Efficiency Evaluation of Forward Euler Method-Based Numerical Methods for Approximating 2nd-order Ordinary Differential Equations

JQX051[1]

Higher Level Mathematics Analysis and Approaches

I. Introduction

In writing an extended essay on airfoil efficiency optimization, I used a computational fluid dynamics solver (CFD) to approximate forces exerted on rigid bodies by fluids. Fundamentally, the program solved ordinary and partial differential equations of varying orders among other intermediary processes. Each iteration required considerable computational capacity and time, and thus, finding the most efficient numerical method for solving ordinary differential equations (ODEs) would allow processes like CFD solving to be optimized. The efficiency of a method, as defined for this exploration, is maximized when complexity—measured through the number of operations—is minimized. Efficiency metrics evaluated in this exploration include convergence rate and complexity-accuracy ratios.

II. Numerical Methods

Three ground-level approaches to approximating second order ODEs will be evaluated. Namely, the Euler method, the Runge-Kutta midpoint (RK2) method, and the multistep predictor-corrector (P-C) method. These foreward Euler method-based methods provide varying approaches with the same fundamentals for computing solutions to differential equations, as will be detailed, which enables effective efficency comparisons.

In describing each method, the second-order ODE describing the simple harmonic motion of a mass-pendulum system,

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -\omega^2 x, \tag{1}$$

will be used for sample calculations. For all methods, the equation must be decomposed into a system of first-order ODEs:

[1]

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t} &= v \\ \frac{\mathrm{d}^2x}{\mathrm{d}t^2} &= \frac{\mathrm{d}v}{\mathrm{d}t} = -\omega^2 x \end{aligned} \tag{2}$$

The samples calculations will consider the initial conditions $\omega = \sqrt{10}$, $x_0 = 1$, $v_0 = 0$, and $h = 0.01$ where ω is the angular frequency of the system, x_0 is the initial displacement of the mass from equilibrium, v_0 is the initial velocity of the mass, and h is the iterative step size.

A. Forward Euler Method

The forward Euler method is an explicit method that uses the slope at the point to approximate the function's value at the next point. To find x_{n+1} and v_{n+1} at time $t_{n+1} = t_n + h$, the following equations are forumlated:

$$\begin{aligned} v_{n+1} &= v_n + h \times -\omega^2 x_n \\ x_{n+1} &= x_n + h \times v_n. \end{aligned} \tag{3}$$

Given the prescribed initial conditions, the ODE can be approximated with the foreward Euler method as follows:

$$\begin{aligned} v_{n+1} &= 0 + 0.01 \times -10 = -0.1 \\ x_{n+1} &= 1 + 0.01 \times 0 = 1. \end{aligned} \tag{4}$$

To predict the motion of the pendulum system over time, these final values are used to approximate the next iteration. The RK2 method involves 2 main calculations per iteration for 2nd order ODEs, and thus, the number of operations is proportional to the number of iterations.

B. Runge-Kutta Midpoint (RK2) Method

The Runge-Kutta Midpoint (RK2) method is a numerical method for computing 2nd order ODEs based on the higher order RK4 method. RK2 adopts a similar approach to the foreward Euler method, but introduces a half step (midpoint) between iterations to improve the accuracy of the approximations, as illustrated in Fig. 1.

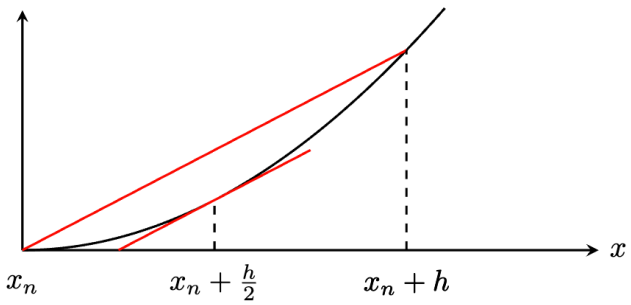

Fig. 1 RK2 Method

To find x_{n+1} and v_{n+1} at time $t_{n+1} = t_n + h$, the derivative k of the initial point is used to calculate values at midpoint m:

$$\begin{aligned} k1_v &= -\omega^2 x_n \\ k1_x &= v_n \\ m_v &= v_n + \frac{1}{2} h \times k1_v \\ m_x &= x_n + \frac{1}{2} h \times k1_x. \end{aligned} \tag{5}$$

The slopes at the midpoint are then used to calculate the final values:

$$\begin{aligned} k2_v &= -\omega^2 m_x \\ k2_x &= m_v \\ x_{n+1} &= x_n + h \times k2_x \\ v_{n+1} &= v_n + h \times k2_v. \end{aligned} \tag{6}$$

Given the prescribed initial conditions, the ODE can be approximated with RK2 as follows, starting with the half step,

$$\begin{aligned} k1_v &= -10 \times 1 = -10 \\ k1_x &= 0 \\ m_v &= 0 + \frac{1}{2} \times 0.01 \times (-10) = -0.05 \\ m_x &= 1 + \frac{1}{2} \times 0.01 \times 0 = 1, \end{aligned} \tag{7}$$

followed by the full step,

$$\begin{aligned} k2_v &= -10 \times 1 = -10 \\ k2_x &= -0.05 \\ x_{n+1} &= 1 + 0.01 \times (-0.05) = 0.9995 \\ v_{n+1} &= 0 + 0.01 \times (-10) = -0.1. \end{aligned} \tag{8}$$

The RK2 method involves 8 main calculations per iteration for 2nd order ODEs, and thus, the number of operations is proportional to the number of iterations.

C. Multistep Predictor-Corrector Method (P-C)

The predictor-corrector (P-C) method is a multistep explicit method involving a predictor step, in which the values are predicted using the foreward Euler method, then adjusted in the corrector step using the Adams-Bashforth

Method. To find x_{n+1} and v_{n+1} at time $t_{n+1} = t_n + h$, the foreward Euler method is applied to predict v and y at time t_{n+1}:

$$\begin{aligned} v_{\text{prediction}} &= v_n + h \times -\omega^2 x_n \\ x_{\text{prediction}} &= x_n + h \times v_n. \end{aligned} \tag{9}$$

This is followed by the corrector step, which is applied conditionally. For the first iteration where a previous v value is unavailable,

$$v_{\text{corrected}} = v_{\text{prediction}}. \tag{10}$$

Otherwise, for later iterations where a previous v value is available,

$$v_{\text{corrected}} = v_n + h\left(\frac{3}{2}(-\omega^2 x_n) - \frac{1}{2}(-\omega^2 x_{n-1})\right). \tag{11}$$

In both cases, $x_{\text{corrected}}$ is calculated as

$$x_{\text{corrected}} = x_n + \frac{h}{2}(v_n + v_{\text{corrected}}). \tag{12}$$

For the computation of following iterations, the following assignments can be made:

$$\begin{aligned} x_{n+1} &= x_{\text{corrected}} \\ v_{n+1} &= v_{\text{corrected}}. \end{aligned} \tag{13}$$

Given the prescribed initial conditions, the ODE can be approximated with P-C as follows, starting with the prediction,

$$\begin{aligned} v_{\text{prediction}} &= 0 + 0.01 \times (-10 \times 1) = -0.1 \\ x_{\text{prediction}} &= 1 + 0.01 \times 0 = 1, \end{aligned} \tag{14}$$

followed by the corrector,

$$\begin{aligned} v_{n+1} &= -0.1 \quad \text{(since it is the first step)} \\ x_{n+1} &= 1 + \frac{0.01}{2}(0 - 0.1) = 0.9995. \end{aligned} \tag{15}$$

To predict the motion of the pendulum system over time, these final values are used to approximate the next iteration following Equation (11). The P-C method involves 4 main calculations per iteration for 2nd order ODEs.

III. Efficiency Metrics

The efficiency of each numerical method can be quantified by its computational metrics. Namely, its convergence rate, complexity, and its complexity-accuracy ratio. To evaluate each metric, each numerical method was employed to approximate two real-world initial value problems (IVPs): The equation of motion of a simple harmonic oscillator (IVP1),

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -\omega^2 x, \tag{16}$$

and the equation of motion of a damped harmonic oscillator (IVP2),

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + b\frac{\mathrm{d}x}{\mathrm{d}t} + kx = 0. \tag{17}$$

These 2nd order IVPs were selected for their linearity and homogeneity, allowing analytical solutions to be derived.

A. Convergence Rate

The convergence rate of a method is defined by its rate of reduction in error with respect its resolution (time step) h. A suitable truncation error metric for examining convergence is the root-mean-square-error (RMSE). The metric uses the same scale as the target variable, enabling a maximum absolute error threshold to be defined—a crucial consideration for computing physical quantities. RMSE quantifies the deviation between analytical and numerical solutions. Thus, to calculate it, analytical solutions are derived for comparison with the first (IVP1) using the standard second-order homogeneous ODE form:

$$\begin{aligned} \frac{\mathrm{d}^2x}{\mathrm{d}t^2} &= -\omega^2 x \\ \frac{\mathrm{d}^2x}{\mathrm{d}t^2} + \omega^2 x &= 0 \end{aligned} \tag{18}$$

letting $x'' = r^2$, we have

$$\begin{aligned} r^2 &= -\omega^2 \\ r &= \pm\omega i. \end{aligned} \tag{19}$$

Because r has complex conjugate roots in the form of $\alpha \pm \beta i$, its general solution is in the form

$$\begin{aligned} x(t) &= e^{\alpha t}[C_1\cos(\beta t) + C_2\sin(\beta t)], \\ \therefore x(t) &= C_1\cos(\omega t) + C_2\sin(\omega t). \end{aligned} \tag{20}$$

Considering the simple harmonic motion initial conditions $v_0 = 0$ and $x_0 = 1$, when we let $x(0) = x_0$, the cosine term is 1 and the sine term is 0, therefore, $C_1 = x_0$. Likewise, given $\frac{\mathrm{d}x}{\mathrm{d}t}(0) = v_0$,

$$\begin{aligned}
\frac{\mathrm{d}x}{\mathrm{d}t} &= -\omega C_1 \sin(\omega t) + \omega C_2 \cos(\omega t), \\
\therefore v_0 &= -\omega C_1 \sin(0) + \omega C_2 \cos(0) \\
v_0 &= \omega C_2 \\
C_2 &= \frac{v_0}{\omega}.
\end{aligned} \tag{21}$$

Thus, we arrive at the analytical solution

$$\boxed{x(t) = x_0 \cos(\omega t) + \frac{v_0}{\omega} \sin(\omega t).} \tag{22}$$

IVP2 is solved similarly using the standard second-order homogeneous ODE form:

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} + b\frac{\mathrm{d}x}{\mathrm{d}t} + kx = 0. \tag{23}$$

letting $x'' = r^2$, we have

$$\begin{aligned}
r^2 + br + k = 0, \\
\therefore r = \frac{-b \pm \sqrt{b^2 - 4k}}{2}.
\end{aligned} \tag{24}$$

For $b^2 - 4k > 0$, the following generalization can be made for computation when b and k are known, based on the general solution form:

$$\begin{aligned}
x(t) &= C_1 e^{r_1 t} + C_2 e^{r_2 t} \\
r_1 &= -b + \sqrt{b^2 - 4k} \\
r_2 &= -b - \sqrt{b^2 - 4k}.
\end{aligned} \tag{25}$$

Applying the initial conditions $v_0 = 0$ and $x_0 = 1$,

$$\begin{aligned}
1 &= C_1 + C_2 \\
0 &= C_1 r_1 + C_2 r_2, \\
\therefore C_2 &= x_0 - C_1, \\
\therefore C_1 &= \frac{v_0 - r_2 x_0}{r_2 - r_2}.
\end{aligned} \tag{26}$$

Thus,

$$\boxed{x(t) = \frac{v_0 - r_2 x_0}{r_2 - r_2} \cdot e^{-b + \sqrt{b^2 - 4k} \cdot t} + \left(x_0 - \frac{v_0 - r_2 x_0}{r_2 - r_2} \right) \cdot e^{-b - \sqrt{b^2 - 4k} \cdot t}.} \tag{27}$$

For $b^2 - 4k < 0$, the general solution is in the form

$$x(t) = e^{\alpha t}[C_1 \cos(\beta t) + C_2 \sin(\beta t)]. \tag{28}$$

Applying the initial conditions $v_0 = 0$ and $x_0 = 1$,

$$1 = e^0[C_1 \cos(0) + C_2 \sin(0)] = C_1 \tag{29}$$

and

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t} &= \alpha e^{\alpha t}[C_1 \cos(\beta t) + C_2 \sin(\beta t)] + e^{\alpha t}[-\beta C_1 \sin(\beta t) + \beta C_2 \cos(\beta t)] \\ &= e^{\alpha t}[(\alpha C_1 - \beta C_2)\cos(\beta t) + (\alpha C_2 - \beta C_1)\sin(\beta t)] \\ \therefore 0 &= e^0[(\alpha C_1 - \beta C_2)\cos(0) + (\alpha C_2 - \beta C_1)\sin(0)] \\ 0 &= \alpha C_2 - \beta C_1. \end{aligned} \tag{30}$$

Since $x_0 = C_1$,

$$C_2 = \frac{v_0 - \alpha x_0}{\beta}. \tag{31}$$

Thus, given $\alpha = -\dfrac{b}{2}$ and $\beta = \dfrac{\sqrt{4k - b^2}}{2}$ from the complex conjugate roots of r in the form $\alpha + \beta i$,

$$x(t) = e^{-\frac{1}{2}bt}\left[x_0 \cos\left(\frac{\sqrt{4k-b^2}}{2} \cdot t\right) + \frac{2v_0 + bx_0}{\sqrt{4k-b^2}} \sin\left(\frac{\sqrt{4k-b^2}}{2} \cdot t\right)\right]. \tag{32}$$

Using the analytical solutions in Equation (22), Equation (27), and Equation (32), the RMSE values for each resolution can be calculated per the formula

$$\mathrm{RMSE}_h = \sqrt{\sum_{i=1}^{N} \frac{|\widehat{x_i} - x_i|^2}{N}} \tag{33}$$

using numerically approximated values $\widehat{x_i}$ and analytically derived values x_i, where $N = \left\lfloor \dfrac{t_f - t_0}{h} \right\rfloor$. To evaluate the convergence rate of each method, test case initial conditions are defined for each IVP:

$$\text{Test Case 1 (IVP1)} = \begin{cases} v_0 = 0 \\ x_0 = 1 \\ t_0 = 0 \\ t_f = 5 \\ \omega = \sqrt{10} \end{cases} \qquad \text{Test Case 2 (IVP2)} = \begin{cases} v_0 = 0 \\ x_0 = 1 \\ t_0 = 0 \\ t_f = 5 \\ b = 10 \\ k = 100 \end{cases}$$

Test cases 1 and 2 represent stable and semi-stiff equations commonly encountered in physics, which are representative of the array of initial conditions in CFD computations. The numerical solutions are computed until the solution converges to a maximum absolute truncation error threshold of $\mathrm{RMSE} = 0.001$. To start, from computing test case 1 at an arbitrary $h = 2^{-6}$ with the Euler method for visualization, the absolute error of the Euler method oscillates noticably increases over time as seen in Fig. 2.

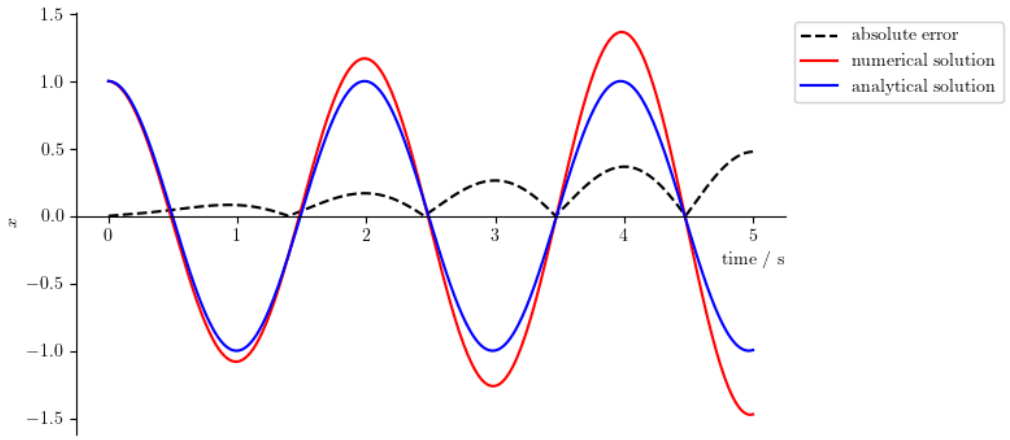

Fig. 2 Euler Method Absolute Error for Test Case 1 at $h = 2^{-6}$

Computing test case 1 at the same time step ($h = 2^{-6}$) for each numerical method reveals oscillating absolute error magnitudes over time for all methods. Fig. 3 reveals a significant linear increase in the absolute error maxima of the Euler and P-C methods each oscillatory period, while RK2 sees a gradual decrease.

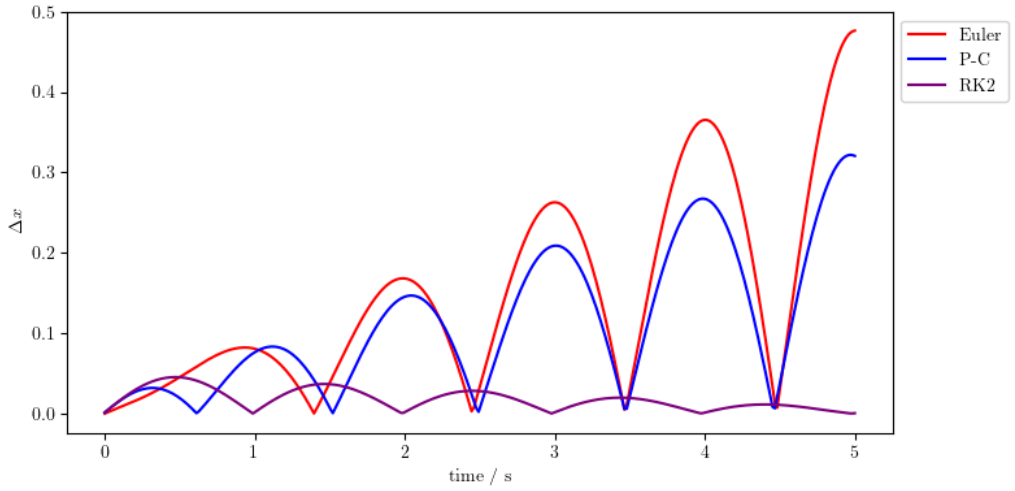

Fig. 3 Euler vs. P-C vs. RK2 Method Errors for Test Case 1 at $h = 2^{-6}$

The error distribution of each method at different resolutions ($\mathrm{d}t$) can be plotted separately to reveal the converging behavior of each method. By incrementally refining resolutions by negative powers of two (halving the time step), the error a method produces can be examined at varying orders of magnitude to deduce its convergence rate. As seen in Fig. 4, the cumulative error decreases with time step for all methods. At lower resolutions ($\mathrm{d}t > 2^{-2}$), the error linearly diverges over time for all methods and test cases, indicating the instability of the IVPs. The Euler method exhibits numerical instability for the greatest range of resolutions, followed by P-C, then RK2, which appears to start converging from resolution $\mathrm{d}t = 2^{-3}$.

Per Fig. 4, test case 2 reveals the shortcomings of the Euler and 2-step predictor-corrector methods—their instability at low resolutions for semi-stiff IVPs. RK2 starts to converge at the lowest resolution, but exhibits diminishing improvements in accuracy as the resolution increases.

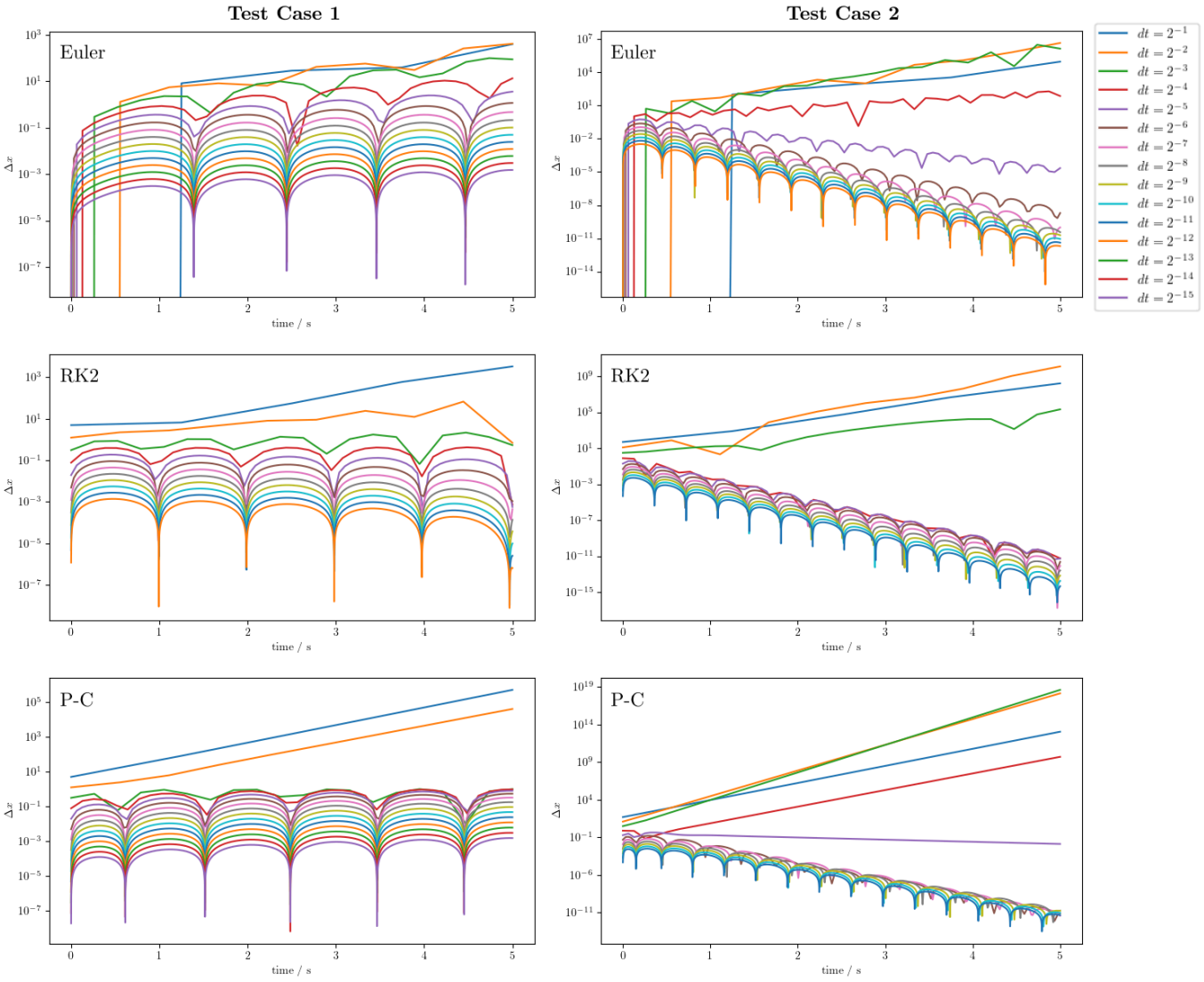

Fig. 4 Convergence of Numerical Approximations of Test Case 1 & 2

Plotting the RMSE value of each error distribution against the resolution on a log-log scale, the convergence rate can be calculated based on the rate of change in error with respect to resolution. Linearized best-fit equations on a log-log plot are given in the form

$$\begin{aligned} y &= e^{mx+b} \\ &= x^m \cdot e^c \end{aligned} \tag{34}$$

where m represents the order of change in error with respect to time step. Fig. 5 shows the convergence rate and scaling factor of each method and test case.

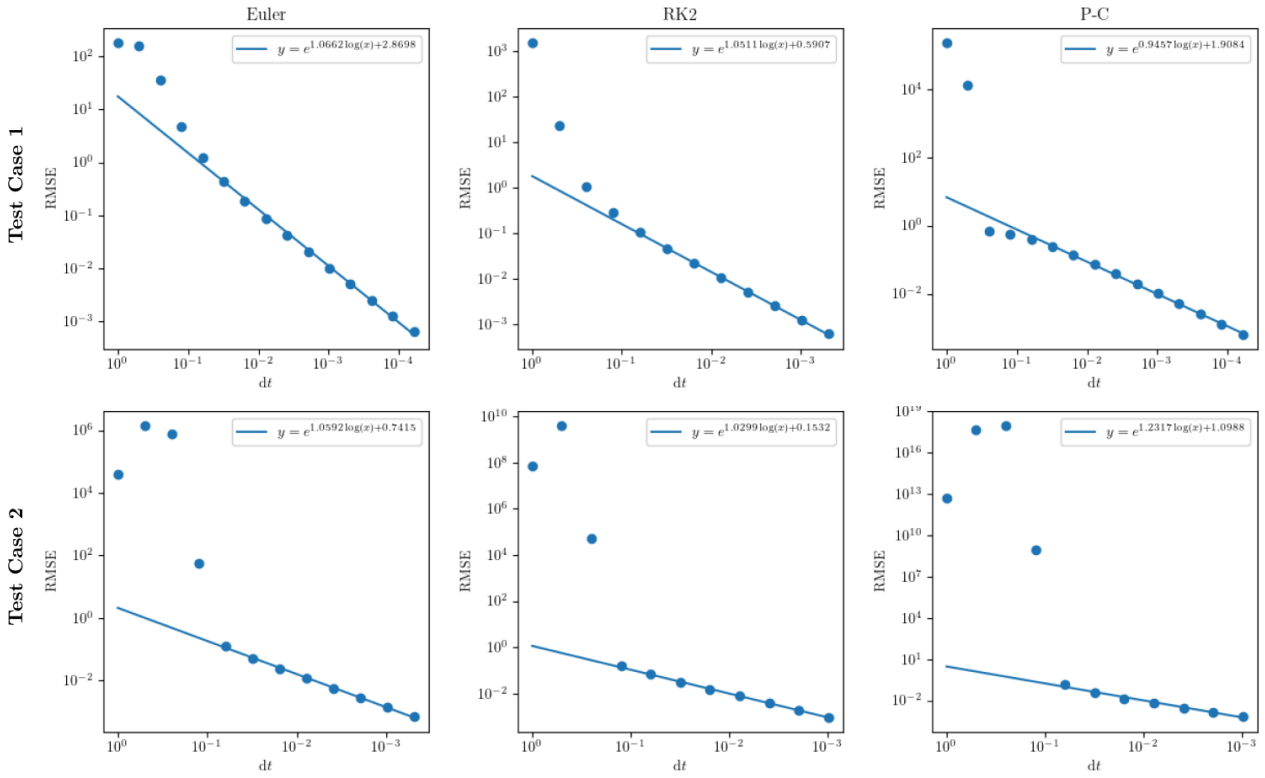

Fig. 5 Convergence Rate of Numerical Methods

In calculating the linearized curves of best fit, the points which exhibit instability were manually omitted, and the regression was calculated to fit data points that exhibit strong convergent behavior. The convergence rates from Fig. 5 are effectively compared in Fig. 6.

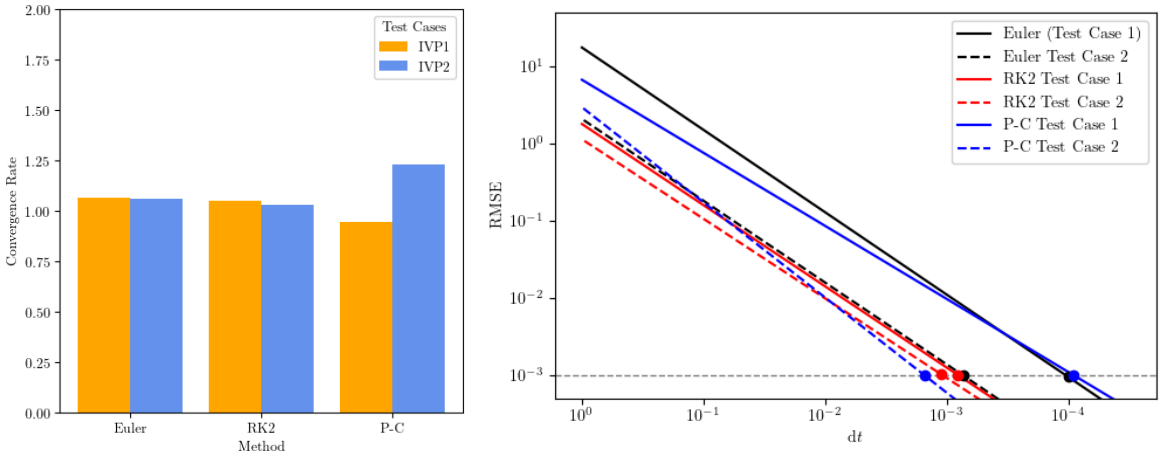

Fig. 6 Convergence of Numerical Methods

While the Euler and P-C methods vary in resolution required to reach the threshold maximum error between test cases, the RK2 method consistently succeeds the other methods for both test cases. Although all three methods exhibit first order convergence rates for the selected IVPs, the RK2 method is, in theory, a 2nd-order method. Thus,

RK2 is vastly more efficient for computing stiff ODEs than Euler and P-C, as seen in Fig. 6. Although the convergence patterns in Fig. 4 reveals the effectiveness of each method, the convergence rate itself is a weak indicator of efficiency for the selected IVP test cases.

B. Complexity

Before a numerical approximation is calculated, the optimal time step is unknown. Thus, numerical solutions are often computed until solutions converge to a desired threshold accuracy level. However, with exponentiating time steps, the cumulative error can accumulate to extremely large values over time, so the selection of a suitable starting time step is crucial for maximizing computational efficiency.

The complexity of a method is quantified by the number of operations it makes to calculate an approximation, which is proportional to computational runtime in single-threaded processing units. The Euler, P-C, and RK2 methods make 2, 4, and 8 calculations per iteration, respectively. The total number of operations a solution takes to converge from various starting time step can be calculated with these figures and used to determine a method's efficiency, per Fig. 7. The x-axis values, labeled $h_0 \text{ order}$, represent the order of the starting time step,

$$h_{\text{starting}} = \frac{h_{0\text{max}}}{2^{(h_0 \text{ order})}}, \tag{35}$$

and within the context of this exploration,

$$h_{\text{starting}} = \frac{1}{2^{(h_0 \text{ order})}} \tag{36}$$

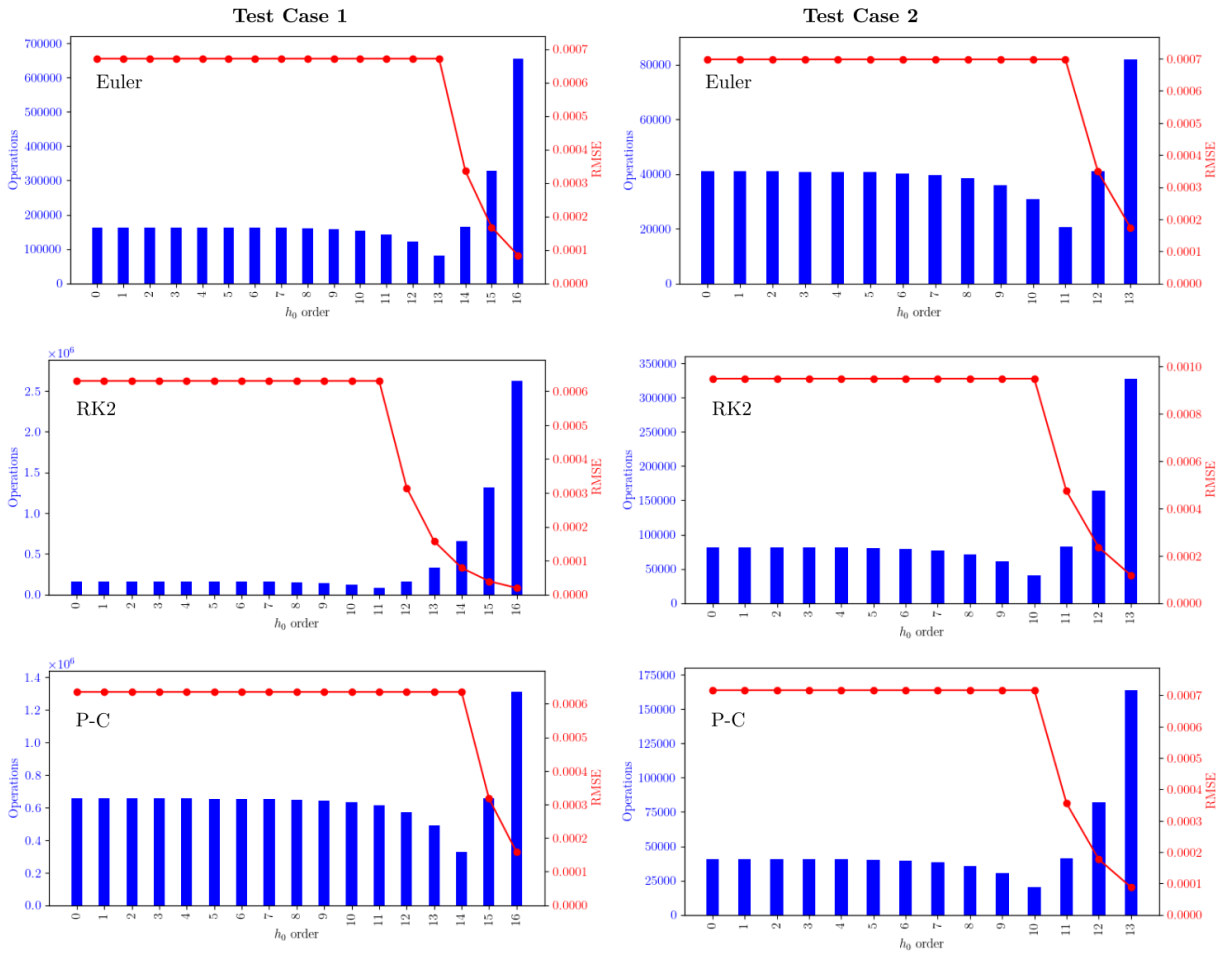

Fig. 7 Starting Time Step vs. Number of Operations

For all numerical methods the cumulative number of operations required to reach the threshold error slowly diminishes as the starting time step (h_{starting}) gets smaller. This is a result of fewer full approximation cycles being executed as h_{starting} approaches the threshold time step. At h_{starting} where the RMSE crosses the truncation error threshold—the *optimal starting time step*—the cumulative number of operations reaches a minimum, as only 1 cycle is computed. In Fig. 7, this is where the h_0 dip meets the last threshold RMSE point for every method and test case. Every successive h_{starting} value also only computes 1 cycle, but the number of operations per cycle exponentiaties, and thus, starting at such time steps is inefficient. Note that larger optimal starting time steps offer slightly more efficiency when selecting h_0 at random, but also presents a higher probability of overprecision, and thus, throttled complexity.

The RK2 method consistently has the highest number of operations for the first stable IVP, exceeding the other methods twofold for pre-optimal h_{starting} values, shown in Fig. 8. Its minimum complexity is also double that of the other two methods. The stiffer test case, however, sees RK2 reaching the optimal time stepsize first, and at the lowest minimum complexity, reinforcing its strength in approximating less stable IVPs. The Euler method consistently has the lowest complexity due to its simplicity, but has a smaller optimal starting time steps. The predictor-corrector

method performs exceedingly well for the first test case, almost mirroring the low complexity of the Euler method despite its greater number of operations per iteration. Overall, the Euler method has the lowest average complexity out of the three, and has the lowest rate of growth after the optimal starting time step, making it a strong method for calculations that prioritize speed.

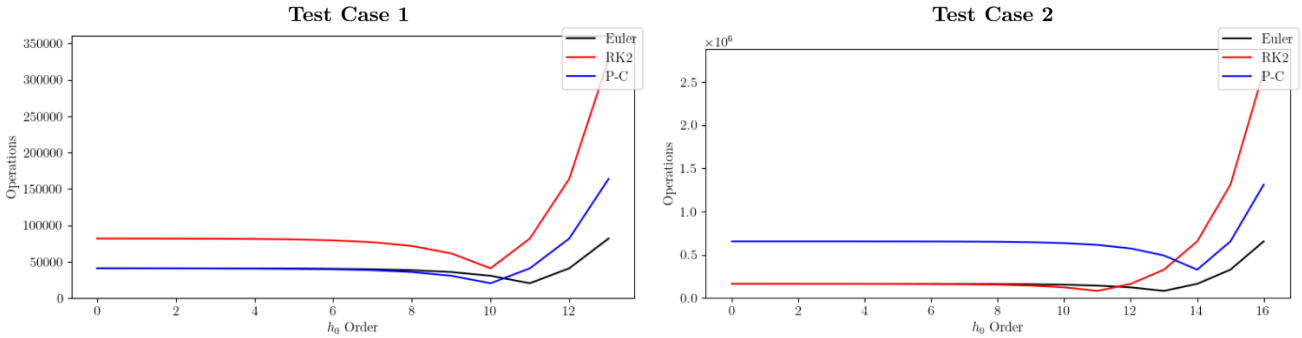

Fig. 8 Starting Time Step vs. Number of Operations

C. Complexity-Accuracy Comparison

To formulate a useful ratio for efficiency evaluation, one component must be maximized while the other is minimized. Because both RMSE and efficiency must be minimized for maximum efficiency, a simple accuracy metric is defined as the inverse of RMSE:

$$\text{accuracy} = \frac{1}{\text{RMSE}}. \tag{37}$$

According to this definition, the efficiency is maximized for computations with starting time step = h_0 and N_{h_0} operations when $N_{h_0} \cdot \text{RMSE}_f$ is minimized. The efficiency is plotted against starting time step for both test cases in Fig. 9. Each line in the plot resembles a logarithmic function reflected about a line parallel to the y-axis. This is simply an amplification of the decay in complexity, which was scaled by a constant threshold RMSE. The line reaches an abrubpt point—the optimal starting time step—after which the rate of change of the line becomes 0, as the exponential decrease in RMSE and exponential increase in runtime cancel out and produce a constant proportional to the convergence rate of the method and IVP.

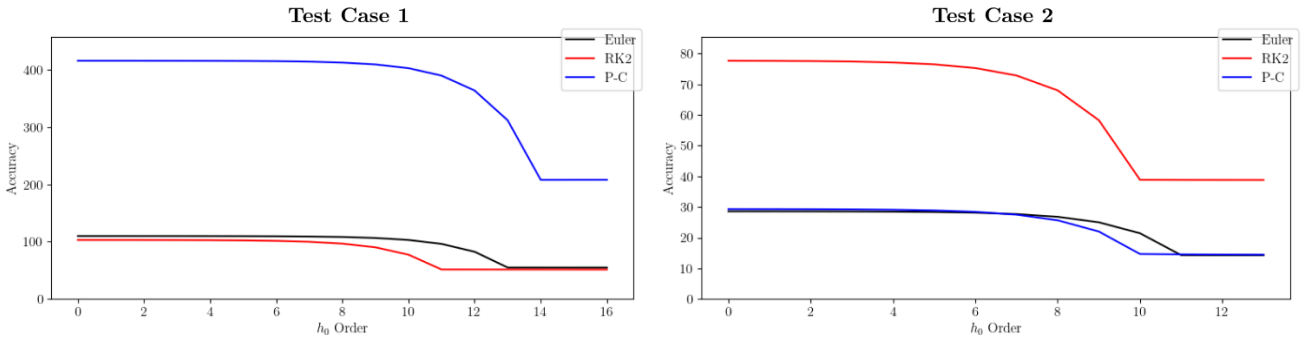

Fig. 9 Starting Time Step vs. Number of Operations

As seen in Fig. 9, the predictor-corrector method has the highest accuracy to starting time step ratio for the first, more stable test case. RK2 has the lowest accuracy to complexity ratio in this test case, exceeded by the Euler method due to the latter's later optimal starting time step. In the stiffer test case 2, RK2 nearly triples the efficiency of the other two methods. The magnitude reveals that RK2 is not more efficient, but is able to overcome the stiffness and instability of the IVP, while the other two methods take more iterations to reach the desired RMSE.

IV. Conclusion

The nexus of convergence rate, complexity, and complexity-accuracy metrics reveals the advantages and pitfalls of each foreward Euler method-based numerical method for approximating 2nd-order ordinary differential equations. Overall, the most efficient method for stiff problems and is the 2nd-order Runge-Kutta method, as revealed by the accuracy ratio and convergence rate, while the Adams-Bashforth predictor-corrector method excels in efficiency for more stable problems. While the Euler method struggles with efficiency due to its instability, it consistently has the lowest complexity. However, this is overshadowed by the equivalent complexities of the other two methods for specific stiffness conditions, making the Euler method the least efficient of the three. Overall, the 2nd-order Runge-Kutta method is the most *efficient* for stiff equations and the predictor-corrector method is the most *efficient* for smoother initial value problems. While the foreward Euler method is fundamental, it has been proven to be the least efficient.

In practice, the foreward-euler based methods evaluated in this exploration lack the capacity to approximate higher order ordinary and partial differential equations, which are cornerstones of the technology and innovation in this data-driven world. Thus, it is highly worth exploring higher-order numerical approximation methods that can solve the most prominent physical problems, like the *k-omega sst* turbulence model that inspired this exploration, to evaluate how current computational processes and software can be further optimized. With that said, rigorous methods such as the 4th-order Runge-Kutta method and higher order predictor-corrector methods are built upon the discussed foundations, bestowing significant value on this exploration of computational mathematics fundamentals.

References

[1] Burkardt, J., “The Midpoint and Runge Kutta Methods”, Jul 13 2011.

[2] P. Kavitha, and K. Prathiba, “Adams-Bashforth Corrector-Predictor Method Using MATLAB”, International Journal of Mechanical Engineering, Kalahari Journals, Apr2022.. Retrieved 8 January 2024. https://kalaharijournals.com/resources/APRIL_93.pdf

[3] Gupta, S., “RMSE: What Does It Mean?”, Apr2021.. https://medium.com/@mygreatlearning/rmse-what-does-it-mean-2d446c0b1d0e

[4] King, K., “Solving Second-Order Homogeneous Differential Equations”.. https://www.kristakingmath.com/blog/homogeneous-differential-equations-second-order

[5] LeMesurier, B., “5. Measures of Error and Order of Convergence — MATH 375. Elementary Numerical Analysis (With Python)”.. https://lemesurierb.people.cofc.edu/elementary-numerical-analysis-python/notebooks/error-measures-convergence-rates.html

Appendix

A. Test Case 1: Convergence

```
import numpy as np
import matplotlib.pyplot as plt
import scripts.tc1methods as tc1methods

plt.rc('text', usetex=True)
plt.rc('font', family='serif')

# Damped SHM
class IVP1: # SHM
    def __init__(self, omega, y0, v0, t0, tf, rmsef) -> None:
        self.omega = omega # Angular frequency
        self.y0 = y0 # Initial position
        self.v0 = v0 # Initial velocity
        self.t0 = t0 # Initial time
        self.tf = tf # Final time
        self.rmsef = rmsef # Target L2 norm

    def function(self, y, omega):
        return -omega**2 * y

    def analytical_solution(self, t, y0, v0, omega):
        A = y0
        B = v0 / omega
        return A * np.cos(omega * t) + B * np.sin(omega * t)

# Configure test case
IVP1TC1 = IVP1(np.sqrt(10), 1, 0, 0, 5, 0.001)
tc = IVP1TC1

# Configure starting step size
H0MAX = 1

l2norm = {}
data = {}

target_rmsef = tc.rmsef
tf = tc.tf
```

```
t0 = tc.t0
omega = tc.omega
y0 = tc.y0
v0 = tc.v0

print(tc.tf)
print(tc.rmsef)

h = H0MAX
rmse = 1

step = 0
plt.figure(figsize=(8, 4))

# Convergence Step
while rmse > target_rmsef:
    N = int((tf - t0) / h)
    t_values = np.linspace(t0, tf, N)
    v_prev = None
    step += 1
    # Initial values
    y = y0
    v = v0

    errors = []
    y_values = []
    v_values = []
    analytical_values = []

    # Euler Method
    for t in t_values:
        y, v, v_predict = tc1methods.euler(tc, y, v, h, omega, v_prev) # Configure Method
        v_prev = v

        y_values.append(y)
        v_values.append(v)
        y_analytical = tc.analytical_solution(t, y0, v0, omega)

        analytical_values.append(y_analytical)
```

```
        error = abs(y - y_analytical)
        errors.append(error)

    rmse = np.sqrt(np.mean(np.array(errors)**2))
    l2norm[h] = rmse
    print(h)
    print(rmse)
    print("--------")
    h = h / 2

    # Plot the errors for the current step size
    plt.plot(t_values, errors, label=f'$dt=2^{{-{step}}}$')

# Finalize the plot
plt.xlabel('time / s')
plt.ylabel('$\Delta x$')
plt.yscale('log')
plt.legend()
plt.legend(loc=' leftupper', bbox_to_anchor=(1, 1))
plt.tight_layout()

plt.savefig('convergence.png', bbox_inches='tight')



# Plot Convergence Rate Graph
xval, yval = zip(*sorted(l2norm.items()))

plt.figure(figsize=(4, 4))
plt.loglog()
plt.scatter(xval, yval)
plt.xlabel('d$t$')
plt.ylabel('RMSE')
plt.title('Euler') # Configure Method Name
plt.gca().invert_xaxis()
plt.tick_params(axis='x', which='minor', bottom=False)
plt.tick_params(axis='y', which='minor', left=False)

h_threshold = 10**(-1)
```

```
stable_xval = [x for x in xval if x < h_threshold]
stable_yval = [y for x, y in zip(xval, yval) if x < h_threshold]

m, c = np.polyfit(np.log(stable_xval), np.log(stable_yval), 1)
y_fit = np.exp(m*np.log(xval) + c)

plt.plot(xval, y_fit, label=f"$y = e^{{{m:.4f}\\log(x) + {c:.4f}}}$")
plt.tight_layout()
plt.legend()

print(f"y = e^({m:.4f}*log(x) + {c:.4f}")

plt.savefig('convergencerate.png', bbox_inches='tight')

# Show the plot
plt.show()
```

B. Test Case 1: Efficiency

```
import numpy as np
import matplotlib.pyplot as plt
import tc1methods as tc1methods

plt.rc('text', usetex=True)
plt.rc('font', family='serif')

# Damped SHM
class IVP1: # SHM
    def __init__(self, omega, y0, v0, t0, tf, rmsef) -> None:
        self.omega = omega # Angular frequency
        self.y0 = y0 # Initial position
        self.v0 = v0 # Initial velocity
        self.t0 = t0 # Initial time
        self.tf = tf # Final time
        self.rmsef = rmsef # Target L2 norm

    def function(self, y, omega):
        return -omega**2 * y

    def analytical_solution(self, t, y0, v0, omega):
        A = y0
        B = v0 / omega
        return A * np.cos(omega * t) + B * np.sin(omega * t)

# Configure test case
IVP1TC1 = IVP1(np.sqrt(10), 1, 0, 0, 5, 0.001)
tc = IVP1TC1

# Configure starting step size
H0MAX = 1

l2norm = {}
euler_data = {}
rk2_data = {}
pc_data = {}

target_rmsef = tc.rmsef
tf = tc.tf
```

```
t0 = tc.t0
omega = tc.omega
y0 = tc.y0
v0 = tc.v0
h0 = H0MAX
iteration = 0

for i in range(17): # Configure iterations
    h = h0
    euler_rmse = 1
    rk2_rmse = 1
    pc_rmse = 1
    euler_operations = 0
    rk2_operations = 0
    pc_operations = 0
    iteration += 1

    while euler_rmse > target_rmsef:
        N = int((tf - t0) / h)
        t_values = np.linspace(t0, tf, N)
        v_prev = None
        euler_y = y0
        euler_v = v0

        euler_errors = []
        euler_y_values = []
        euler_v_values = []
        analytical_values = []

        for t in t_values:
            y_analytical = tc.analytical_solution(t, y0, v0, omega)
            analytical_values.append(y_analytical)

             euler_y, euler_v, v_predict = tc1methods.pc(tc,euler_y, euler_v, h, omega, v_prev) #
Configure Method
            euler_y_values.append(euler_y)
            euler_v_values.append(euler_v)
            euler_operations += 2
            euler_error = abs(euler_y - y_analytical)
```

```
            euler_errors.append(euler_error)
            print(i, h, t)

        euler_rmse = np.sqrt(np.mean(np.array(euler_errors)**2))

        h = h / 2

    h = h0

    while rk2_rmse > target_rmsef:
        N = int((tf - t0) / h)
        t_values = np.linspace(t0, tf, N)
        v_prev = None
        rk2_y = y0
        rk2_v = v0
        pc_v = v0

        rk2_errors = []
        rk2_y_values = []
        rk2_v_values = []
        analytical_values = []

        for t in t_values:
            y_analytical = tc.analytical_solution(t, y0, v0, omega)
            analytical_values.append(y_analytical)

           rk2_y, rk2_v, v_predict = tc1methods.rk2(tc, rk2_y, rk2_v, h, omega, v_prev) # Configure
Method
            rk2_y_values.append(rk2_y)
            rk2_v_values.append(rk2_v)
            rk2_operations += 8
            rk2_error = abs(rk2_y - y_analytical)
            rk2_errors.append(rk2_error)
            print(i, h, t)

        rk2_rmse = np.sqrt(np.mean(np.array(rk2_errors)**2))

        h = h / 2
```

```
    h = h0

    while pc_rmse > target_rmsef:
        N = int((tf - t0) / h)
        t_values = np.linspace(t0, tf, N)
        v_prev = None
        pc_y = y0
        pc_v = v0

        pc_errors = []
        pc_y_values = []
        pc_v_values = []
        analytical_values = []

        for t in t_values:
            y_analytical = tc.analytical_solution(t, y0, v0, omega)
            analytical_values.append(y_analytical)

             pc_y, pc_v, v_predict = tc1methods.pc(tc, pc_y, pc_v, h, omega, v_prev) # Configure
Method
            v_prev = pc_v
            pc_y_values.append(pc_y)
            pc_v_values.append(pc_v)
            pc_operations += 4
            pc_error = abs(pc_y - y_analytical)
            pc_errors.append(pc_error)
            print(i, h, t)

        pc_rmse = np.sqrt(np.mean(np.array(pc_errors)**2))

        h = h / 2

    euler_data[(iteration - 1)] = [euler_rmse, euler_operations]
    rk2_data[(iteration - 1)] = [rk2_rmse, rk2_operations]
    pc_data[(iteration - 1)] = [pc_rmse, pc_operations]
    h0 = h0 / 2

print("done.")
```

```python
def plotops(method_data):
    # Assuming 'data' is already defined
    xval, yval = zip(*sorted(method_data.items()))
    rmseplot = []
    opsplot = []
    ratioplot = []

    for item in yval:
        rmseplot.append(item[0])  # Root Mean Square Error values
        opsplot.append(item[1])  # Runtime values
        ratioplot.append(item[0] * item[1])

    X = np.arange(len(method_data))

    # Standardize font sizes
    plt.rcParams.update({'font.size': 12})

    # Define consistent limits and buffers for the plots
    min_operations, max_operations = 0, max(opsplot) * 1.1
    min_rmse, max_rmse = 0, max(rmseplot) * 1.1
    min_ratio, max_ratio = 0, max(ratioplot) * 1.1

    # Define the number of x ticks and create a consistent set of labels
    x_ticks = np.arange(len(method_data))

    # Set a consistent color scheme
    color_operations = 'blue'
    color_ratio = 'red'



    # Create the plot
    fig, ax1 = plt.subplots(figsize=(8, 4))

    # Operations plot
    ax1.bar(x_ticks, opsplot, color=color_operations, label='Operations', width=0.4)
    ax1.set_ylabel('Operations', color=color_operations)
    ax1.tick_params(axis='y', labelcolor=color_operations)
    ax1.set_ylim([min_operations, max_operations])
    ax1.set_xticks(x_ticks)
```

```
ax1.set_xticklabels(X, rotation=90)
ax1.set_xlabel('$h_0$ order')

# Ratio plot
ax2 = ax1.twinx()
ax2.plot(x_ticks, rmseplot, color=color_ratio, label='RMSE', linestyle='-', marker='o')
ax2.set_ylabel('RMSE', color=color_ratio)
ax2.tick_params(axis='y', labelcolor=color_ratio)
ax2.set_ylim([min_rmse, max_rmse])

plt.xlabel('$h_0$ Order')

# Ensure the display is not too tight
plt.tight_layout()

# Save the figure
plt.savefig('eff_plot.png', bbox_inches='tight')



# Create the plot
fig, ax1 = plt.subplots(figsize=(8, 4))

# Operations plot
ax1.bar(x_ticks, opsplot, color=color_operations, label='Operations', width=0.4)
ax1.set_ylabel('Operations', color=color_operations)
ax1.tick_params(axis='y', labelcolor=color_operations)
ax1.set_ylim([min_operations, max_operations])
ax1.set_xticks(x_ticks)
ax1.set_xticklabels(X, rotation=90)
ax1.set_xlabel('$h_0$ order')

# Ratio plot
ax2 = ax1.twinx()
ax2.plot(x_ticks, ratioplot, color=color_ratio, label='Accuracy', linestyle='-', marker='o')
ax2.set_ylabel('Accuracy', color=color_ratio)
ax2.tick_params(axis='y', labelcolor=color_ratio)
ax2.set_ylim([min_ratio, max_ratio])

plt.xlabel('$h_0$ Order')
```

```
    # Ensure the display is not too tight
    plt.tight_layout()

    # Save the figure
    plt.savefig('eff_plot.png', bbox_inches='tight')

def combined(d1, d2, d3):
    x1val, y1val = zip(*sorted(d1.items()))
    x2val, y2val = zip(*sorted(d2.items()))
    x3val, y3val = zip(*sorted(d3.items()))

    opsplot1 = []
    opsplot2 = []
    opsplot3 = []

    accplot1 = []
    accplot2 = []
    accplot3 = []

    for item in y1val:
        opsplot1.append(item[1])  # Runtime values
        accplot1.append(item[0] * item[1])
    for item in y2val:
        opsplot2.append(item[1])  # Runtime values
        accplot2.append(item[0] * item[1])
    for item in y3val:
        opsplot3.append(item[1])  # Runtime values
        accplot3.append(item[0] * item[1])

    # Standardize font sizes
    plt.rcParams.update({'font.size': 12})

    # Define consistent limits and buffers for the plots
    min_operations, max_operations = 0, max(opsplot1 + opsplot2 + opsplot3) * 1.1
    min_acc, max_acc = 0, max(accplot1 + accplot2 + accplot3) * 1.1

    # Define the number of x ticks and create a consistent set of labels
    x_ticks = np.arange(len(d1))
```

```
# Create the plot
fig, ax = plt.subplots(figsize=(8, 4))

ax.plot(x_ticks, opsplot1, color='k', label='Euler', linestyle='-')
ax.plot(x_ticks, opsplot2, color='red', label='RK2', linestyle='-')
ax.plot(x_ticks, opsplot3, color='blue', label='P-C', linestyle='-')
ax.set_ylabel('Operations')
ax.tick_params(axis='y')
ax.set_ylim([min_operations, max_operations])

# Add combined legend
fig.legend()

plt.xlabel('$h_0$ Order')

# Ensure the display is not too tight
plt.tight_layout()

# Save the figure
plt.savefig('eff_plot.png')

# Create the plot
fig, ax = plt.subplots(figsize=(8, 4))

ax.plot(x_ticks, accplot1, color='k', label='Euler', linestyle='-')
ax.plot(x_ticks, accplot2, color='red', label='RK2', linestyle='-')
ax.plot(x_ticks, accplot3, color='blue', label='P-C', linestyle='-')
ax.set_ylabel('Accuracy')
ax.tick_params(axis='y')
ax.set_ylim([min_acc, max_acc])

# Add combined legend
fig.legend()

plt.xlabel('$h_0$ Order')

# Ensure the display is not too tight
plt.tight_layout()
```

```
    # Save the figure
    plt.savefig('e_plot.png')

plotops(euler_data)
plotops(rk2_data)
plotops(pc_data)

combined(euler_data, rk2_data, pc_data)

# Show the plot
plt.show()
```

C. Test Case 1: Methods

```
def rk2(tc, y, v, h, omega, v_prev):
    k1_v = tc.function(y, omega)
    k1_y = v
    midpoint_v = v + 0.5 * h * k1_v
    midpoint_y = y + 0.5 * h * k1_y
    k2_v = tc.function(midpoint_y, omega)
    k2_y = midpoint_v
    y_next = y + h * k2_y
    v_next = v + h * k2_v
    return y_next, v_next, None

def euler(tc, y, v, h, omega, v_prev):
    v_next = v + h * tc.function(y, omega)
    y_next = y + h * v
    return y_next, v_next, None

def pc(tc, y, v, h, omega, v_prev):
    # Predictor (Euler method)
    v_predict = v + h * tc.function(y, omega)
    y_predict = y + h * v

    # Corrector (Adams-Bashforth method)
    if v_prev is not None:
        v_correct = v + h * (1.5 * tc.function(y_predict, omega) - 0.5 * tc.function(y, omega))
    else:
        v_correct = v_predict
    y_correct = y + 0.5 * h * (v + v_correct)

    return y_correct, v_correct, v_predict
```

D. Test Case 2: Convergence

```
import numpy as np
import matplotlib.pyplot as plt
import scripts.tc2methods as tc2methods

plt.rc('text', usetex=True)
plt.rc('font', family='serif')

# Damped SHM
class IVP2:
    def __init__(self, b, k, y0, v0, t0, tf, rmsef) -> None:
        self.b = b # Damping coefficient
        self.k = k # Stiffness coefficient
        self.y0 = y0 # Initial position
        self.v0 = v0 # Initial velocity
        self.t0 = t0 # Initial time
        self.tf = tf # Final time
        self.rmsef = rmsef # Target L2 norm

    def function(self, y, v, b, k):
        return -b * v - k * y

    def analytical_solution(self, t, y0, v0, b, k):
        discriminant = b**2 - 4*k
        if discriminant >= 0:  # Overdamped or critically damped
            r1 = (-b + np.sqrt(discriminant)) / 2
            r2 = (-b - np.sqrt(discriminant)) / 2
            A = (v0 - r2 * y0) / (r1 - r2)
            B = y0 - A
            return A * np.exp(r1 * t) + B * np.exp(r2 * t)
        else:  # Underdamped
            alpha = -b / 2
            omega = np.sqrt(-discriminant) / 2
            A = y0
            B = (v0 - alpha * y0) / omega
            return np.exp(alpha * t) * (A * np.cos(omega * t) + B * np.sin(omega * t))

IVP2TC1 = IVP2(10, 100, 1, 0, 0, 5, 0.001)
tc = IVP2TC1
```

```
# Configure starting step size
H0MAX = 1

l2norm = {}
data = {}

target_rmsef = tc.rmsef
tf = tc.tf
t0 = tc.t0
b = tc.b
k = tc.k
y0 = tc.y0
v0 = tc.v0

print(tc.tf)
print(tc.rmsef)

h = H0MAX
rmse = 1

step = 0
plt.figure(figsize=(8, 4))

# Convergence Step
while rmse > target_rmsef:
    N = int((tf - t0) / h)
    t_values = np.linspace(t0, tf, N)
    v_prev = None
    step += 1
    # Initial values
    y = y0
    v = v0

    errors = []
    y_values = []
    v_values = []
    analytical_values = []

    # Euler Method
```

```
    for t in t_values:
        y, v, v_predict = tc2methods.pc(tc, y, v, h, b, k, v_prev) # Configure Method
        v_prev = v

        y_values.append(y)
        v_values.append(v)
        y_analytical = tc.analytical_solution(t, y0, v0, b, k)

        analytical_values.append(y_analytical)
        error = abs(y - y_analytical)
        errors.append(error)

    rmse = np.sqrt(np.mean(np.array(errors)**2))
    l2norm[h] = rmse
    print(h)
    print(rmse)
    print("--------")
    h = h / 2

    # Plot the errors for the current step size
    plt.plot(t_values, errors, label=f'$dt=2^{{-{step}}}$')

# Finalize the plot
plt.xlabel('time / s')
plt.ylabel('$\Delta x$')
plt.yscale('log')
plt.legend()
plt.legend(loc='upper left', bbox_to_anchor=(1, 1))
plt.tick_params(axis='x', which='minor', bottom=False)
plt.tick_params(axis='y', which='minor', left=False)
plt.tight_layout()

plt.savefig('convergence.png', bbox_inches='tight')



# Plot Convergence Rate Graph
xval, yval = zip(*sorted(l2norm.items()))
```

```
plt.figure(figsize=(4, 4))
plt.loglog()
plt.scatter(xval, yval)
plt.xlabel('d$t$')
plt.ylabel('RMSE')
plt.title('P-C') # Configure Method Name
plt.tick_params(axis='x', which='minor', bottom=False)
plt.tick_params(axis='y', which='minor', left=False)
plt.gca().invert_xaxis()

h_threshold = 10**(-1)

stable_xval = [x for x in xval if x < h_threshold]
stable_yval = [y for x, y in zip(xval, yval) if x < h_threshold]

m, c = np.polyfit(np.log(stable_xval), np.log(stable_yval), 1)
y_fit = np.exp(m*np.log(xval) + c)

plt.plot(xval, y_fit, label=f"$y = e^{{{m:.4f}\\log(x) + {c:.4f}}}$")
plt.tight_layout()
plt.legend()

print(f"y = e^({m:.4f}*log(x) + {c:.4f}")

plt.savefig('convergencerate.png', bbox_inches='tight')

# Show the plot
plt.show()
```

E. Test Case 2: Efficiency

```
import numpy as np
import matplotlib.pyplot as plt
import tc2methods

plt.rc('text', usetex=True)
plt.rc('font', family='serif')

# Damped SHM
class IVP2:
    def __init__(self, b, k, y0, v0, t0, tf, rmsef) -> None:
        self.b = b # Damping coefficient
        self.k = k # Stiffness coefficient
        self.y0 = y0 # Initial position
        self.v0 = v0 # Initial velocity
        self.t0 = t0 # Initial time
        self.tf = tf # Final time
        self.rmsef = rmsef # Target L2 norm

    def function(self, y, v, b, k):
        return -b * v - k * y

    def analytical_solution(self, t, y0, v0, b, k):
        discriminant = b**2 - 4*k
        if discriminant >= 0:  # Overdamped or critically damped
            r1 = (-b + np.sqrt(discriminant)) / 2
            r2 = (-b - np.sqrt(discriminant)) / 2
            A = (v0 - r2 * y0) / (r1 - r2)
            B = y0 - A
            return A * np.exp(r1 * t) + B * np.exp(r2 * t)
        else:  # Underdamped
            alpha = -b / 2
            omega = np.sqrt(-discriminant) / 2
            A = y0
            B = (v0 - alpha * y0) / omega
            return np.exp(alpha * t) * (A * np.cos(omega * t) + B * np.sin(omega * t))

IVP2TC1 = IVP2(10, 100, 1, 0, 0, 5, 0.001)
tc = IVP2TC1
```

```
# Configure starting step size
H0MAX = 1

l2norm = {}
euler_data = {}
rk2_data = {}
pc_data = {}

target_rmsef = tc.rmsef
tf = tc.tf
t0 = tc.t0
b = tc.b
k = tc.k
y0 = tc.y0
v0 = tc.v0
h0 = H0MAX
iteration = 0

for i in range(14): # Configure iterations
    h = h0
    euler_rmse = 1
    rk2_rmse = 1
    pc_rmse = 1
    euler_operations = 0
    rk2_operations = 0
    pc_operations = 0
    iteration += 1

    while euler_rmse > target_rmsef:
        N = int((tf - t0) / h)
        t_values = np.linspace(t0, tf, N)
        v_prev = None
        euler_y = y0
        euler_v = v0

        euler_errors = []
        euler_y_values = []
        euler_v_values = []
        analytical_values = []
```

```
    for t in t_values:
        y_analytical = tc.analytical_solution(t, y0, v0, b, k)
        analytical_values.append(y_analytical)

        euler_y, euler_v, v_predict = tc2methods.euler(tc, euler_y, euler_v, h, b, k, v_prev)
        euler_y_values.append(euler_y)
        euler_v_values.append(euler_v)
        euler_operations += 2
        euler_error = abs(euler_y - y_analytical)
        euler_errors.append(euler_error)
        print(i, h, t)

    euler_rmse = np.sqrt(np.mean(np.array(euler_errors)**2))

    h = h / 2

h = h0

while rk2_rmse > target_rmsef:
    N = int((tf - t0) / h)
    t_values = np.linspace(t0, tf, N)
    v_prev = None
    rk2_y = y0
    rk2_v = v0
    pc_v = v0

    rk2_errors = []
    rk2_y_values = []
    rk2_v_values = []
    analytical_values = []

    for t in t_values:
        y_analytical = tc.analytical_solution(t, y0, v0, b, k)
        analytical_values.append(y_analytical)

        rk2_y, rk2_v, v_predict = tc2methods.rk2(tc, rk2_y, rk2_v, h, b, k, v_prev)
        rk2_y_values.append(rk2_y)
        rk2_v_values.append(rk2_v)
```

```
            rk2_operations += 8
            rk2_error = abs(rk2_y - y_analytical)
            rk2_errors.append(rk2_error)
            print(i, h, t)

        rk2_rmse = np.sqrt(np.mean(np.array(rk2_errors)**2))

        h = h / 2

    h = h0

    while pc_rmse > target_rmsef:
        N = int((tf - t0) / h)
        t_values = np.linspace(t0, tf, N)
        v_prev = None
        pc_y = y0
        pc_v = v0

        pc_errors = []
        pc_y_values = []
        pc_v_values = []
        analytical_values = []

        for t in t_values:
            y_analytical = tc.analytical_solution(t, y0, v0, b, k)
            analytical_values.append(y_analytical)

            pc_y, pc_v, v_predict = tc2methods.pc(tc, pc_y, pc_v, h, b, k, v_prev)
            v_prev = pc_v
            pc_y_values.append(pc_y)
            pc_v_values.append(pc_v)
            pc_operations += 4
            pc_error = abs(pc_y - y_analytical)
            pc_errors.append(pc_error)
            print(i, h, t)

        pc_rmse = np.sqrt(np.mean(np.array(pc_errors)**2))

        h = h / 2
```

```
    euler_data[(iteration - 1)] = [euler_rmse, euler_operations]
    rk2_data[(iteration - 1)] = [rk2_rmse, rk2_operations]
    pc_data[(iteration - 1)] = [pc_rmse, pc_operations]
    h0 = h0 / 2

print("done.")

def plotops(method_data):
    # Assuming 'data' is already defined
    xval, yval = zip(*sorted(method_data.items()))
    rmseplot = []
    opsplot = []
    ratioplot = []

    for item in yval:
        rmseplot.append(item[0])  # Root Mean Square Error values
        opsplot.append(item[1])  # Runtime values
        ratioplot.append(item[0] * item[1])

    X = np.arange(len(method_data))

    # Standardize font sizes
    plt.rcParams.update({'font.size': 12})

    # Define consistent limits and buffers for the plots
    min_operations, max_operations = 0, max(opsplot) * 1.1
    min_rmse, max_rmse = 0, max(rmseplot) * 1.1
    min_ratio, max_ratio = 0, max(ratioplot) * 1.1

    # Define the number of x ticks and create a consistent set of labels
    x_ticks = np.arange(len(method_data))

    # Set a consistent color scheme
    color_operations = 'blue'
    color_ratio = 'red'


    # Create the plot
```

```
fig, ax1 = plt.subplots(figsize=(8, 4))

# Operations plot
ax1.bar(x_ticks, opsplot, color=color_operations, label='Operations', width=0.4)
ax1.set_ylabel('Operations', color=color_operations)
ax1.tick_params(axis='y', labelcolor=color_operations)
ax1.set_ylim([min_operations, max_operations])
ax1.set_xticks(x_ticks)
ax1.set_xticklabels(X, rotation=90)
ax1.set_xlabel('$h_0$ order')

# Ratio plot
ax2 = ax1.twinx()
ax2.plot(x_ticks, rmseplot, color=color_ratio, label='RMSE', linestyle='-', marker='o')
ax2.set_ylabel('RMSE', color=color_ratio)
ax2.tick_params(axis='y', labelcolor=color_ratio)
ax2.set_ylim([min_rmse, max_rmse])

plt.xlabel('$h_0$ Order')

# Ensure the display is not too tight
plt.tight_layout()

# Save the figure
plt.savefig('eff_plot.png', bbox_inches='tight')



# Create the plot
fig, ax1 = plt.subplots(figsize=(8, 4))

# Operations plot
ax1.bar(x_ticks, opsplot, color=color_operations, label='Operations', width=0.4)
ax1.set_ylabel('Operations', color=color_operations)
ax1.tick_params(axis='y', labelcolor=color_operations)
ax1.set_ylim([min_operations, max_operations])
ax1.set_xticks(x_ticks)
ax1.set_xticklabels(X, rotation=90)
ax1.set_xlabel('$h_0$ order')
```

```
    # Ratio plot
    ax2 = ax1.twinx()
    ax2.plot(x_ticks, ratioplot, color=color_ratio, label='Accuracy', linestyle='-', marker='o')
    ax2.set_ylabel('Accuracy', color=color_ratio)
    ax2.tick_params(axis='y', labelcolor=color_ratio)
    ax2.set_ylim([min_ratio, max_ratio])

    plt.xlabel('$h_0$ Order')

    # Ensure the display is not too tight
    plt.tight_layout()

    # Save the figure
    plt.savefig('eff_plot.png', bbox_inches='tight')

def combined(d1, d2, d3):
    x1val, y1val = zip(*sorted(d1.items()))
    x2val, y2val = zip(*sorted(d2.items()))
    x3val, y3val = zip(*sorted(d3.items()))

    opsplot1 = []
    opsplot2 = []
    opsplot3 = []

    accplot1 = []
    accplot2 = []
    accplot3 = []

    for item in y1val:
        opsplot1.append(item[1])  # Runtime values
        accplot1.append(item[0] * item[1])
    for item in y2val:
        opsplot2.append(item[1])  # Runtime values
        accplot2.append(item[0] * item[1])
    for item in y3val:
        opsplot3.append(item[1])  # Runtime values
        accplot3.append(item[0] * item[1])

    # Standardize font sizes
```

```
plt.rcParams.update({'font.size': 12})

# Define consistent limits and buffers for the plots
min_operations, max_operations = 0, max(opsplot1 + opsplot2 + opsplot3) * 1.1
min_acc, max_acc = 0, max(accplot1 + accplot2 + accplot3) * 1.1

# Define the number of x ticks and create a consistent set of labels
x_ticks = np.arange(len(d1))

# Create the plot
fig, ax = plt.subplots(figsize=(8, 4))

ax.plot(x_ticks, opsplot1, color='k', label='Euler', linestyle='-')
ax.plot(x_ticks, opsplot2, color='red', label='RK2', linestyle='-')
ax.plot(x_ticks, opsplot3, color='blue', label='P-C', linestyle='-')
ax.set_ylabel('Operations')
ax.tick_params(axis='y')
ax.set_ylim([min_operations, max_operations])

# Add combined legend
fig.legend()

plt.xlabel('$h_0$ Order')

# Ensure the display is not too tight
plt.tight_layout()

# Save the figure
plt.savefig('eff_plot.png')

# Create the plot
fig, ax = plt.subplots(figsize=(8, 4))

ax.plot(x_ticks, accplot1, color='k', label='Euler', linestyle='-')
ax.plot(x_ticks, accplot2, color='red', label='RK2', linestyle='-')
ax.plot(x_ticks, accplot3, color='blue', label='P-C', linestyle='-')
ax.set_ylabel('Accuracy')
ax.tick_params(axis='y')
ax.set_ylim([min_acc, max_acc])
```

```
    # Add combined legend
    fig.legend()

    plt.xlabel('$h_0$ Order')

    # Ensure the display is not too tight
    plt.tight_layout()

    # Save the figure
    plt.savefig('e_plot.png')

plotops(euler_data)
plotops(rk2_data)
plotops(pc_data)

combined(euler_data, rk2_data, pc_data)

# Show the plot
plt.show()
```

F. Test Case 2: Methods

```
def rk2(tc, y, v, h, b, k, v_prev):
    k1_v = tc.function(y, v, b, k)
    k1_y = v
    midpoint_v = v + 0.5 * h * k1_v
    midpoint_y = y + 0.5 * h * k1_y
    k2_v = tc.function(midpoint_y, midpoint_v, b, k)
    k2_y = midpoint_v
    y_next = y + h * k2_y
    v_next = v + h * k2_v
    return y_next, v_next, None

def euler(tc, y, v, h, b, k, v_prev):
    v_next = v + h * tc.function(y, v, b, k)
    y_next = y + h * v
    return y_next, v_next, None

def pc(tc, y, v, h, b, k, v_prev):
    # Predictor (Euler method)
    v_predict = v + h * tc.function(y, v, b, k)
    y_predict = y + h * v

    # Corrector (Adams-Bashforth method)
    if v_prev is not None:
        v_correct = v + h * (1.5 * tc.function(y_predict, v_predict, b, k) - 0.5 * tc.function(y,
v_prev, b, k))
    else:
        # For the first step, fall back to Euler method as we don't have v_prev
        v_correct = v_predict
    y_correct = y + 0.5 * h * (v + v_correct)

    return y_correct, v_correct, v_predict
```

G. Convergence Graph

```
import matplotlib.pyplot as plt
import numpy as np

plt.rc('text', usetex=True)
plt.rc('font', family='serif')

plt.figure(figsize=(6, 4))
plt.loglog()
plt.xlabel('d$t$')
plt.ylabel('RMSE')
plt.gca().invert_xaxis()
plt.tick_params(axis='x', which='minor', bottom=False)
plt.tick_params(axis='y', which='minor', left=False)

x = np.logspace(-4.5, 0, 100)  # For dt ranging from 0.0001 to 1

euler1_y_fit = np.exp(1.0662*np.log(x) + 2.8698)
euler2_y_fit = np.exp(1.0592*np.log(x) + 0.7415)
rk21_y_fit = np.exp(1.0511*np.log(x) + 0.5907)
rk22_y_fit = np.exp(1.0299*np.log(x) + 0.1352)
pc1_y_fit = np.exp(0.9457*np.log(x) + 1.9084)
pc2_y_fit = np.exp(1.2317*np.log(x) + 1.0899)

plt.plot(x, euler1_y_fit, color="k", label="Euler (Test Case 1)")
plt.plot(x, euler2_y_fit, linestyle="dashed", color="k", label="Euler Test Case 2")
plt.plot(x, rk21_y_fit, color="red", label="RK2 Test Case 1")
plt.plot(x, rk22_y_fit, linestyle="dashed", color="red", label="RK2 Test Case 2")
plt.plot(x, pc1_y_fit, color="blue", label="P-C Test Case 1")
plt.plot(x, pc2_y_fit, linestyle="dashed", color="blue", label="P-C Test Case 2")

target_rmse = 0.001
plt.axhline(y=target_rmse, color='grey', linestyle='--', linewidth=1)

methods_y_fits = [euler1_y_fit, euler2_y_fit, rk21_y_fit, rk22_y_fit, pc1_y_fit, pc2_y_fit]
colors = ['k', 'k', 'red', 'red', 'blue', 'blue']
linestyles = ['solid', 'dashed', 'solid', 'dashed', 'solid', 'dashed']

for y_fit, color, linestyle in zip(methods_y_fits, colors, linestyles):
    intersect_dt = x[np.argmin(np.abs(y_fit - target_rmse))]
```

```
    intersect_rmse = y_fit[np.argmin(np.abs(y_fit - target_rmse))]
    plt.plot(intersect_dt, intersect_rmse, 'o', color=color)

plt.tight_layout()
plt.ylim(0.0005, 50)
plt.legend()

plt.savefig('allconvergencerates.png', bbox_inches='tight')

plt.show
```

H. Composite Error Graph

```
import numpy as np
import matplotlib.pyplot as plt

def f(y, omega):
    return -omega**2 * y

def rk2(y, v, h, omega, v_prev):
    k1_v = f(y, omega)
    k1_y = v
    midpoint_v = v + 0.5 * h * k1_v
    midpoint_y = y + 0.5 * h * k1_y
    k2_v = f(midpoint_y, omega)
    k2_y = midpoint_v
    y_next = y + h * k2_y
    v_next = v + h * k2_v
    return y_next, v_next

def euler(y, v, h, omega, v_prev):
    v_next = v + h * f(y, omega)
    y_next = y + h * v
    return y_next, v_next

def pc(y, v, h, omega, v_prev):
    # Predictor (Euler method)
    v_predict = v + h * f(y, omega)
    y_predict = y + h * v

    # Corrector (Adams-Bashforth method)
    if v_prev is not None:
        v_correct = v + h * (1.5 * f(y_predict, omega) - 0.5 * f(y, omega))
    else:
        v_correct = v_predict
    y_correct = y + 0.5 * h * (v + v_correct)

    return y_correct, v_correct, v_predict

def analytical_solution(t, y0, v0, omega):
    A = y0
    B = v0 / omega
```

```
    return A * np.cos(omega * t) + B * np.sin(omega * t)

# Parameters and initial conditions
omega = np.sqrt(10)  # Adjust as needed
y0 = 1
v0 = 0
t0 = 0
tf = 5
h = 1/(2**6)
N = int((tf - t0) / h)

# Arrays for storing values
t_values = np.linspace(t0, tf, N)
analytical_values = []

euler_y_values = []
euler_v_values = []
euler_errors = []

rk2_y_values = []
rk2_v_values = []
rk2_errors = []

pc_y_values = []
pc_v_values = []
pc_errors = []

# Initial values
euler_y = y0
euler_v = v0

rk2_y = y0
rk2_v = v0

pc_y = y0
pc_v = v0

plt.rc('text', usetex=True)
plt.rc('font', family='serif')
```

```
plt.figure(figsize=(8, 4))
ax = plt.gca()

v_prev = None
# Euler Method
for t in t_values:
    y_analytical = analytical_solution(t, y0, v0, omega)
    analytical_values.append(y_analytical)

    euler_y, euler_v = euler(euler_y, euler_v, h, omega, v_prev)
    euler_y_values.append(euler_y)
    euler_v_values.append(euler_v)
    abserr = abs(euler_y - y_analytical)
    euler_errors.append(abserr)

    rk2_y, rk2_v = rk2(rk2_y, rk2_v, h, omega, v_prev)
    rk2_y_values.append(rk2_y)
    rk2_v_values.append(rk2_v)
    abserr = abs(rk2_y - y_analytical)
    rk2_errors.append(abserr)

    pc_y, pc_v, v_predict = pc(pc_y, pc_v, h, omega, v_prev)
    pc_y_values.append(pc_y)
    pc_v_values.append(pc_v)

    abserr = abs(pc_y - y_analytical)
    pc_errors.append(abserr)

    v_prev = pc_v

plt.plot(t_values, euler_errors, label='Euler', color="red")
plt.plot(t_values, pc_errors, label='P-C', color="blue")
plt.plot(t_values, rk2_errors, label='RK2', color="purple")

# Finalize the plot
plt.xlabel('time / s')
plt.ylabel('$\Delta x$')
plt.legend(loc='upper left', bbox_to_anchor=(1, 1))
plt.tight_layout()
```

```
plt.show()


# Calculate RMSE
euler_rmse = np.sqrt(np.mean(np.array(euler_errors)**2))
rk2_rmse = np.sqrt(np.mean(np.array(rk2_errors)**2))
pc_rmse = np.sqrt(np.mean(np.array(pc_errors)**2))
print("EULER RMSE:", euler_rmse)
print("  RK2 RMSE:", rk2_rmse)
print("   PC RMSE:", pc_rmse)
```

I. Convergence Rate Chart

```
import numpy as np
import matplotlib.pyplot as plt

plt.rc('text', usetex=True)
plt.rc('font', family='serif')

data = [[1.0662, 1.0511, 0.9457],
        [1.0592, 1.0299, 1.2317]]

X = np.arange(3)
fig = plt.figure(figsize=(4, 4))
ax = fig.add_axes([0,0,1,1])
ax.set_ylim(0, 2)
method = ['Euler', 'RK2', 'P-C']
ax.bar(X + 0.1, data[0], color = 'orange', width = 0.4)
ax.bar(X + 0.5, data[1], color = 'cornflowerblue', width = 0.4)



ax.set_ylabel('Convergence Rate')
ax.set_xlabel('Method')

plt.xticks(X + 0.3, method)

ax.legend(labels=['IVP1', 'IVP2'], title="Test Cases")
```