimate Euler convergence order.\n\n');
end
```

```
Estimated convergence order (Euler): 0.541 (expected: ~1.0)
```

Explicit Trapezoid Method (Improved Euler / Heun's Method): Theoretical Description

The Explicit Trapezoid Method, also known as the Improved Euler Method or Heun's Method, is a second-order explicit Runge-Kutta scheme for solving initial value problems of the form:

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0.$$

This method improves upon the basic Euler method by incorporating an estimate of the slope at the end of the interval, thereby achieving higher accuracy without significantly increasing computational cost.

The algorithm proceeds in two stages. First, an explicit Euler predictor is used to estimate the solution at the next time step:

$$\widetilde{y}_{n+1} = y_n + h \cdot f(t_n, y_n).$$

This predicted value is then used to evaluate the derivative at the end of the interval, $ f(t_{n+1}, \tilde{y}_{n+1}) $. The final update is computed as the average of the slopes at the beginning and the end of the step:

$$y_{n+1} = y_n + \frac{h}{2}\left[f(t_n, y_n) + f(t_{n+1}, \widetilde{y}_{n+1})\right].$$

This formulation can be interpreted as applying the trapezoidal rule to the integral form of the ODE:

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(t, y(t))\, dt,$$

where the integrand is approximated by a linear interpolant between $(t_n, f(t_n, y_n))$ and $(t_{n+1}, f(t_{n+1}, \widetilde{y}_{n+1}))$. Although the classical trapezoidal rule is implicit, this explicit variant uses a predicted endpoint to avoid solving a nonlinear equation, hence the name "explicit trapezoid".

The method is second-order accurate, meaning that the local truncation error is $O(h^3)$ and the global error is $O(h^2)$. This represents a significant improvement over the first-order Euler method, typically reducing the error by a factor of approximately four when the step size is halved.

The Explicit Trapezoid Method requires two evaluations of the function f per step, compared to one for Euler, but delivers substantially higher accuracy. It is particularly useful as a simple yet effective stepping scheme for non-stiff problems and serves as the predictor in predictor-corrector pairs.

Its stability region is larger than that of the Euler method but smaller than that of implicit schemes. The method is not A-stable and is therefore unsuitable for stiff equations, but it is robust and efficient for smooth, non-stiff systems.

In summary, the Explicit Trapezoid Method provides an excellent balance between simplicity, computational cost, and accuracy, making it a natural intermediate step between the Euler method and higher-order Runge-Kutta schemes such as RK4.

```
%% EXPLICIT TRAPEZOID (PREDICTOR) — IMPROVED EULER

%% Explicit Trapezoid (Improved Euler / Heun's Method)
y_trap_expl = zeros(1, N+1);
y_trap_expl(1) = y0;

for n = 1:N
    t_n = t_grid(n);
    y_n = y_trap_expl(n);
    k1 = f(t_n, y_n);
    k2 = f(t_n + h, y_n + h * k1);  % явный прогноз
    y_trap_expl(n+1) = y_n + (h/2) * (k1 + k2);
end

%% Error vs Reference
err_trap_expl = norm(y_trap_expl - y_exact);
fprintf('\n=== PHASE 2: EXPLICIT TRAPEZOID (PREDICTOR) ===\n');
```

```
=== PHASE 2: EXPLICIT TRAPEZOID (PREDICTOR) ===
```

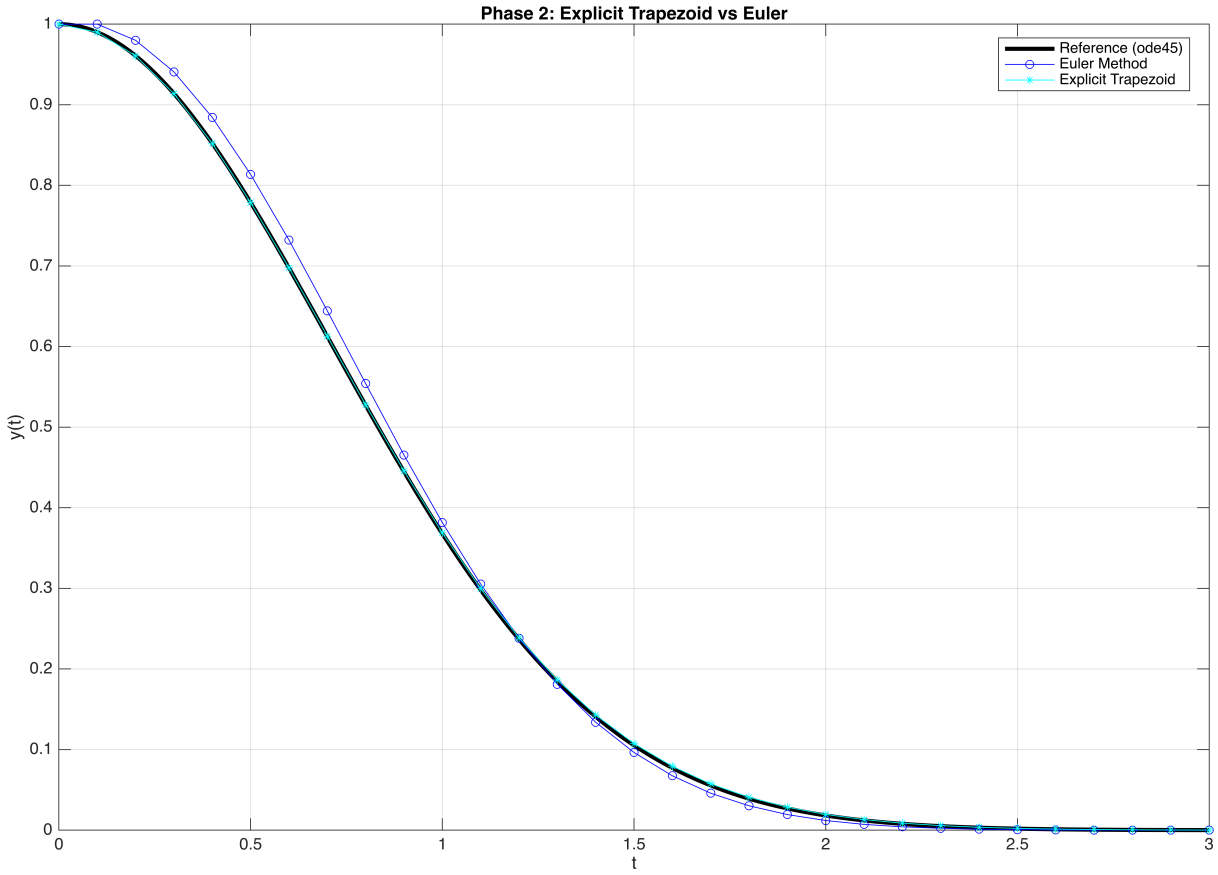
```
fprintf('Global Error (Explicit Trapezoid vs Reference): %.2e\n',
err_trap_expl);
```

```
Global Error (Explicit Trapezoid vs Reference): 6.05e-03
```

```
fprintf('Improvement over Euler: %.1fx smaller error\n', err_euler/
err_trap_expl);
```

```
Improvement over Euler: 14.4x smaller error
```

```
%% Plot: Reference + Euler + Explicit Trapezoid
figure('Position', [100, 100, 900, 600]);
plot(t_plot, y_plot, 'k-', 'LineWidth', 2.5, 'DisplayName', 'Reference
(ode45)');
hold on;
plot(t_grid, y_euler,     'bo-', 'MarkerSize', 5, 'DisplayName', 'Euler
Method');
plot(t_grid, y_trap_expl, 'c*-', 'MarkerSize', 5, 'DisplayName', 'Explicit
Trapezoid');
xlabel('t');
ylabel('y(t)');
title('Phase 2: Explicit Trapezoid vs Euler');
legend('Location', 'best');
grid on;
hold off;
```




```
%% Convergence Analysis for Explicit Trapezoid
errors_trap_expl = zeros(size(h_values));

for idx = 1:length(h_values)
    h_temp = h_values(idx);
    N_temp = (t_end - t0) / h_temp;
    t_temp = linspace(t0, t_end, N_temp+1);

    y_temp_trap_expl = zeros(1, N_temp+1);
    y_temp_trap_expl(1) = y0;
    for n = 1:N_temp
        k1 = f(t_temp(n), y_temp_trap_expl(n));
        k2 = f(t_temp(n) + h_temp, y_temp_trap_expl(n) + h_temp * k1);
        y_temp_trap_expl(n+1) = y_temp_trap_expl(n) + (h_temp/2) * (k1 +
k2);
    end

    y_exact_temp = y_reference(t_temp);
    errors_trap_expl(idx) = norm(y_temp_trap_expl - y_exact_temp);
end
```

```
%% Sort for convergence plot
[sorted_h_phase2, idx_phase2] = sort(h_values);
sorted_errors_euler_phase2 = errors_euler(idx_phase2);
sorted_errors_trap_expl_phase2 = errors_trap_expl(idx_phase2);

%% Print comparison table
fprintf('\n--- COMPARISON: Step Size vs Error (Euler vs Explicit Trapezoid)
---\n');
```

```
--- COMPARISON: Step Size vs Error (Euler vs Explicit Trapezoid) ---
```

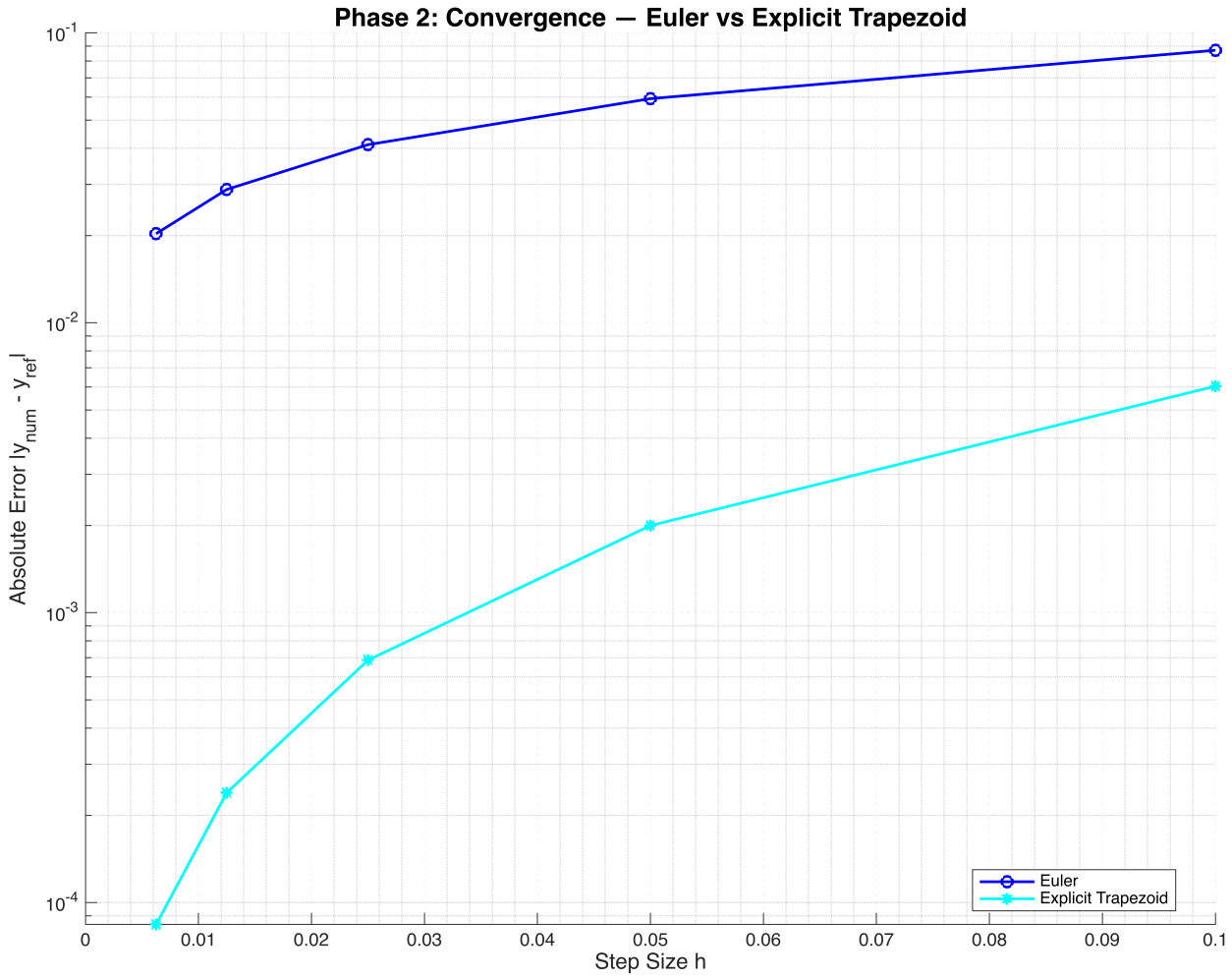
```
fprintf('%8s %12s %12s\n', 'h', 'Euler Error', 'Expl Trap Error');
```

```
       h  Euler Error Expl Trap Error
```

```
for i = 1:length(h_values)
    fprintf('%8.5f %12.2e %12.2e\n', sorted_h_phase2(i),
sorted_errors_euler_phase2(i), sorted_errors_trap_expl_phase2(i));
end
```

```
 0.00625     2.03e-02     8.40e-05
 0.01250     2.88e-02     2.39e-04
 0.02500     4.12e-02     6.86e-04
 0.05000     5.93e-02     2.00e-03
 0.10000     8.71e-02     6.05e-03
```

```

%% Convergence Plot
figure('Position', [100, 100, 800, 600]);
semilogy(sorted_h_phase2, sorted_errors_euler_phase2,      'b-o',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Euler');
hold on;
semilogy(sorted_h_phase2, sorted_errors_trap_expl_phase2,  'c*-',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Explicit Trapezoid');
xlabel('Step Size h', 'FontSize', 12);
ylabel('Absolute Error |y_{num} - y_{ref}|', 'FontSize', 12);
title('Phase 2: Convergence - Euler vs Explicit Trapezoid', 'FontSize', 14,
'FontWeight', 'bold');
grid on; grid minor;
set(gca, 'GridLineStyle', '--', 'GridColor', [0.7, 0.7, 0.7]);
box off;
legend('Location', 'best');
```




```
%% Estimate convergence order
h_sub_trap = sorted_h_phase2(3:end);
err_sub_trap = sorted_errors_trap_expl_phase2(3:end);
if ~isempty(h_sub_trap) && ~isempty(err_sub_trap)
    p_trap_expl = log(err_sub_trap(1)/err_sub_trap(end)) /
log(h_sub_trap(1)/h_sub_trap(end));
    fprintf('Estimated convergence order (Explicit Trapezoid): %.3f
(expected: ~2.0)\n', p_trap_expl);
else
    fprintf('Not enough points to estimate Explicit Trapezoid convergence
order.\n');
end
```

```
Estimated convergence order (Explicit Trapezoid): 1.570 (expected: ~2.0)
```

2. Runge-Kutta Method of Fourth Order (RK4): Formal Description

2.1. Problem Statement

Consider the initial value problem for a first-order ordinary differential equation:

$$\frac{dy}{dt} = f(t, y), \quad t \in [t_0, T], \; y(t_0) = y_0,$$

where t denotes the independent variable, $y(t)$ is the unknown scalar or vector-valued function, and $f(t, y)$ is a given continuous function satisfying the Lipschitz condition with respect to y in a domain containing the initial point (t_0, y_0). This condition ensures the existence and uniqueness of the solution on the interval $[t_0, T]$.

The objective is to construct a numerical approximation to the solution $y(t)$ at discrete points within the interval.

2.2. Derivation and Formula

The fourth-order Runge-Kutta method belongs to a family of one-step explicit methods derived from matching the Taylor expansion of the solution up to the fourth-order term. It achieves high accuracy without requiring explicit computation of higher-order derivatives by evaluating the function f at several intermediate points within each step.

Given a uniform discretization of the interval $[t_0, T]$ with step size h, and denoting $t_n = t_0 + nh$, the method computes the approximation $y_{n+1} \approx y(t_{n+1})$ from the current value $y_n \approx y(t_n)$ using the following weighted average of four slope estimates:

$$k_1 = f(t_n, y_n),$$

$$k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1),$$

$$k_3 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right),$$

$$k_4 = f(t_n + h, y_n + hk_3),$$

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4).$$

This formulation ensures that the local truncation error is of order $O(h^5)$, and the global error is of order $O(h^4)$, making RK4 a fourth-order method.

2.3. Algorithmic Implementation

In computational implementation, the Euler method proceeds iteratively:

1. Initialize t_0 and y_0 as given by the initial condition.

2. For each n from 0 to $N - 1$:

• Evaluate the right-hand side: $\mathbf{k}_1 = \mathbf{f}(t_n, y_n)$.

• Compute the next state: $y_{n+1} = y_n + h \cdot \mathbf{k}_1$.

• Advance time: $t_{n+1} = t_n + h$.

3. Return the arrays $\{t_n\}$ and $\{y_n\}$ as the numerical solution.

The method requires exactly N evaluations of the function $\mathbf{f}$ and N vector additions, making it computationally inexpensive per step. However, due to its first-order global convergence, achieving high accuracy necessitates small step sizes, which may lead to increased cumulative computational cost and potential accumulation of rounding errors.

2.4. Accuracy and Convergence

The global convergence order of the RK4 method is four, meaning that halving the step size reduces the global error by approximately a factor of 16. This high order of accuracy makes RK4 significantly more efficient than lower-order methods such as Euler's method for achieving a given level of precision.

The local truncation error, which measures the error introduced in a single step assuming exact initial data, is proportional to h^5. This high local accuracy, combined with the method's stability for non-stiff problems, has established RK4 as a standard tool in scientific computing for non-stiff initial value problems.

2.5. Stability and Applicability

The region of absolute stability for RK4 includes a significant portion of the left half of the complex plane, making it suitable for mildly stiff problems, though not recommended for strongly stiff systems. For non-stiff equations with smooth solutions, RK4 offers an excellent balance of accuracy, simplicity, and computational efficiency.

Its primary limitation is the fixed step size — for problems requiring adaptive step control, more sophisticated embedded methods such as Runge-Kutta-Fehlberg (RKF45) or Dormand-Prince (used in MATLAB's `ode45`) are preferred. Nevertheless, RK4 remains a cornerstone of numerical analysis education and a reliable choice for fixed-step integration.

```
%%  RK4 METHOD
%% 4. RK4 Method Implementation
y_rk4 = zeros(1, N+1);
y_rk4(1) = y0;

for n = 1:N
    k1 = f(t_grid(n), y_rk4(n));
    k2 = f(t_grid(n) + h/2, y_rk4(n) + h/2 * k1);
    k3 = f(t_grid(n) + h/2, y_rk4(n) + h/2 * k2);
    k4 = f(t_grid(n) + h, y_rk4(n) + h * k3);
    y_rk4(n+1) = y_rk4(n) + h/6 * (k1 + 2*k2 + 2*k3 + k4);
end
```

```
%% Error vs Reference
err_rk4 = norm(y_rk4 - y_exact);
fprintf('\n=== PHASE 3: RK4 METHOD ===\n');
```

```
=== PHASE 3: RK4 METHOD ===
```

```
fprintf('Global Error (RK4 vs Reference): %.2e\n', err_rk4);
```

```
Global Error (RK4 vs Reference): 2.22e-05
```

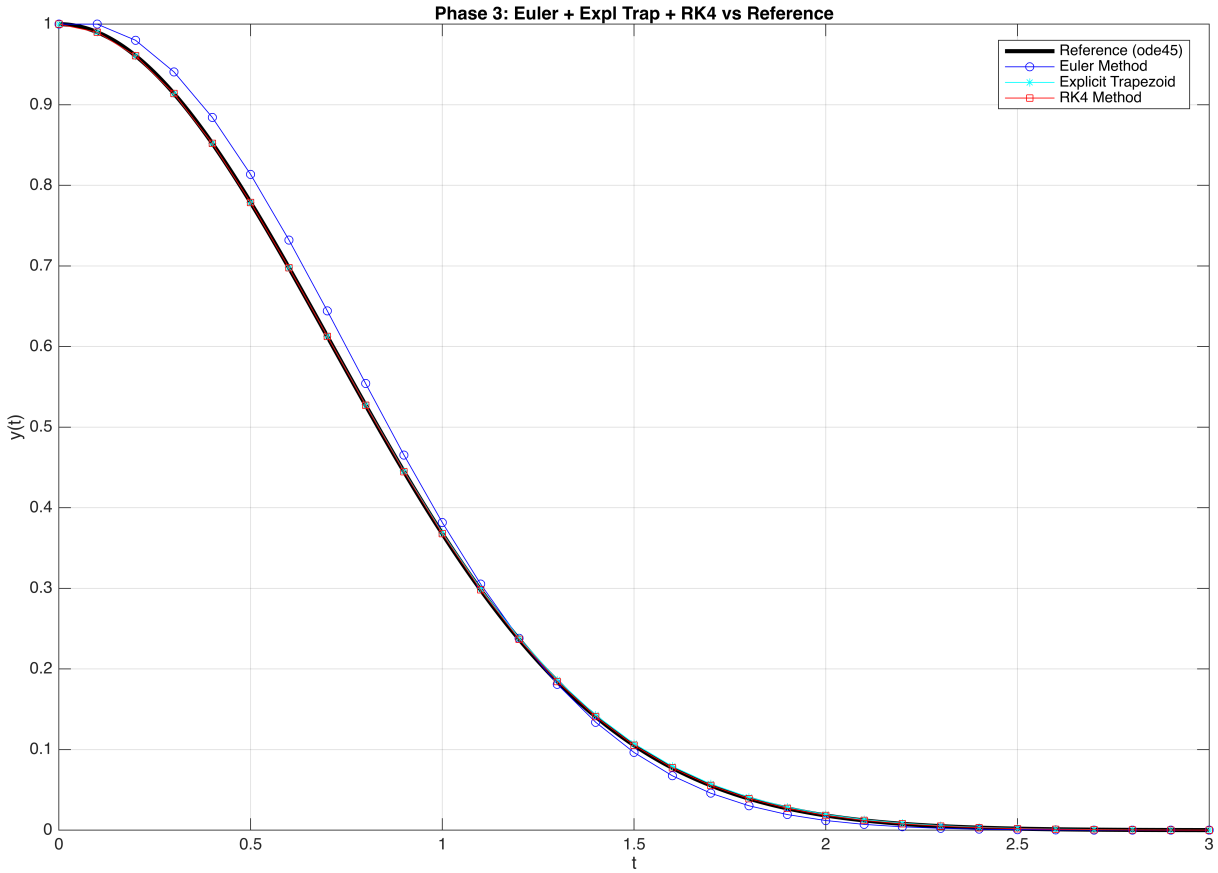
```
fprintf('Improvement over Euler: %.1fx smaller error\n', err_euler/err_rk4);
```

```
Improvement over Euler: 3930.1x smaller error
```

```
fprintf('Improvement over Explicit Trapezoid: %.1fx smaller error\n',
err_trap_expl/err_rk4);
```

```
Improvement over Explicit Trapezoid: 272.7x smaller error
```

```
%% Plot: Reference + Euler + Expl Trap + RK4
figure('Position', [100, 100, 900, 600]);
plot(t_plot, y_plot, 'k-', 'LineWidth', 2.5, 'DisplayName', 'Reference
(ode45)');
hold on;
plot(t_grid, y_euler,     'bo-', 'MarkerSize', 5, 'DisplayName', 'Euler
Method');
plot(t_grid, y_trap_expl, 'c*-', 'MarkerSize', 5, 'DisplayName', 'Explicit
Trapezoid');
plot(t_grid, y_rk4,       'rs-', 'MarkerSize', 5, 'DisplayName', 'RK4
Method');
xlabel('t');
ylabel('y(t)');
title('Phase 3: Euler + Expl Trap + RK4 vs Reference');
legend('Location', 'best');
grid on;
hold off;
```




```
%% Convergence Analysis for RK4
errors_rk4 = zeros(size(h_values));

for idx = 1:length(h_values)
    h_temp = h_values(idx);
    N_temp = (t_end - t0) / h_temp;
    t_temp = linspace(t0, t_end, N_temp+1);

    y_temp_rk4 = zeros(1, N_temp+1);
    y_temp_rk4(1) = y0;
    for n = 1:N_temp
        k1 = f(t_temp(n), y_temp_rk4(n));
        k2 = f(t_temp(n) + h_temp/2, y_temp_rk4(n) + h_temp/2 * k1);
        k3 = f(t_temp(n) + h_temp/2, y_temp_rk4(n) + h_temp/2 * k2);
        k4 = f(t_temp(n) + h_temp, y_temp_rk4(n) + h_temp * k3);
        y_temp_rk4(n+1) = y_temp_rk4(n) + h_temp/6 * (k1 + 2*k2 + 2*k3 +
k4);
    end

    y_exact_temp = y_reference(t_temp);
    errors_rk4(idx) = norm(y_temp_rk4 - y_exact_temp);
```

```
end

%% Sort for convergence plot
[sorted_h_phase3, idx_phase3] = sort(h_values);
sorted_errors_euler_phase3 = errors_euler(idx_phase3);
sorted_errors_trap_expl_phase3 = errors_trap_expl(idx_phase3);
sorted_errors_rk4_phase3   = errors_rk4(idx_phase3);

%% Print comparison table
fprintf('\n--- COMPARISON: Step Size vs Error ---\n');
```

```
--- COMPARISON: Step Size vs Error ---
```

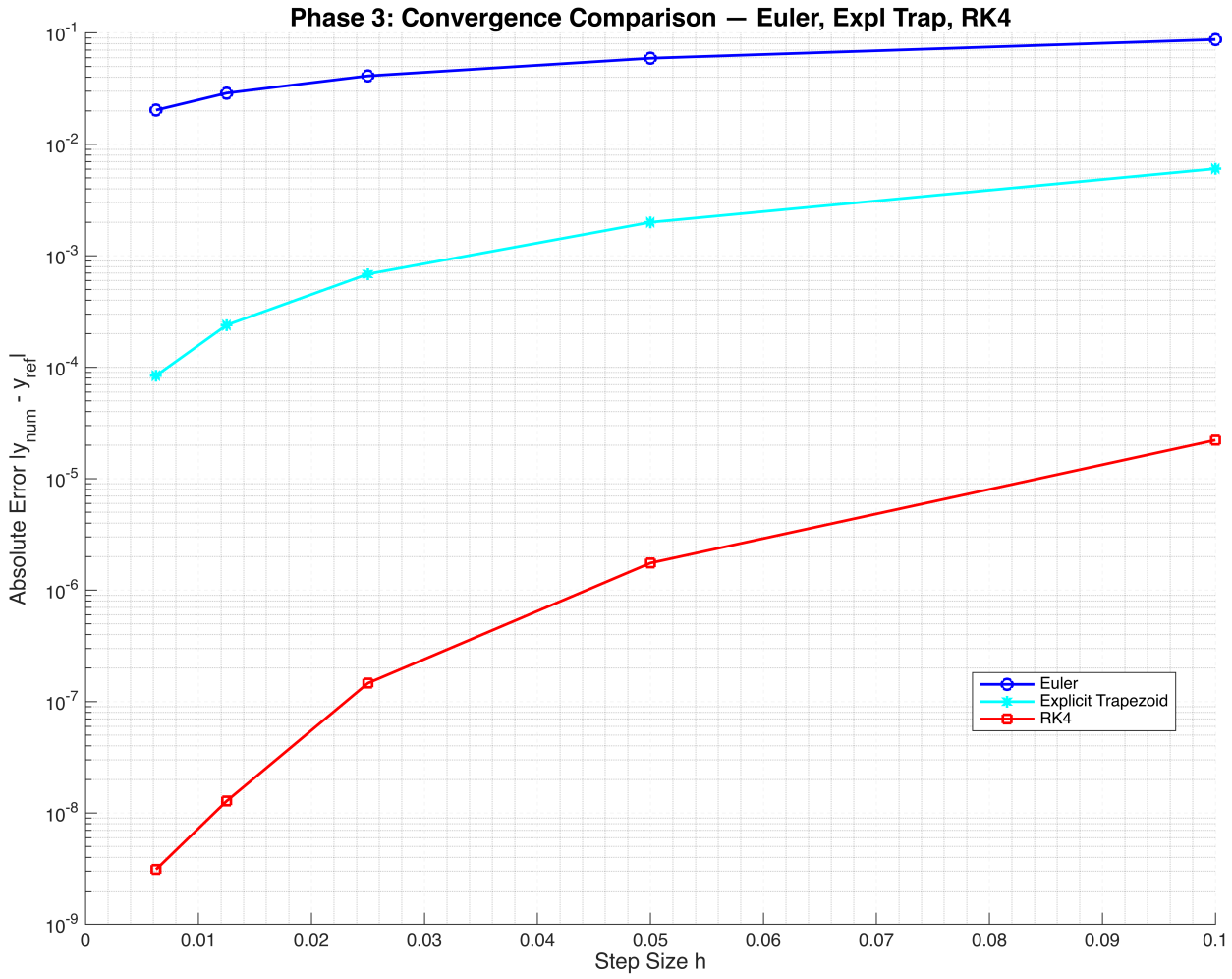
```
fprintf('%8s %12s %12s %12s\n', 'h', 'Euler', 'Expl Trap', 'RK4');
```

```
       h        Euler    Expl Trap          RK4
```

```
for i = 1:length(h_values)
    fprintf('%8.5f %12.2e %12.2e %12.2e\n', sorted_h_phase3(i),
sorted_errors_euler_phase3(i), sorted_errors_trap_expl_phase3(i),
sorted_errors_rk4_phase3(i));
end
```

```
 0.00625     2.03e-02     8.40e-05     3.11e-09
 0.01250     2.88e-02     2.39e-04     1.28e-08
 0.02500     4.12e-02     6.86e-04     1.46e-07
 0.05000     5.93e-02     2.00e-03     1.75e-06
 0.10000     8.71e-02     6.05e-03     2.22e-05
```

```

%% Convergence Plot
figure('Position', [100, 100, 800, 600]);
semilogy(sorted_h_phase3, sorted_errors_euler_phase3,      'b-o',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Euler');
hold on;
semilogy(sorted_h_phase3, sorted_errors_trap_expl_phase3,  'c*-',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Explicit Trapezoid');
semilogy(sorted_h_phase3, sorted_errors_rk4_phase3,        'r-s',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'RK4');
xlabel('Step Size h', 'FontSize', 12);
ylabel('Absolute Error |y_{num} - y_{ref}|', 'FontSize', 12);
title('Phase 3: Convergence Comparison — Euler, Expl Trap, RK4',
'FontSize', 14, 'FontWeight', 'bold');
grid on; grid minor;
set(gca, 'GridLineStyle', '--', 'GridColor', [0.7, 0.7, 0.7]);
box off;
legend('Location', 'best');
```




```
%% Estimate RK4 convergence order
h_sub_rk4 = sorted_h_phase3(3:end);
err_sub_rk4 = sorted_errors_rk4_phase3(3:end);
if ~isempty(h_sub_rk4) && ~isempty(err_sub_rk4)
    p_rk4 = log(err_sub_rk4(1)/err_sub_rk4(end)) / log(h_sub_rk4(1)/
h_sub_rk4(end));
    fprintf('Estimated convergence order (RK4): %.3f (expected: ~4.0)\n',
p_rk4);
else
    fprintf('Not enough points to estimate RK4 convergence order.\n');
end
```

```
Estimated convergence order (RK4): 3.621 (expected: ~4.0)
```

3. Backward Euler Method: Formal Description

3.1. Derivation and Formula

The Backward Euler method is an implicit, single-step numerical scheme derived from the backward difference approximation of the derivative. Unlike the explicit Euler method, which evaluates the right-hand side at the beginning of the interval, the Backward Euler method evaluates it at the end of the interval, leading to an implicit equation for the next solution value.

Given a uniform discretization of the interval $[t_0, T]$ with step size h, and denoting $t_n = t_0 + nh$, the method computes the approximation $y_{n+1} \approx y(t_{n+1})$ by solving the nonlinear equation:

$y_{n+1} = y_n + h \cdot f(t_{n+1}, y_{n+1})$.

This equation is implicit because y_{n+1} appears on both sides. The method is unconditionally stable for linear problems and suitable for stiff systems, where explicit methods such as Euler or RK4 may require prohibitively small step sizes for stability.

3.2. Algorithmic Implementation

To implement the Backward Euler method for the initial value problem $dy/dt = f(t, y), y(t0) = y0$, perform the following steps:

1. Define the time grid:

• Set initial time t_0, final time T, and step size h.

• Compute number of steps: $N = (T - t0)/h$.

• Generate time nodes: $t_n = t0 + n * h$, for $n = 0, 1, ..., N$.

2. Initialize the solution array:

• Allocate vector y of size N+1.

• Set $y(1) = y0$ (initial condition).

3. For each time step n = 1 to N:

• Define the nonlinear equation for y_{n+1}:

G(z) = z - h * f(t_{n+1}, z) - y_n = 0

• Solve G(z) = 0 for z using a root-finding method (e.g., Newton-Raphson or fzero).

— If using Newton-Raphson:

* Choose initial guess z0 (e.g., z0 = y_n).

* Iterate: $z_{k+1} = z_k - G(z_k)/G'(z_k)$,

where $G'(z) = 1 - h * df/dy(t_{n+1}, z_k)$

* Stop when $|z_{k+1} - z_k| < tolerance$.

— If using fzero:

* Define anonymous function: $G = @(z)z - h * f(t(n+1), z) - y(n)$;

* Call: $z = fzero(G, y(n))$;

• Set $y(n+1) = z$ (the root found).

4. Return the arrays t and y as the numerical solution.

Note: The method requires solving a nonlinear equation at each step. The choice of solver and initial guess may affect convergence and efficiency.

3.3. Accuracy and Convergence

The global convergence order of the Backward Euler method is one, meaning that halving the step size reduces the global error by approximately a factor of two. The local truncation error is of order $ O(h^2) $, consistent with the first-order global convergence. While the method is less accurate per step than higher-order schemes such as RK4, its primary advantage lies in its stability rather than its accuracy. For non-stiff problems, explicit methods are generally preferred due to their lower computational cost. However, for stiff systems, where stability constraints dominate accuracy considerations, the Backward Euler method is often the method of choice due to its unconditional stability.

3.4. Stability and Applicability

The region of absolute stability for the Backward Euler method includes the entire left half of the complex plane, making it unconditionally stable for linear problems with negative real eigenvalues. This property renders the method particularly suitable for stiff differential equations, where explicit methods suffer from severe step size restrictions. The method is also A-stable, meaning that it remains stable for all step sizes when applied to the scalar test equation $ y' = \lambda y $ with $ \text{Re}(\lambda) < 0 $. Its primary limitation is the need to solve a nonlinear equation at each step, which may require additional computational effort and careful initialization to ensure convergence of the nonlinear solver.

```
%%  BACKWARD EULER
%% 5. Backward Euler Method Implementation
y_beuler = zeros(1, N+1);
y_beuler(1) = y0;

for n = 1:N
    t_next = t_grid(n+1);
    G = @(z) z - h * f(t_next, z) - y_beuler(n);
    y_beuler(n+1) = fzero(G, y_beuler(n));
end
```

```
%% Error vs Reference
err_beuler = norm(y_beuler - y_exact);
fprintf('\n=== PHASE 4: BACKWARD EULER METHOD ===\n');
```

```
=== PHASE 4: BACKWARD EULER METHOD ===
```

```
fprintf('Global Error (Backward Euler vs Reference): %.2e\n', err_beuler);
```

```
Global Error (Backward Euler vs Reference): 7.51e-02
```

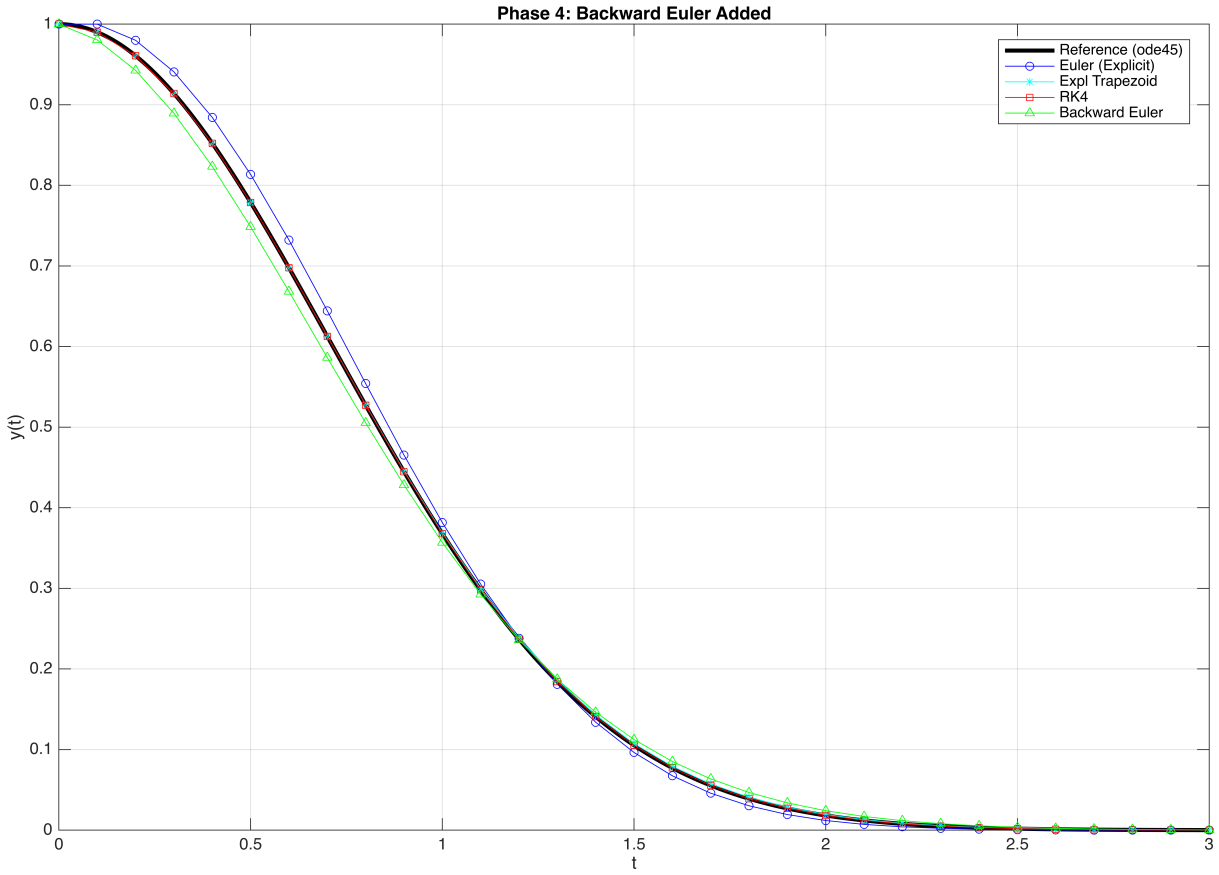
```
fprintf('Improvement over Euler: %.1fx smaller error\n', err_euler/
err_beuler);
```

```
Improvement over Euler: 1.2x smaller error
```

```
if exist('p_rk4', 'var')
    if err_rk4 < err_beuler
        comp_str = 'more';
    else
        comp_str = 'less';
    end
    fprintf('Comparison with RK4: %.1fx %s accurate\n', ...
        max(err_rk4,err_beuler)/min(err_rk4,err_beuler), comp_str);
end
```

```
Comparison with RK4: 3385.2x more accurate
```

```
%% Plot: All explicit + Backward Euler
figure('Position', [100, 100, 900, 600]);
plot(t_plot, y_plot, 'k-', 'LineWidth', 2.5, 'DisplayName', 'Reference
(ode45)');
hold on;
plot(t_grid, y_euler,     'bo-', 'MarkerSize', 5, 'DisplayName', 'Euler
(Explicit)');
plot(t_grid, y_trap_expl, 'c*-', 'MarkerSize', 5, 'DisplayName', 'Expl
Trapezoid');
plot(t_grid, y_rk4,       'rs-', 'MarkerSize', 5, 'DisplayName', 'RK4');
plot(t_grid, y_beuler,    'g^-', 'MarkerSize', 5, 'DisplayName', 'Backward
Euler');
xlabel('t');
ylabel('y(t)');
title('Phase 4: Backward Euler Added');
legend('Location', 'best');
grid on;
hold off;
```




```
%% Convergence Analysis for Backward Euler
errors_beuler = zeros(size(h_values));

for idx = 1:length(h_values)
    h_temp = h_values(idx);
    N_temp = (t_end - t0) / h_temp;
    t_temp = linspace(t0, t_end, N_temp+1);

    y_temp_beuler = zeros(1, N_temp+1);
    y_temp_beuler(1) = y0;
    for n = 1:N_temp
        t_next = t_temp(n+1);
        G = @(z) z - h_temp * f(t_next, z) - y_temp_beuler(n);
        y_temp_beuler(n+1) = fzero(G, y_temp_beuler(n));
    end

    y_exact_temp = y_reference(t_temp);
    errors_beuler(idx) = norm(y_temp_beuler - y_exact_temp);
end

%% Sort for convergence plot
```

```
[sorted_h_phase4, idx_phase4] = sort(h_values);
sorted_errors_euler_phase4 = errors_euler(idx_phase4);
sorted_errors_trap_expl_phase4 = errors_trap_expl(idx_phase4);
sorted_errors_rk4_phase4   = errors_rk4(idx_phase4);
sorted_errors_beuler_phase4 = errors_beuler(idx_phase4);

%% Print full comparison table
fprintf('\n--- FULL COMPARISON: Step Size vs Error ---\n');
```

```
--- FULL COMPARISON: Step Size vs Error ---
```

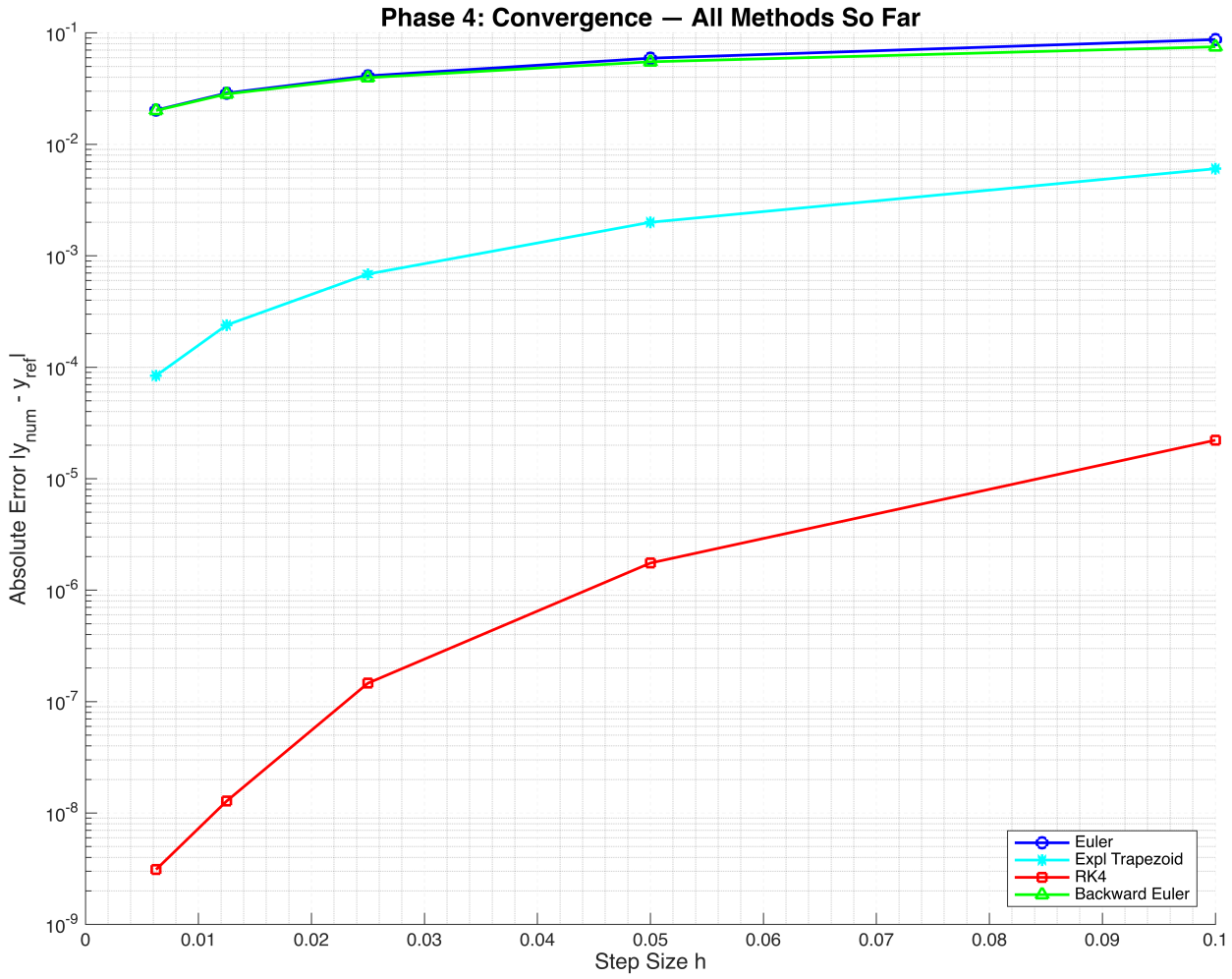
```
fprintf('%8s %12s %12s %12s %12s\n', 'h', 'Euler', 'Expl Trap', 'RK4',
'Backw Eul');
```

```
       h        Euler    Expl Trap          RK4    Backw Eul
```

```
for i = 1:length(h_values)
    fprintf('%8.5f %12.2e %12.2e %12.2e %12.2e\n', sorted_h_phase4(i),
sorted_errors_euler_phase4(i), sorted_errors_trap_expl_phase4(i),
sorted_errors_rk4_phase4(i), sorted_errors_beuler_phase4(i));
end
```

```
 0.00625     2.03e-02     8.40e-05     3.11e-09     2.01e-02
 0.01250     2.88e-02     2.39e-04     1.28e-08     2.83e-02
 0.02500     4.12e-02     6.86e-04     1.46e-07     3.97e-02
 0.05000     5.93e-02     2.00e-03     1.75e-06     5.51e-02
 0.10000     8.71e-02     6.05e-03     2.22e-05     7.51e-02
```

```
%% Convergence Plot
figure('Position', [100, 100, 800, 600]);
semilogy(sorted_h_phase4, sorted_errors_euler_phase4,      'b-o',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Euler');
hold on;
semilogy(sorted_h_phase4, sorted_errors_trap_expl_phase4,  'c*-',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Expl Trapezoid');
semilogy(sorted_h_phase4, sorted_errors_rk4_phase4,        'r-s',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'RK4');
semilogy(sorted_h_phase4, sorted_errors_beuler_phase4,     'g^-',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Backward Euler');
xlabel('Step Size h', 'FontSize', 12);
ylabel('Absolute Error |y_{num} - y_{ref}|', 'FontSize', 12);
title('Phase 4: Convergence — All Methods So Far', 'FontSize', 14,
'FontWeight', 'bold');
grid on; grid minor;
set(gca, 'GridLineStyle', '--', 'GridColor', [0.7, 0.7, 0.7]);
box off;
legend('Location', 'best');
```




```
%% Estimate Backward Euler convergence order
h_sub_beuler = sorted_h_phase4(3:end);
err_sub_beuler = sorted_errors_beuler_phase4(3:end);
if ~isempty(h_sub_beuler) && ~isempty(err_sub_beuler)
    p_beuler = log(err_sub_beuler(1)/err_sub_beuler(end)) /
log(h_sub_beuler(1)/h_sub_beuler(end));
    fprintf('Estimated convergence order (Backward Euler): %.3f (expected:
~1.0)\n', p_beuler);
else
    fprintf('Not enough points to estimate Backward Euler convergence
order.\n');
end
```

```
Estimated convergence order (Backward Euler): 0.460 (expected: ~1.0)
```

Implicit Trapezoid Method: Theoretical Description

The Implicit Trapezoid Method is a second-order implicit Runge–Kutta scheme for solving initial value problems of the form:

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0.$$

This method is derived by applying the classical trapezoidal rule to the integral formulation of the ODE:

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(t, y(t))\, dt.$$

The integrand is approximated by the average of its values at the endpoints of the interval, leading to the update formula:

$$y_{n+1} = y_n + \frac{h}{2}[f(t_n, y_n) + f(t_{n+1}, y_{n+1})].$$

Unlike the explicit trapezoid (Heun's) method, which uses a predicted value for y_{n+1} in the second term, the implicit version uses the unknown y_{n+1} itself. This results in an implicit equation that must be solved at each step, typically via a **root-finding algorithm such as Newton's method or MATLAB's `fzero`**.

The method is second-order accurate, with a local truncation error of $O(h^3)$ and a global error of $O(h^2)$. It is also A-stable, meaning its region of absolute stability includes the entire left half of the complex plane. This property makes it particularly suitable for stiff differential equations, where explicit methods would require prohibitively small step sizes to maintain stability.

The Implicit Trapezoid Method is symplectic for Hamiltonian systems and conserves certain geometric properties, which is advantageous for long-term integration. Its primary drawback is the computational cost associated with solving a nonlinear equation at each step, which may require iterative solvers and careful initialization.

In summary, the Implicit Trapezoid Method offers an excellent balance between accuracy, stability, and simplicity for non-stiff and mildly stiff problems, and serves as a foundational scheme in the family of implicit Runge–Kutta methods.

Implicit Trapezoid Method: Pseudocode

To solve the initial value problem dy/dt = f(t, y), y(t0) = y0 using the Implicit Trapezoid Method, perform the following steps:

1. Define the time grid:

• Set initial time t0, final time T, and step size h.

• Compute number of steps: N = (T - t0) / h.

• Generate time nodes: t_n = t0 + n*h, for n = 0, 1, ..., N.

2. Initialize the solution array:

• Allocate vector y of size N+1.

• Set y(1) = y0 (initial condition).

3. For each time step n = 1 to N:

• Let t_n = t(n), t_{n+1} = t(n+1), and y_n = y(n).

• Define the implicit equation for the next value z = y_{n+1}:

G(z) = z - y_n - (h/2) * [f(t_n, y_n) + f(t_{n+1}, z)] = 0

• Solve the nonlinear equation G(z) = 0 for z using a root-finding method:

— If using fzero: provide G and an initial guess (e.g., z0 = y_n).

— If using Newton-Raphson:

* Choose initial guess z0 (e.g., z0 = y_n).

* Iterate: z_{k+1} = z_k - G(z_k) / G'(z_k),

where G'(z) = 1 - (h/2) * df/dy(t_{n+1}, z_k)

* Stop when |z_{k+1} - z_k| < tolerance.

• Set y(n+1) = z (the root found).

4. Return the arrays t and y as the numerical solution.

Note: The method is second-order accurate and A-stable, making it suitable for stiff problems. The main computational cost lies in solving the nonlinear equation at each step.

```
%% IMPLICIT TRAPEZOID – A-STABLE, 2ND ORDER

%% Implicit Trapezoid Method Implementation
y_trap_impl = zeros(1, N+1);
y_trap_impl(1) = y0;

for n = 1:N
    t_n = t_grid(n);
    t_np1 = t_grid(n+1);
    y_n = y_trap_impl(n);

    % Define implicit equation: G(z) = z – y_n – (h/2)*(f(t_n, y_n) +
f(t_np1, z)) = 0
    G = @(z) z – y_n – (h/2) * (f(t_n, y_n) + f(t_np1, z));
```

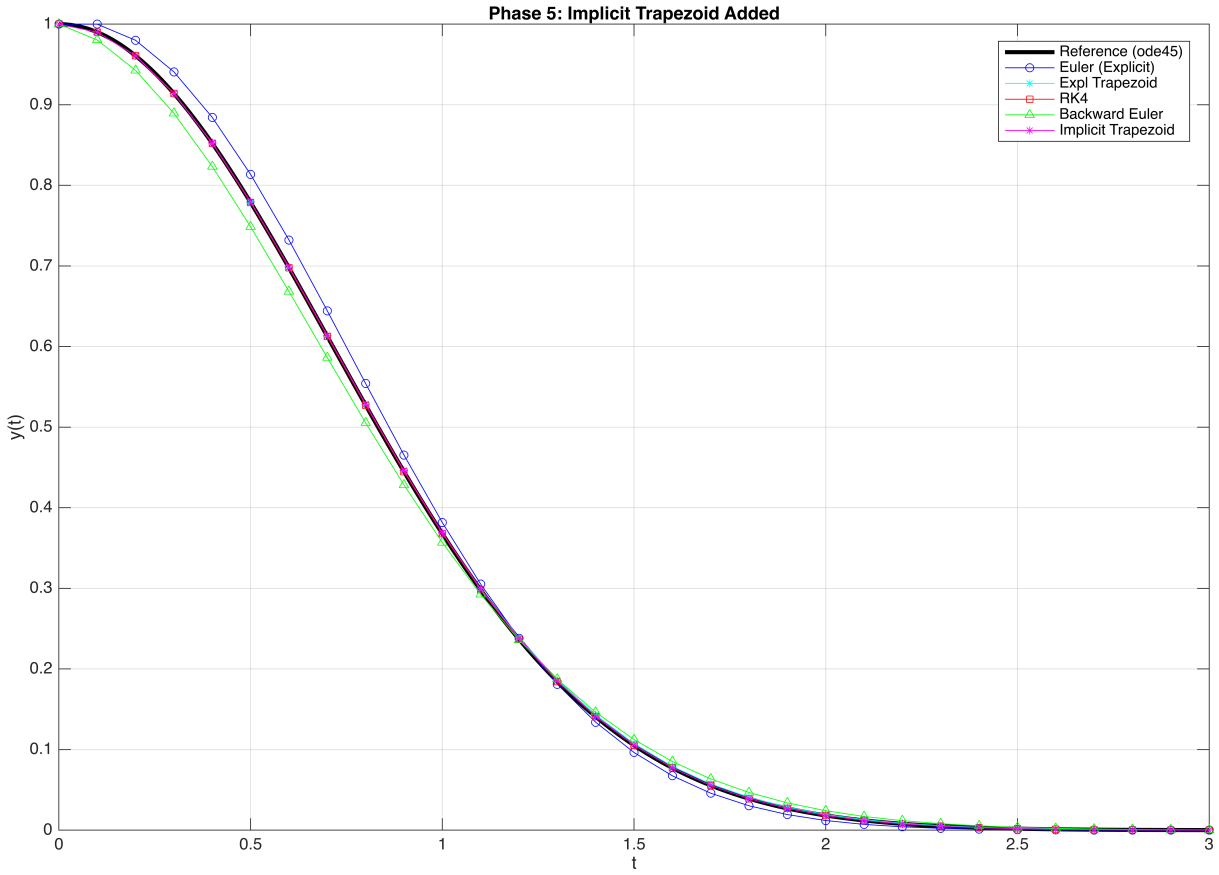
```
    % Solve using fzero
    y_trap_impl(n+1) = fzero(G, y_n);
end


% Error vs Reference
err_trap_impl = norm(y_trap_impl - y_exact);  % или y_reference(t_grid),
если используете ode45
```

```
=== PHASE 3: IMPLICIT TRAPEZOID ===
```

```
fprintf('Global Error (Implicit Trapezoid vs Reference): %.2e\n',
err_trap_impl);
```

```
Global Error (Implicit Trapezoid vs Reference): 3.47e-03
Improvement over Euler: 25.1x smaller error
```

```
%% Plot: All methods + Implicit Trapezoid
figure('Position', [100, 100, 900, 600]);
plot(t_plot, y_plot, 'k-', 'LineWidth', 2.5, 'DisplayName', 'Reference
(ode45)');
hold on;
plot(t_grid, y_euler,     'bo-', 'MarkerSize', 5, 'DisplayName', 'Euler
(Explicit)');
plot(t_grid, y_trap_expl, 'c*-', 'MarkerSize', 5, 'DisplayName', 'Expl
Trapezoid');
plot(t_grid, y_rk4,       'rs-', 'MarkerSize', 5, 'DisplayName', 'RK4');
plot(t_grid, y_beuler,    'g^-', 'MarkerSize', 5, 'DisplayName', 'Backward
Euler');
plot(t_grid, y_trap_impl, 'm*-', 'MarkerSize', 5, 'DisplayName', 'Implicit
Trapezoid');
xlabel('t');
ylabel('y(t)');
title('Phase 5: Implicit Trapezoid Added');
legend('Location', 'best');
grid on;
hold off;
```




```
%% Convergence Analysis for Implicit Trapezoid
errors_trap_impl = zeros(size(h_values));

for idx = 1:length(h_values)
    h_temp = h_values(idx);
    N_temp = (t_end - t0) / h_temp;
    t_temp = linspace(t0, t_end, N_temp+1);

    y_temp_trap_impl = zeros(1, N_temp+1);
    y_temp_trap_impl(1) = y0;
    for n = 1:N_temp
        t_n = t_temp(n);
        t_np1 = t_temp(n+1);
        y_n = y_temp_trap_impl(n);
        G = @(z) z - y_n - (h_temp/2) * (f(t_n, y_n) + f(t_np1, z));
        y_temp_trap_impl(n+1) = fzero(G, y_n);
    end

    y_exact_temp = y_reference(t_temp);
    errors_trap_impl(idx) = norm(y_temp_trap_impl - y_exact_temp);
end
```

```
%% Sort for convergence plot
[sorted_h_phase5, idx_phase5] = sort(h_values);
sorted_errors_euler_phase5 = errors_euler(idx_phase5);
sorted_errors_trap_expl_phase5 = errors_trap_expl(idx_phase5);
sorted_errors_rk4_phase5   = errors_rk4(idx_phase5);
sorted_errors_beuler_phase5 = errors_beuler(idx_phase5);
sorted_errors_trap_impl_phase5 = errors_trap_impl(idx_phase5);

%% Print full comparison table
fprintf('\n--- FULL COMPARISON: Step Size vs Error ---\n');
```

```
--- FULL COMPARISON: Step Size vs Error ---
```

```
fprintf('%8s %12s %12s %12s %12s %12s\n', 'h', 'Euler', 'Expl Trap', 'RK4',
'Backw Eul', 'Impl Trap');
```

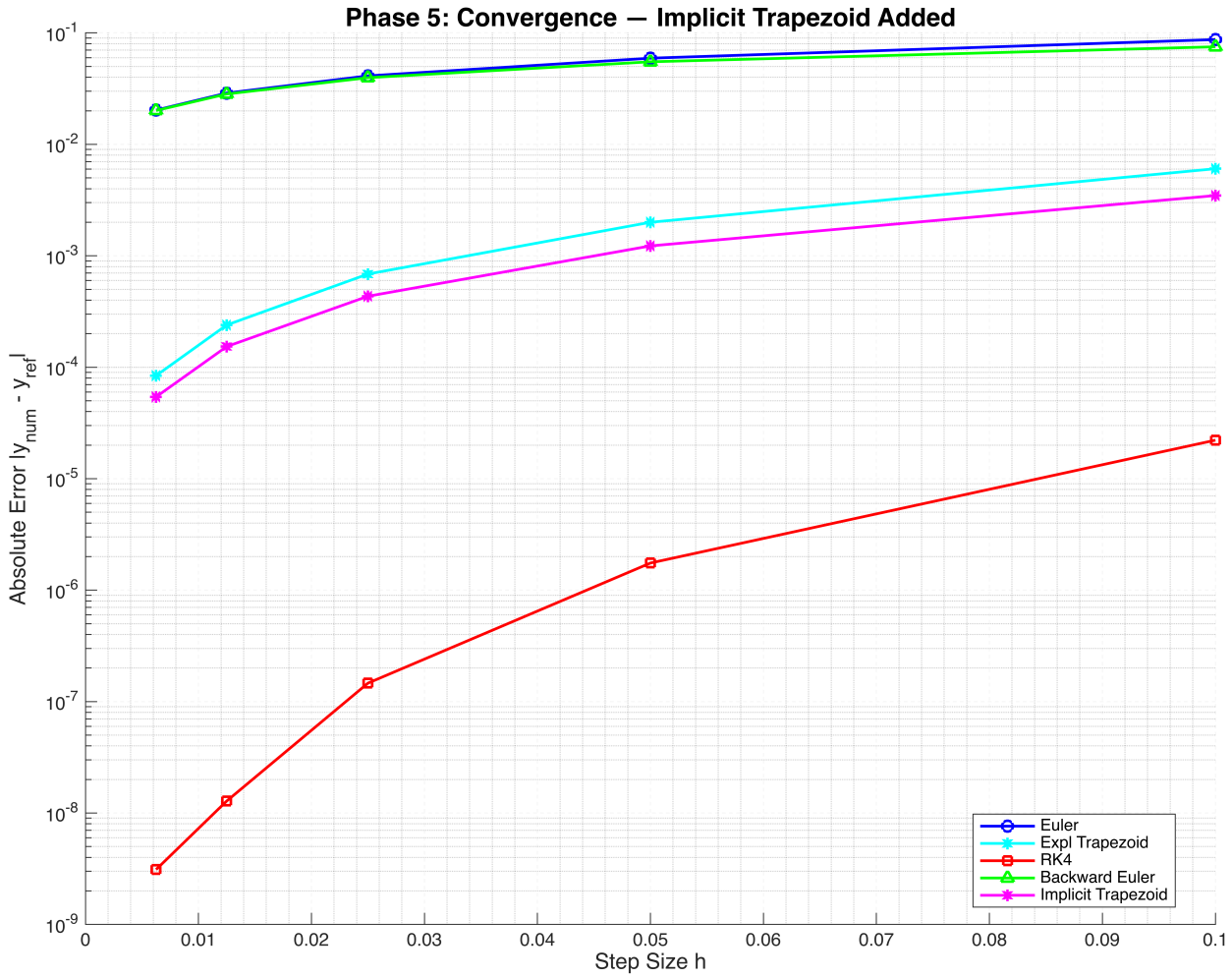
```
       h        Euler    Expl Trap          RK4    Backw Eul    Impl Trap
```

```
for i = 1:length(h_values)
    fprintf('%8.5f %12.2e %12.2e %12.2e %12.2e %12.2e\n', ...
        sorted_h_phase5(i), ...
        sorted_errors_euler_phase5(i), ...
        sorted_errors_trap_expl_phase5(i), ...
        sorted_errors_rk4_phase5(i), ...
        sorted_errors_beuler_phase5(i), ...
        sorted_errors_trap_impl_phase5(i));
end
```

```
 0.00625     2.03e-02     8.40e-05     3.11e-09     2.01e-02     5.42e-05
 0.01250     2.88e-02     2.39e-04     1.28e-08     2.83e-02     1.53e-04
 0.02500     4.12e-02     6.86e-04     1.46e-07     3.97e-02     4.34e-04
 0.05000     5.93e-02     2.00e-03     1.75e-06     5.51e-02     1.23e-03
 0.10000     8.71e-02     6.05e-03     2.22e-05     7.51e-02     3.47e-03
```

```
%% Convergence Plot
figure('Position', [100, 100, 800, 600]);
semilogy(sorted_h_phase5, sorted_errors_euler_phase5,      'b-o',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Euler');
hold on;
semilogy(sorted_h_phase5, sorted_errors_trap_expl_phase5,  'c*-',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Expl Trapezoid');
semilogy(sorted_h_phase5, sorted_errors_rk4_phase5,        'r-s',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'RK4');
semilogy(sorted_h_phase5, sorted_errors_beuler_phase5,     'g^-',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Backward Euler');
semilogy(sorted_h_phase5, sorted_errors_trap_impl_phase5,  'm*-',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Implicit Trapezoid');
xlabel('Step Size h', 'FontSize', 12);
ylabel('Absolute Error |y_{num} - y_{ref}|', 'FontSize', 12);
title('Phase 5: Convergence — Implicit Trapezoid Added', 'FontSize', 14,
'FontWeight', 'bold');
```

```
grid on; grid minor;
set(gca, 'GridLineStyle', '--', 'GridColor', [0.7, 0.7, 0.7]);
box off;
legend('Location', 'best');
```




```
%% Estimate Implicit Trapezoid convergence order
h_sub_trap_impl = sorted_h_phase5(3:end);
err_sub_trap_impl = sorted_errors_trap_impl_phase5(3:end);
if ~isempty(h_sub_trap_impl) && ~isempty(err_sub_trap_impl)
    p_trap_impl = log(err_sub_trap_impl(1)/err_sub_trap_impl(end)) /
log(h_sub_trap_impl(1)/h_sub_trap_impl(end));
    fprintf('Estimated convergence order (Implicit Trapezoid): %.3f
(expected: ~2.0)\n', p_trap_impl);
else
    fprintf('Not enough points to estimate Implicit Trapezoid convergence
order.\n');
end
```

```
Estimated convergence order (Implicit Trapezoid): 1.501 (expected: ~2.0)
```

4. Implicit Runge-Kutta Method (Gauss 4th Order): Formal Description

4.1 Derivation and Formula

Implicit Runge-Kutta (IRK) methods are a class of numerical integrators for ordinary differential equations that evaluate the function f at intermediate points within each step, where the values at these points depend implicitly on the solution at the end of the step. Unlike explicit methods, IRK methods require the solution of a system of nonlinear equations at each step, but offer superior stability properties and higher achievable order of accuracy.

The Gauss methods are a family of IRK schemes based on Gaussian quadrature. The two-stage Gauss method achieves fourth-order accuracy and is defined by the following Butcher tableau:

a11	a12
c1 | 1/4	1/4 - sqrt(3)/6
c2 | 1/4 + sqrt(3)/6	1/4
--------|----------	
| b1	b2
| 1/2	1/2

Given a uniform discretization of the interval $[t_0, T]$ with step size $ h $, and denoting $t_n = t_0 + nh$, the method computes the approximation $y_{n+1} \approx y(t_{n+1})$ by solving for the stage values k_1, k_2 that satisfy the implicit system:

$k_1 = f(t_n + c_1 h, y_n + h(a_{11}k_1 + a_{12}k_2)),$

$k_2 = f(t_n + c_2 h, y_n + h(a_{21}k_1 + a_{22}k_2)).$

The new solution is then given by: $y_{n+1} = y_n + h(b_1 k_1 + b_2 k_2)$.

This system of equations is generally nonlinear and must be solved numerically at each step using methods such as Newton-Raphson or **MATLAB's `fsolve`**.

4.3. Algorithmic Implementation (Pseudocode)

To implement the two-stage Gauss method for the initial value problem $dy/dt = f(t, y),\ y(t_0) = y_0$, perform the following steps:

1. Define the time grid:

• Set initial time t_0, final time T, and step size h.

• Compute number of steps: $N = (T - t_0)/h$.

• Generate time nodes: $t_n = t_0 + n \cdot h$, for $n = 0, 1, \ldots, N$.

2. Initialize the solution array:

• Allocate vector y of size $N + 1$.

• Set $y(1) = y_0$ (initial condition).

3. For each time step $n = 1$ to N:

• Define the system of nonlinear equations for stages k_1, k_2:

$G_1(k_1, k_2) = k_1 - f(t_n + c_1 h, y_n + h(a_{11}k_1 + a_{12}k_2)) = 0,$

$G_2(k_1, k_2) = k_2 - f(t_n + c_2 h, y_n + h(a_{21}k_1 + a_{22}k_2)) = 0.$

• Solve the system $\mathbf{G}(\mathbf{k}) = \mathbf{0}$ for $\mathbf{k} = [k_1, k_2]^T$ using a nonlinear solver (e.g., Newton-Raphson or `fsolve`).

— Use initial guess: $\mathbf{k}^{(0)} = [f(t_n, y_n), f(t_n, y_n)]^T$.

• Compute the next solution value: $y_{n+1} = y_n + h(b_1 k_1 + b_2 k_2)$.

4. Return the arrays t and y as the numerical solution.

Note: The method requires solving a system of nonlinear equations at each step. The choice of solver and initial guess may affect convergence and efficiency.

4.4. Accuracy and Convergence

The two-stage Gauss method is of fourth order, meaning that the global error is proportional to h^4. The local truncation error is of order $O(h^5)$, consistent with the fourth-order global convergence. The method is symplectic and preserves geometric properties of Hamiltonian systems, making it particularly suitable for long-term integration of conservative systems.

4.5. Stability and Applicability

The Gauss methods are A-stable and symplectic, making them suitable for stiff and conservative systems. The region of absolute stability includes the entire left half of the complex plane, ensuring unconditional stability for linear problems with negative real eigenvalues. The primary limitation is the computational cost associated with solving a nonlinear system at each step, which may require additional effort and careful initialization to ensure convergence.

```
%%  IMPLICIT RK4 (GAUSS 4TH ORDER)
%% 6. Implicit RK4 (Gauss 4th Order) Implementation
% Coefficients for 2-stage Gauss method (4th order)
```

```
c = [1/2 - sqrt(3)/6; 1/2 + sqrt(3)/6];  % nodes
A = [1/4, 1/4 - sqrt(3)/6;
     1/4 + sqrt(3)/6, 1/4];              % matrix
b = [1/2, 1/2];                          % weights

y_irk4 = zeros(1, N+1);
y_irk4(1) = y0;

options = optimoptions('fsolve', 'Display', 'off');

for n = 1:N
    t_n = t_grid(n);
    y_n = y_irk4(n);

    % Define system of nonlinear equations for stages K1, K2
    stage_eq = @(K) [
        K(1) - f(t_n + c(1)*h, y_n + h*(A(1,1)*K(1) + A(1,2)*K(2)));
        K(2) - f(t_n + c(2)*h, y_n + h*(A(2,1)*K(1) + A(2,2)*K(2)))
    ];

    % Initial guess for stages: use explicit Euler slope
    K0 = [f(t_n, y_n); f(t_n, y_n)];

    % Solve for stages K1, K2
    K = fsolve(stage_eq, K0, options);

    % Compute next step
    y_irk4(n+1) = y_n + h * (b(1)*K(1) + b(2)*K(2));
end

%% Error vs Reference
err_irk4 = norm(y_irk4 - y_exact);
fprintf('\n=== PHASE 6: IMPLICIT RK4 (GAUSS) METHOD ===\n');
```

```
=== PHASE 6: IMPLICIT RK4 (GAUSS) METHOD ===
```

```
fprintf('Global Error (Implicit RK4 vs Reference): %.2e\n', err_irk4);
```

```
Global Error (Implicit RK4 vs Reference): 2.70e-06
```

```
fprintf('Improvement over Euler: %.1fx smaller error\n', err_euler/
err_irk4);
```

```
Improvement over Euler: 32324.6x smaller error
```

```
fprintf('Improvement over Implicit Trapezoid: %.1fx smaller error\n',
err_trap_impl/err_irk4);
```

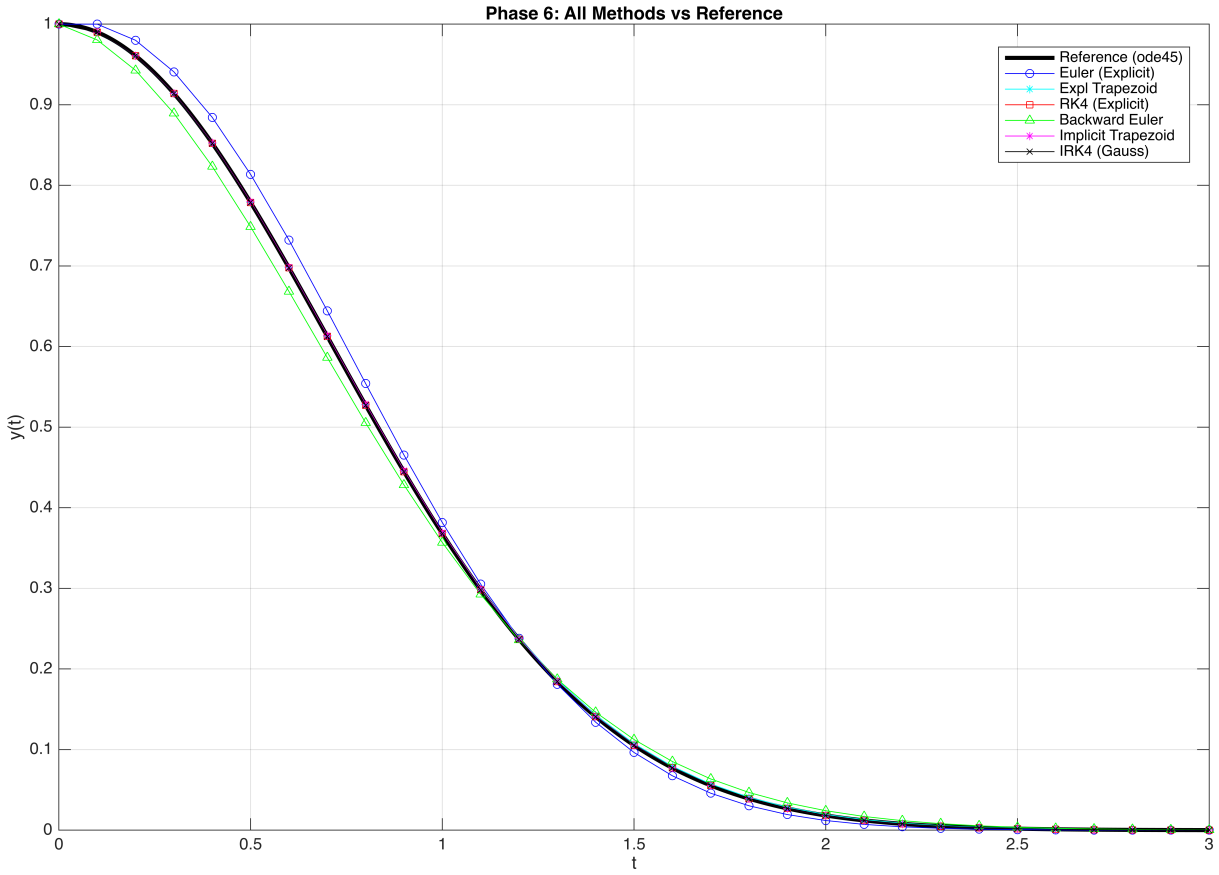
```
Improvement over Implicit Trapezoid: 1288.4x smaller error
```

```
if exist('p_rk4', 'var')
    if err_rk4 < err_irk4
        comp_str = 'more';
```

```
    else
        comp_str = 'less';
    end
    fprintf('Comparison with Explicit RK4: %.1fx %s accurate\n', ...
        max(err_rk4,err_irk4)/min(err_rk4,err_irk4), comp_str);
end
```

```
Comparison with Explicit RK4: 8.2x less accurate
```

```
%% Plot: All six methods
figure('Position', [100, 100, 900, 600]);
plot(t_plot, y_plot, 'k-', 'LineWidth', 2.5, 'DisplayName', 'Reference
(ode45)');
hold on;
plot(t_grid, y_euler,     'bo-', 'MarkerSize', 5, 'DisplayName', 'Euler
(Explicit)');
plot(t_grid, y_trap_expl, 'c*-', 'MarkerSize', 5, 'DisplayName', 'Expl
Trapezoid');
plot(t_grid, y_rk4,       'rs-', 'MarkerSize', 5, 'DisplayName', 'RK4
(Explicit)');
plot(t_grid, y_beuler,    'g^-', 'MarkerSize', 5, 'DisplayName', 'Backward
Euler');
plot(t_grid, y_trap_impl, 'm*-', 'MarkerSize', 5, 'DisplayName', 'Implicit
Trapezoid');
plot(t_grid, y_irk4,      'kx-', 'MarkerSize', 5, 'DisplayName', 'IRK4
(Gauss)');
xlabel('t');
ylabel('y(t)');
title('Phase 6: All Methods vs Reference');
legend('Location', 'best');
grid on;
hold off;
```




```
%% Convergence Analysis for Implicit RK4
errors_irk4 = zeros(size(h_values));

for idx = 1:length(h_values)
    h_temp = h_values(idx);
    N_temp = (t_end - t0) / h_temp;
    t_temp = linspace(t0, t_end, N_temp+1);

    y_temp_irk4 = zeros(1, N_temp+1);
    y_temp_irk4(1) = y0;

    for n = 1:N_temp
        t_n = t_temp(n);
        y_n = y_temp_irk4(n);

        stage_eq = @(K) [
            K(1) - f(t_n + c(1)*h_temp, y_n + h_temp*(A(1,1)*K(1) +
A(1,2)*K(2)));
            K(2) - f(t_n + c(2)*h_temp, y_n + h_temp*(A(2,1)*K(1) +
A(2,2)*K(2)))
        ];
```

```
        K0 = [f(t_n, y_n); f(t_n, y_n)];
        K = fsolve(stage_eq, K0, options);

        y_temp_irk4(n+1) = y_n + h_temp * (b(1)*K(1) + b(2)*K(2));
    end

    y_exact_temp = y_reference(t_temp);
    errors_irk4(idx) = norm(y_temp_irk4 - y_exact_temp);
end

%% Sort for convergence plot
[sorted_h_phase6, idx_phase6] = sort(h_values);
sorted_errors_euler_phase6 = errors_euler(idx_phase6);
sorted_errors_trap_expl_phase6 = errors_trap_expl(idx_phase6);
sorted_errors_rk4_phase6   = errors_rk4(idx_phase6);
sorted_errors_beuler_phase6 = errors_beuler(idx_phase6);
sorted_errors_trap_impl_phase6 = errors_trap_impl(idx_phase6);
sorted_errors_irk4_phase6   = errors_irk4(idx_phase6);

%% Print full comparison table
fprintf('\n--- FULL COMPARISON: Step Size vs Error ---\n');
```

```
--- FULL COMPARISON: Step Size vs Error ---
```

```
fprintf('%8s %12s %12s %12s %12s %12s %12s\n', 'h', 'Euler', 'Expl Trap',
'RK4', 'Backw Eul', 'Impl Trap', 'IRK4');
```

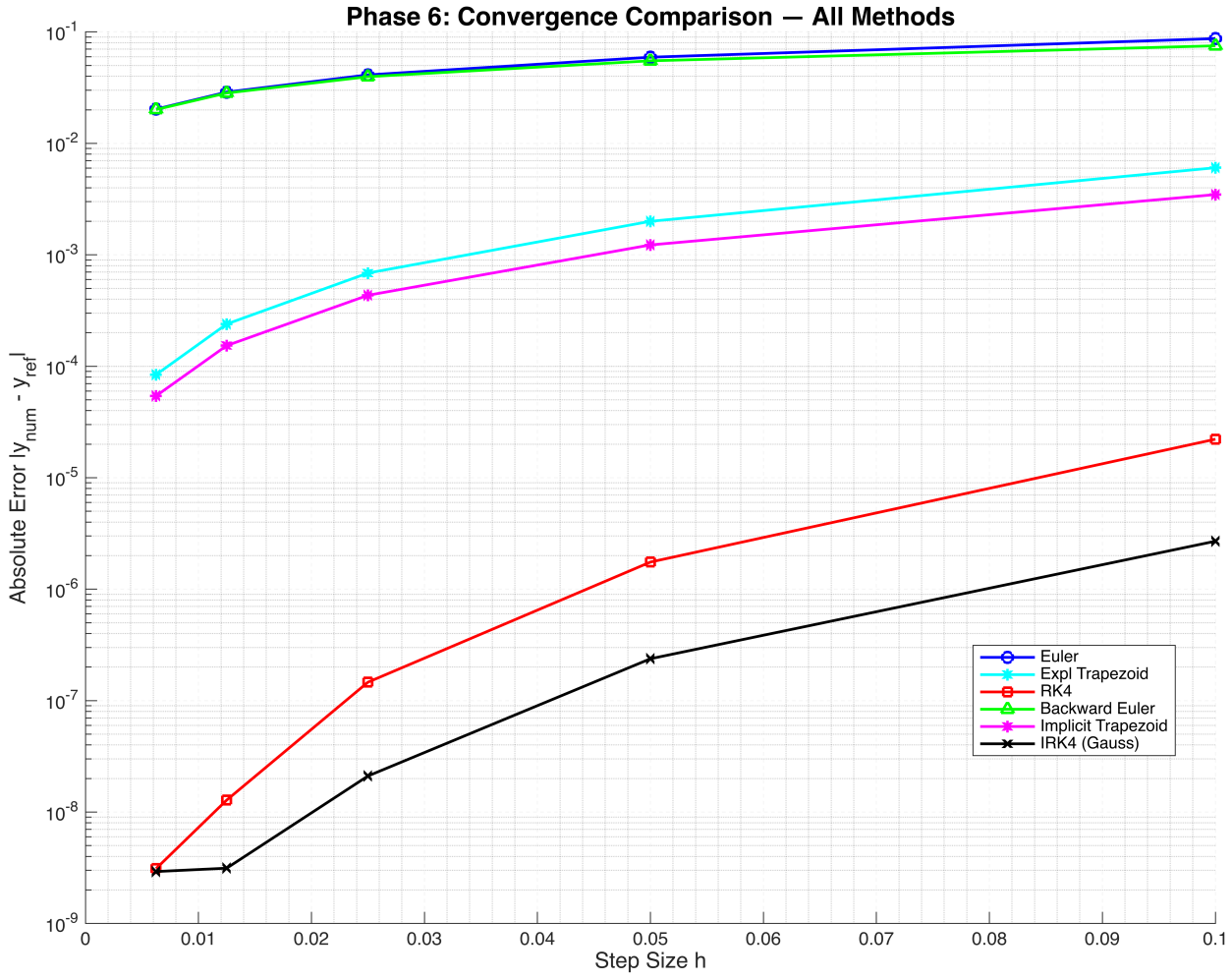
```
       h        Euler    Expl Trap          RK4    Backw Eul    Impl Trap         IRK4
```

```
for i = 1:length(h_values)
    fprintf('%8.5f %12.2e %12.2e %12.2e %12.2e %12.2e %12.2e\n', ...
        sorted_h_phase6(i), ...
        sorted_errors_euler_phase6(i), ...
        sorted_errors_trap_expl_phase6(i), ...
        sorted_errors_rk4_phase6(i), ...
        sorted_errors_beuler_phase6(i), ...
        sorted_errors_trap_impl_phase6(i), ...
        sorted_errors_irk4_phase6(i));
end
```

```
 0.00625     2.03e-02     8.40e-05     3.11e-09     2.01e-02     5.42e-05     2.93e-09
 0.01250     2.88e-02     2.39e-04     1.28e-08     2.83e-02     1.53e-04     3.14e-09
 0.02500     4.12e-02     6.86e-04     1.46e-07     3.97e-02     4.34e-04     2.11e-08
 0.05000     5.93e-02     2.00e-03     1.75e-06     5.51e-02     1.23e-03     2.38e-07
 0.10000     8.71e-02     6.05e-03     2.22e-05     7.51e-02     3.47e-03     2.70e-06
```

```


%% Convergence Plot
figure('Position', [100, 100, 800, 600]);
semilogy(sorted_h_phase6, sorted_errors_euler_phase6,      'b-o',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Euler');
hold on;
```

```
semilogy(sorted_h_phase6, sorted_errors_trap_expl_phase6,  'c*-',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Expl Trapezoid');
semilogy(sorted_h_phase6, sorted_errors_rk4_phase6,        'r-s',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'RK4');
semilogy(sorted_h_phase6, sorted_errors_beuler_phase6,     'g^-',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Backward Euler');
semilogy(sorted_h_phase6, sorted_errors_trap_impl_phase6,  'm*-',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'Implicit Trapezoid');
semilogy(sorted_h_phase6, sorted_errors_irk4_phase6,       'kx-',
'LineWidth', 1.5, 'MarkerSize', 6, 'DisplayName', 'IRK4 (Gauss)');
xlabel('Step Size h', 'FontSize', 12);
ylabel('Absolute Error |y_{num} - y_{ref}|', 'FontSize', 12);
title('Phase 6: Convergence Comparison — All Methods', 'FontSize', 14,
'FontWeight', 'bold');
grid on; grid minor;
set(gca, 'GridLineStyle', '--', 'GridColor', [0.7, 0.7, 0.7]);
box off;
legend('Location', 'best');
```



```
%% Estimate Implicit RK4 convergence order
h_sub_irk4 = sorted_h_phase6(3:end);
err_sub_irk4 = sorted_errors_irk4_phase6(3:end);
if ~isempty(h_sub_irk4) && ~isempty(err_sub_irk4)
    p_irk4 = log(err_sub_irk4(1)/err_sub_irk4(end)) / log(h_sub_irk4(1)/
h_sub_irk4(end));
    fprintf('Estimated convergence order (Implicit RK4): %.3f (expected:
~4.0)\n', p_irk4);
else
    fprintf('Not enough points to estimate Implicit RK4 convergence
order.\n');
end
```

```
Estimated convergence order (Implicit RK4): 3.499 (expected: ~4.0)
```

```
%% Final Summary
fprintf('\n=== FINAL SUMMARY ===\n');
```

```
=== FINAL SUMMARY ===
```

```
fprintf('Euler (Explicit)      global error: %.2e | order: %.2f\n',
err_euler, p_euler);
```

```
Euler (Explicit)      global error: 8.71e-02 | order: 0.54
```

```
if exist('p_trap_expl', 'var')
    fprintf('Expl Trapezoid        global error: %.2e | order: %.2f\n',
err_trap_expl, p_trap_expl);
end
```

```
Expl Trapezoid        global error: 6.05e-03 | order: 1.57
```

```
if exist('p_rk4', 'var')
    fprintf('RK4 (Explicit)        global error: %.2e | order: %.2f\n',
err_rk4, p_rk4);
end
```

```
RK4 (Explicit)        global error: 2.22e-05 | order: 3.62
```

```
if exist('p_beuler', 'var')
    fprintf('Backward Euler        global error: %.2e | order: %.2f\n',
err_beuler, p_beuler);
end
```

```
Backward Euler        global error: 7.51e-02 | order: 0.46
```

```
if exist('p_trap_impl', 'var')
    fprintf('Implicit Trapezoid    global error: %.2e | order: %.2f\n',
err_trap_impl, p_trap_impl);
end
```

```
Implicit Trapezoid    global error: 3.47e-03 | order: 1.50
```

```
if exist('p_irk4', 'var')
    fprintf('IRK4 (Gauss)          global error: %.2e | order: %.2f\n',
err_irk4, p_irk4);
end
```

```
IRK4 (Gauss)          global error: 2.70e-06 | order: 3.50
```