ZJU-UIUC INSTITUTE

SP25 MATH285 Lab Report

Group 10 Members:

Yuan Shihong	323011
Wang Chenyi	323011
Zhang Kerui	323011
Xia Wubing	323011
Li Binghe	323011

HONOR STATEMENT

We declare that this report is our own original work. For its preparation, we have not used any other resources than those cited as references. Every group member has a fair share in this work.

Sign:

Date: 2025.5.25

MATH285 Lab Report

Yuan Shihong , Li Binghe , Wang Chenyi, Zhang Kerui, Xia Wubing

June 2, 2025

Abstract

This lab report aims to investigate the maximal solutions and their domains for two initial value problems (IVPs) using both numerical and analytical methods. The specific problems are as follows:

$$\begin{aligned} &\text{(IVP1)} \quad y' = t^3 + y^3, \quad y(0) = -1 \\ &\text{(IVP2)} \quad y' = t^3 + y^3, \quad y(0) = 0 \\ &\text{(IVP3)} \quad y' = t^3 + y^3, \quad y(0) = 1 \end{aligned}$$

1 Introduction

This experiment focuses on solving three initial value problems (IVPs) using numerical methods such as Euler, Improved Euler, and Runge-Kutta and Power Series.

Specifically, we determined the following for the solution's domain and behavior based on the initial condition y_0:

- When $y_0 = -1$, the function's domain extends from $-\infty$ to approximately **0.502**. The graph of the solution exhibits a shape similar to a "check mark" (or a hyperbola-like curve with two branches).
- When $y_0 = 0$, the function's domain extends from $-\infty$ to approximately **1.647**. The solution is strictly monotonically decreasing across its entire domain.
- When $y_0 = 1$, the function's domain extends from $-\infty$ to approximately **0.498**. Similar to the $y_0 = -1$ case, the solution's graph resembles a "check mark" shape.

2 Analysis Problem

2.1 Euler Method

2.1.1 Mathematical Analysis

The Euler method is a numerical technique for approximating solutions to $y' = f(t, y), y(t_0) = y_0$. Here, $f(t, y) = t^3 + y^3$, and the initial condition is $y(0) = y_0$. The method uses the iterative formula:

$$y_{n+1} = y_n + h * f(t_n, y_n)$$

$$t_{n+1} = t_n + h$$

where h is the step size, y_n is the numerical solution at t_n.

Forward in time (t increasing): $t_{n+1} = t_n + h$, $y_{n+1} = y_n + h(t_n^3 + y_n^3)$.

Backward in time (t decreasing): $t_{n+1} = t_n - h$, $y_{n+1} = y_n - h(t_n^3 + y_n^3)$.

The mathematical foundation of the Euler method derives from the Taylor series expansion of $y(t)$ at t_n:

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

Neglecting the higher-order term $O(h^2)$, the Euler method's approximation formula is obtained. The local truncation error is $O(h^2)$, and the global truncation error is $O(h)$.

The stability of the Euler method depends on the step size h and the properties of the differential equation. For stiff equations, the Euler method may require very small step sizes to remain stable, resulting in significantly increased computational cost.

2.1.2 Results Display

Since solutions can blow up (that is, $|y| \to \infty$), we iterate until $|y_n|$ exceeds a large threshold M (e.g., $M = 1000$), then estimate the blow-up time as the corresponding t_n.

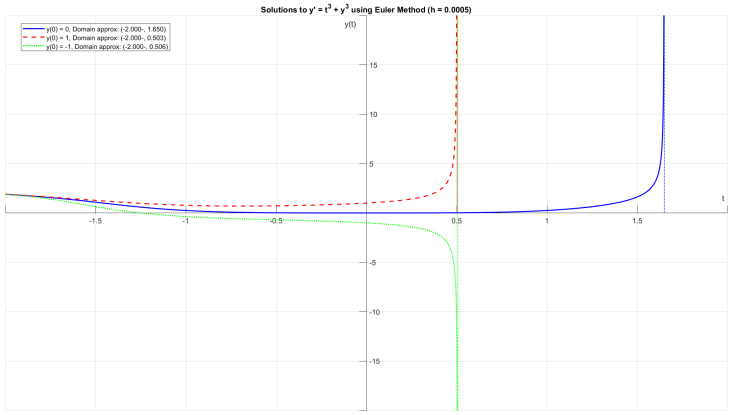


Figure 1: Euler Method

2.1.3 Error Analysis

A smaller step size h generally leads to a more accurate approximation of the solution, but requires more computational steps. The local truncation error (error in one step) for the Euler method is $O(h^2)$, and the global truncation error (total accumulated error) is $O(h)$.

$y_0 = 1$:

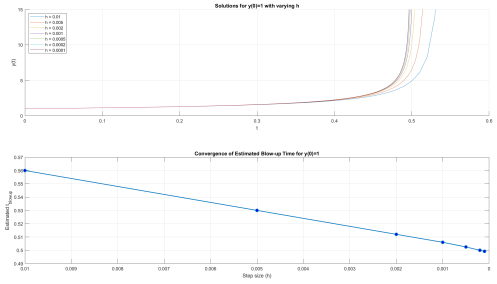


Figure 2: Estimated t_{blowup} vs. h

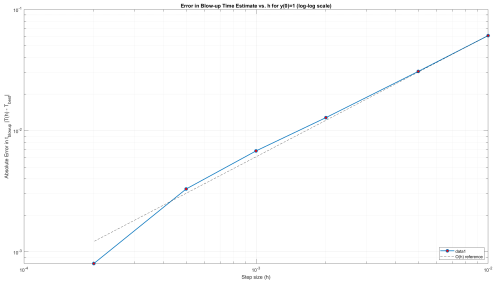


Figure 3: Error in t_{blowup} vs. h (Log-Log Scale)

$y_0 = $ -1 :

$y_0 = \mathbf{0}$:

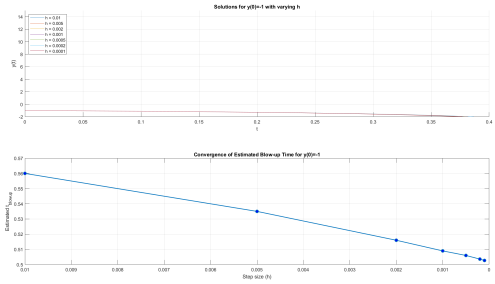


Figure 4: Estimated t_{blowup} vs. h

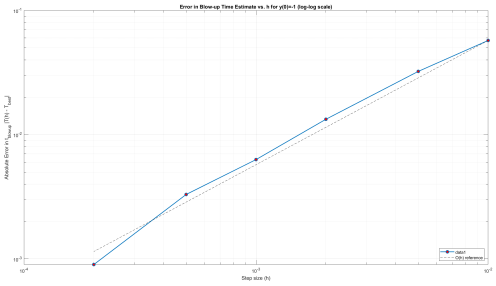


Figure 5: Error in t_{blowup} vs. h (Log-Log Scale)

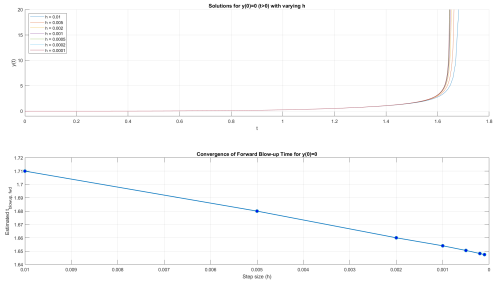


Figure 6: Estimated t_{blowup} vs. h

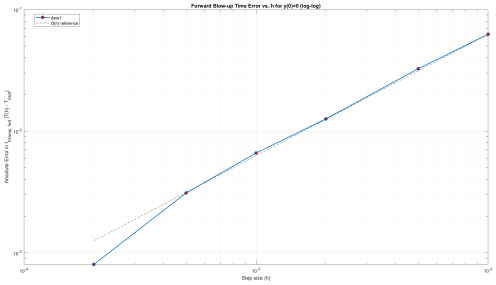


Figure 7: Error in t_{blowup} vs. h (Log-Log Scale)

2.2 Improved Euler Method

2.2.1 Mathematical Analysis

The improved Euler method (also known as the Heun method or trapezoidal method) enhances accuracy by incorporating the trapezoidal integration formula. The basic idea is to use the average slope between the current point and the predicted point to estimate the next point's value. For a given ODE $y' = f(t, y)$ with initial condition $y(t_0) = y_0$, the improved Euler method's recurrence formula is:

$$y_{n+1} = y_n + \frac{h}{2}\left[f(t_n, y_n) + f(t_{n+1}, y_{n+1}^{\text{predict}})\right]$$

where:

- $y_{n+1}^{\text{predict}} = y_n + hf(t_n, y_n)$ is the Euler method prediction.

The improved Euler method can be viewed as applying the trapezoidal rule to the integration problem. By considering the average slope at two adjacent points, the local truncation error is $O(h^3)$, and the global truncation error is $O(h^2)$.

2.2.2 Result Presentation

2.2.3 Error Analysis

The improved Euler method is a second-order method with a local truncation error of $O(h^3)$ and a global truncation error of $O(h^2)$. Compared to the Euler method, the improved Euler method is more stable when dealing with nonlinear and stiff problems. In this study, the improved Euler method shows higher accuracy and stability at the same step size, better capturing the solution's behavior.

For **IVP1** ($y' = t^3 + y^3,\ y(0) = -1$), the solution curve is relatively flat and stable, with no explosive behavior in the visualized domain. The improved Euler method provides a smooth, gradually increasing curve that closely follows the expected trend, indicating accurate approximation in this region.

For **IVP2** ($y' = t^3 + y^3,\ y(0) = 0$), the solution remains regular but begins to exhibit accelerated growth as t increases. The improved Euler method remains stable under moderate step sizes, though minor divergence from the expected curve begins to appear as y^3 dominates.

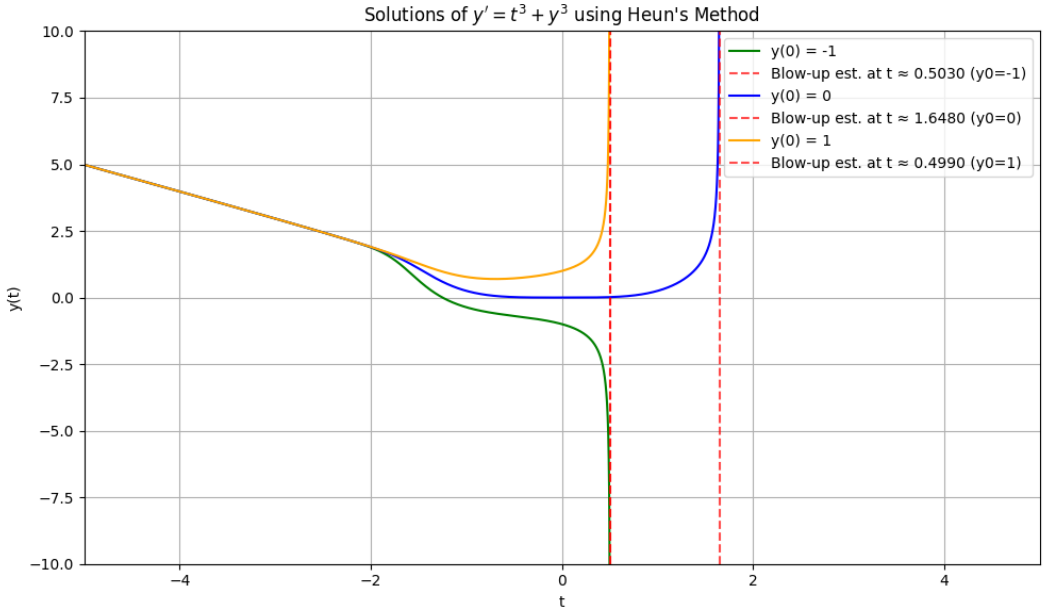


Figure 8: Improved Euler Method

For **IVP3** ($y' = t^3 + y^3,\ y(0) = 1$), the solution exhibits rapid blow-up behavior. The improved Euler method is still able to track the solution before the blow-up time, though numerical instability may eventually occur if the step size is not sufficiently small. Compared to the standard Euler method, this approach offers improved robustness even in stiff or rapidly growing regimes.

2.2.4 Conclusion

Compared to the classical Euler method, the improved Euler method provides higher accuracy and better stability when applied to the nonlinear equation $y' = t^3 + y^3$ under different initial conditions. It is especially advantageous in handling rapidly changing or stiff-like behavior such as in IVP3.

The comparative analysis of IVP1–IVP3 demonstrates that the improved Euler method performs well across all three scenarios, particularly by suppressing error growth in IVP2 and better tracking pre-blow-up behavior in IVP3. This confirms its effectiveness in approximating solutions to nonlinear problems.

While the improved Euler method requires slightly more computation per step than the Euler method, its overall benefits in accuracy and stability make it a practical choice in solving ODEs with strong nonlinearities.

2.3 Picard-Lindelof Iteration

2.3.1 Mathmatical Analysis

Picard-Lindelof Iteration provides a way to approximate the initial value problem. In our problem, $f(t, y_n(t)) = t^3 + y_n^3(t)$, $y_0(t) = y(0) = y_0$ To ensure the applicability of the Picard-Lindelof iteration method, we need to check whether the function f satisfies two key conditions within a rectangular region.

- Continuity of f: The function $f(t, y)$ is a polynomial function that means f is continuous on R^2.
- Lipschitz condition: If y_1 and y_2 are both in the interval $[y_0-b, y_0+b]$, $|y|^2 \leq (|y_0|+b)^2$ for any $y \in [y_0-b, y_0+b]$.

$$|f(t, y_1) - f(t, y_2)| = (t3 + y_1 3) - (t3 + y_2 3) = y_1 3 - y_2 3 = |(y_1 - y_2)(y_1 2 + y_1 y_2 + y_2^2)|$$

As $|(y_1^2 + y_1 y_2 + y_2^2)| \leq 3(|y_0| + b)^2$, we can choose $L = 3(|y_0| + b)^2$. Then $|f(t, y_1) - f(t, y_2)| \leq L|y_1 - y_2|$ is true in an rectangle region around (t_0, y_0).

Then, we can use the Picard-Lindelof iteration to get an approximation of the IVP.

$$y_{n+1}(t) = y_n(t) + \int_0^t f(s, y_n(s))ds = y_n(t) + \int_0^t (s^3 + y_n^3(t))ds$$

2.3.2 Results Display

- $y_0 = 0$:

$$
\begin{aligned}
&y_0(t) = 0 \\
&y_1(t) = y_0 + \int_0^t (s^3 + (y_0(s))^3)\, ds = 0 + \int_0^t (s^3 + 0^3)\, ds = \int_0^t s^3\, ds = \frac{t^4}{4} \\
&y_2(t) = y_0 + \int_0^t (s^3 + (y_1(s))^3)\, ds = 0 + \int_0^t \left(s^3 + \left(\frac{s^4}{4}\right)^3\right) ds = \int_0^t \left(s^3 + \frac{s^{12}}{64}\right) ds \\
&\quad \dots
\end{aligned}
$$

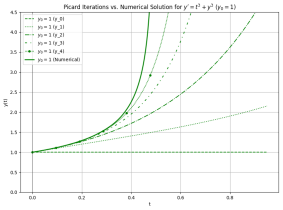


Figure 9: Estimated t_{blowup} vs. h

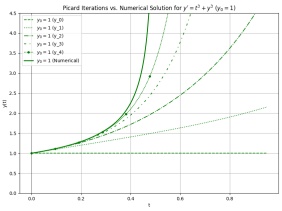


Figure 10: Estimated t_{blowup} vs. h

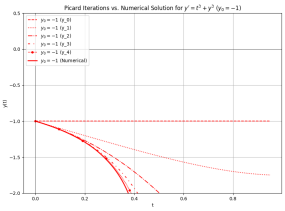


Figure 11: Error in t_{blowup} vs. h (Log-Log Scale)

2.3.3 Error Analysis

Picard-Lindelof theorem provides an error bound for n-th iterate $y_n(t)$.

$$y(t) - y_n(t) \leq \frac{ML^n|t - t_0|^{n+1}}{(n+1)!}$$

Where $M = \mathbf{max}|f(t, y)|$ in region $R = [t_0 - a, t_0 + a] \times [y_0 - b, y_0 + b]$, L is the constant in Lipschitz condition. In our situation, $M = a^3 + (y_0 + b)^3$, $L = 3(|y0| + b)^2$. Since the factorial function grows faster than the exponential function, the limit tends to zero.

2.4 Series method

2.4.1 Method

The series method is to use a power series as the analytic solution.

$$y(t) = \sum_{n=0}^{\infty} a_n (t - t_0)^n, \quad t \in I$$

Basically, there are two methods to obtain the best series. In first method, the solution will be obtained by the following steps.

First Method :

- Determine the n^{th} derivative of the given ODE.

$$
\begin{aligned}
y^{(1)} &= t^3 + y^3 \\
y^{(2)} &= 3t^2 + 3y^2 \cdot y^{(1)} = 3t^2 + 3y^5 + 3y^2 t^3 \\
y^{(3)} &= 6t + 3y^2 \cdot y^{(2)} + 6y \cdot {y^{(1)}}^2 \\
y^{(4)} &= 6 + 6y \cdot y^{(2)} + 3y^2 \cdot y^{(3)} + 6{y^{(1)}}^2 + 12y \cdot y^{(1)} \cdot y^{(2)}
\end{aligned}
$$

- Get the value using initial values.

$$\textbf{while } y(0) = -1, y^{(1)}(0) = -1, y^{(2)}(0) = -3, y^{(3)}(0) = -15$$
$$\textbf{while } y(0) = 0, y^{(1)}(0) = 0, y^{(2)}(0) = 0, y^{(3)}(0) = 0, y^{(4)}(0) = 6$$
$$\textbf{while } y(0) = 1, y^{(1)}(0) = 1, y^{(2)}(0) = 3, y^{(3)}(0) = 15$$

- Compute the solution

$$\textbf{While } y(0) = -1, y(t) = -1 + \frac{-1}{1!}t + \frac{-3}{2!}t^2 + \frac{-15}{3!}t^3 + \cdots$$
$$\textbf{While } y(0) = 0, y(t) = 0 + \frac{0}{1!}t + \frac{0}{2!}t^2 + \frac{0}{3!}t^3 + \frac{6}{4!}t^4 + \cdots$$
$$\textbf{(In this case, the solution start from 4th number)}$$
$$\textbf{While } y(0) = 1, y(t) = 1 + \frac{1}{1!}t + \frac{3}{2!}t^2 + \frac{15}{3!}t^3 + \cdots$$

Then, we can compute the similar result by using the second method.

Second Method :

- As the given function, we could obtain the first derivative.

$$y(t) = \sum_{n=0}^{\infty} a_n \cdot t^n, y^{(1)}(t) = n \cdot \sum_{n=0}^{\infty} a_n \cdot t^{n-1}$$

- According to ODE

$$t^3 + y^3 = y^{(1)} = \sum_{n=1}^{\infty} n \cdot a_n \cdot t^{n-1} = \sum_{n=0}^{\infty} (n+1) \cdot a_{n+1} \cdot t^n, y^3 = \left(\sum_{n=0}^{\infty} a_n t^n \right)^3 = \sum_{n=0}^{\infty} \left(\sum_{\substack{i+j+k=n \\ i,j,k \geq 0}} a_i a_j a_k \right) t^n$$

So

$$\sum_{n=0}^{\infty} \left(\sum_{\substack{i+j+k=n \\ i,j,k \geq 0}} a_i a_j a_k \right) t^n + t^3 = \sum_{n=0}^{\infty} (n+1) \cdot a_{n+1} \cdot t^n$$

- Calculate the coefficient

$$\textbf{While } n \neq 3, (n+1) \cdot a_{n+1} = \sum_{\substack{i+j+k=n \\ i,j,k \geq 0}} a_i a_j a_k$$
$$\textbf{While } n = 3, (n+1) \cdot a_{n+1} = \sum_{\substack{i+j+k=3 \\ i,j,k \geq 0}} a_i a_j a_k + 1$$

- Together with the initial value

$$\textbf{While } a_0 = y(0) = -1, y(t) = -1 + \frac{-1}{1!}t + \frac{-3}{2!}t^2 + \frac{-15}{3!}t^3 + \cdots$$
$$\textbf{While } a_0 = y(0) = 0, y(t) = 0 + \frac{0}{1!}t + \frac{0}{2!}t^2 + \frac{0}{3!}t^3 + \frac{6}{4!}t^4 + \cdots$$
$$\textbf{While } a_0 = y(0) = 1, y(t) = 1 + \frac{1}{1!}t + \frac{3}{2!}t^2 + \frac{15}{3!}t^3 + \cdots$$
$$\textbf{All solutions are the same as previous methods}$$

2.4.2 Results Display

By using computer program, we can show the the third IVP in the following series function.

$$y(t) = 1 + t + \frac{3}{2}t^2 + \frac{5}{2}t^3 + \frac{37}{8}t^4 + \frac{321}{40}t^5 + \frac{197}{40}t^6 + \frac{713}{280}t^7 + \frac{289}{224}t^8 + \frac{2939}{2880}t^9 + \cdots$$

2.4.3 Error Analysis

The result seems fit well in positive part. We noticed that there are some difference in the figure (below) but that is because we only use first ten items to draw this figure, if we use first fifty, the answer will be better.

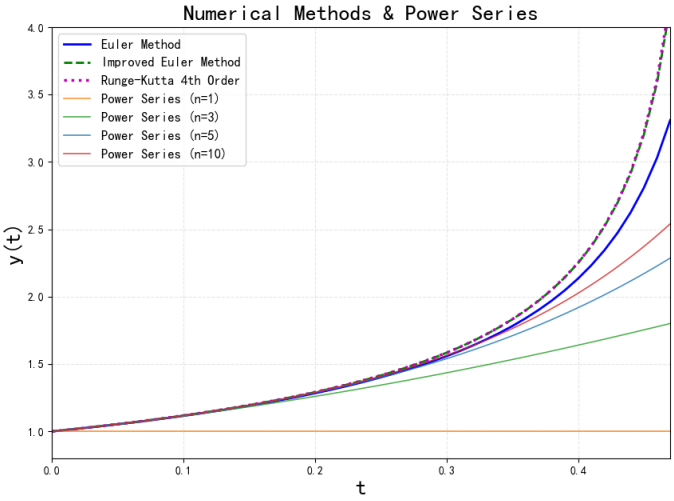


Figure 12: Series Solution of IVP with $y(0) = 1$

We can also see that as the number of item increase the solution is more accurate.

2.5 Runge-Kutta method

2.5.1 Mathematical Analysis

To numerically solve the initial value problem (IVP) given by:

$$y'(t) = y^3(t) + t^3$$

with an initial condition $y(t_0) = y_0$, the fourth-order Runge-Kutta (RK4) method is proposed. This method is well-established for its balance of accuracy and computational efficiency in approximating solutions to ODEs. The RK4 method iteratively computes the solution at discrete time steps $t_{n+1} = t_n + h$, where h is the step size, using the following set of equations:

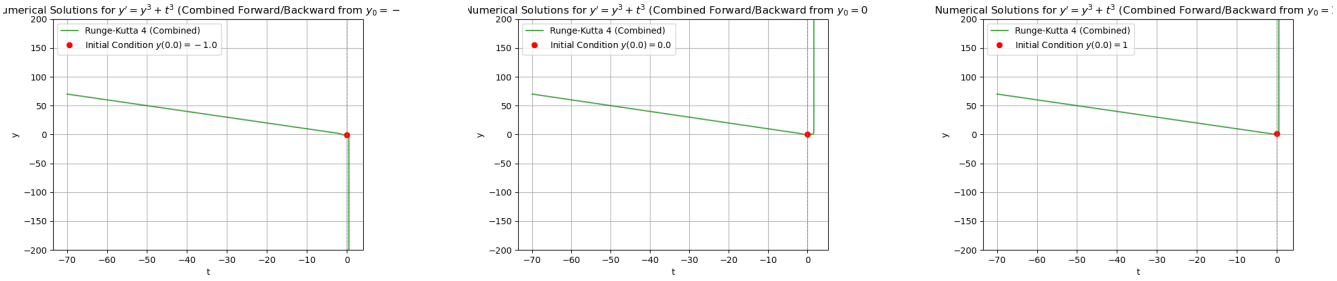
$$\begin{aligned}
k_1 &= y_n^3 + t_n^3 \\
k_2 &= \left(y_n + \frac{h}{2}k_1\right)^3 + \left(t_n + \frac{h}{2}\right)^3 \\
k_3 &= \left(y_n + \frac{h}{2}k_2\right)^3 + \left(t_n + \frac{h}{2}\right)^3 \\
k_4 &= (y_n + hk_3)^3 + (t_n + h)^3
\end{aligned}$$

The value of the solution at the next time step is then updated by:

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

2.5.2 Result Presentation

Here are the results by the Runge-Kutta Method, we use a very little h, because if we use a large h, the function will get fail in the negative x-axis very quickly, so for the little range of x from -70 to 2, we only take h=0.001



(a) y0=-1 (b) y0=0 (c) y0=1

Figure 13: Runge-Kutta method

2.5.3 Error Analysis1: Local Truncation Error (LTE)

The Local Truncation Error (LTE) at step $n+1$ is the error made by a single step of the method, assuming the value y_n at the beginning of the step is exact, i.e., $y_n = y(t_n)$.

$$\text{LTE}_{n+1} = y(t_{n+1}) - (y(t_n) + h\Phi_{RK4}(t_n, y(t_n), h))$$

where $h\Phi_{RK4}(t_n, y(t_n), h) = \frac{h}{6}(k_1^* + 2k_2^* + 2k_3^* + k_4^*)$, with k_i^* calculated using $y(t_n)$ instead of y_n. Equivalently, it is often defined as the principal error term when y_{n+1} is subtracted from the Taylor expansion of $y(t_{n+1})$.

The Taylor expansion of the exact solution $y(t_{n+1})$ around t_n is:

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2!}y''(t_n) + \frac{h^3}{3!}y'''(t_n) + \frac{h^4}{4!}y^{(iv)}(t_n) + \frac{h^5}{5!}y^{(v)}(t_n) + h^6$$

Substituting $y' = f$, $y'' = f'$, $y''' = f''$, etc., where $f' = ft = ft + fyf$, and so on (total derivatives with respect to t):

$$y(t_{n+1}) = y(t_n) + hf + \frac{h^2}{2}f' + \frac{h^3}{6}f'' + \frac{h^4}{24}f''' + \frac{h^5}{120}f^{(iv)} + h^6$$

All functions $f, f', f'', \ldots$ are evaluated at $(t_n, y(t_n))$.

A full and detailed expansion of k_1, k_2, k_3, k_4 and their weighted sum $\Phi_{RK4} = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$ shows that when $y_n + h\Phi_{RK4}$ is formed, its Taylor series expansion in h matches the Taylor series expansion of $y(t_{n+1})$ up to and including the term of order h^4. Specifically, the expansion of $h\Phi_{RK4}(t_n, y(t_n), h)$ is:

$$h\Phi_{RK4} = hf + \frac{h^2}{2}f' + \frac{h^3}{6}f'' + \frac{h^4}{24}f''' + C_{RK4}h^5 + h^6$$

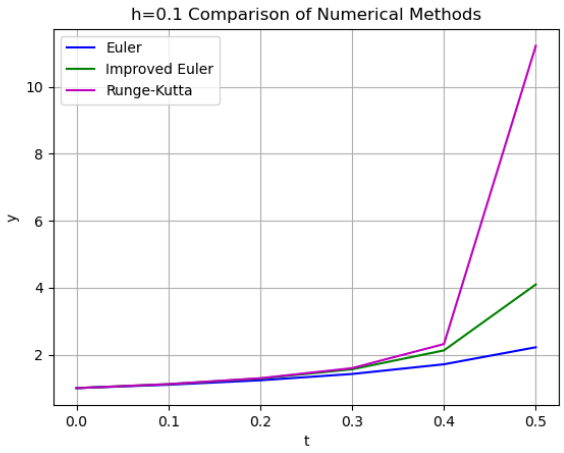
The coefficients of h, h^2, h^3, h^4 in $h\Phi_{RK4}$ match those in the Taylor expansion of $y(t_{n+1}) - y(t_n)$. The Local Truncation Error is then the difference:

$$\text{LTE}_{n+1} = y(t_{n+1}) - (y(t_n) + h\Phi_{RK4}(t_n, y(t_n), h)) = \left(\frac{1}{120}f^{(iv)} - C_{RK4}\right)h^5 + h^6$$

Thus, the LTE for RK4 is of order h^5:

$$\text{LTE}_{n+1} = h^5$$

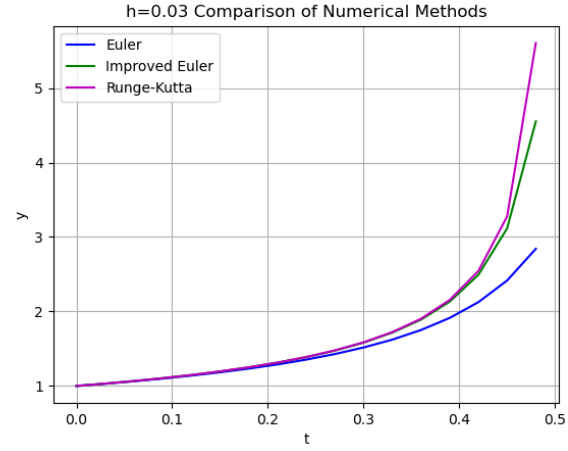
This means that the RK4 method is a fourth-order method (since the LTE is h^{p+1} for a p-th order method).



(a) h=0.1 graph

h=0. 1			
t	Euler_y1	Improved_Euler	RungeKutta
0. 2	1. 2332	1. 285708	1. 294834
0. 3	1. 421543	1. 562152	1. 596351
0. 4	1. 711506	2. 125814	2. 315678
0. 42	1. 813055	2. 519635	4. 096861
0. 43	1. 863829	2. 716546	4. 987453
0. 44	1. 914603	2. 913457	5. 878045
0. 45	1. 965378	3. 110367	6. 768637
0. 46	2. 016152	3. 307278	7. 659228
0. 47	2. 066926	3. 504189	8. 54982
0. 48	2. 117701	3. 701099	9. 440412
0. 49	2. 168475	3. 89801	10. 331004

(b) h=0.1 table



(a) h=0.03 graph

h=0. 03			
t	Euler_y1	Improved_Euler	RungeKutta
0. 2	1. 270307	1. 291926	1. 292975
0. 3	1. 5165	1. 581684	1. 585876
0. 4	1. 984137	2. 251853	2. 284553
0. 42	2. 125445	2. 492443	2. 544579
0. 43	2. 222203	2. 700251	2. 788531
0. 44	2. 318961	2. 90806	3. 032483
0. 45	2. 415719	3. 115868	3. 276435
0. 46	2. 557605	3. 594442	4. 052844
0. 47	2. 69949	4. 073016	4. 829252
0. 48	2. 841376	4. 55159	5. 60566
0. 49	3. 071878	7. 037093	1177. 188868

(b) h=0.03 table

3 Comparison for Different Methods and Different Steps

3.1 Analysis of Results for h = 0.1, 0.03, and 0.001

Here, I present the variations in results for three numerical methods (Euler, Improved Euler, and Runge-Kutta) using step sizes $h = 0.1$, $h = 0.03$, and $h = 0.001$. Tables and figures are provided to simplify the analysis.

The first figure corresponds to $h = 0.1$. It can be observed that the discrepancies between the methods are quite pronounced. The Runge-Kutta method appears to yield the most distinct curve.

The second scenario (presumably with $h = 0.03$, though not explicitly shown as a figure in the provided text) indicates that the Improved Euler method's results more closely approach those of the Runge-Kutta method. This suggests that as the step size h decreases, the accuracy of the Improved Euler method improves.

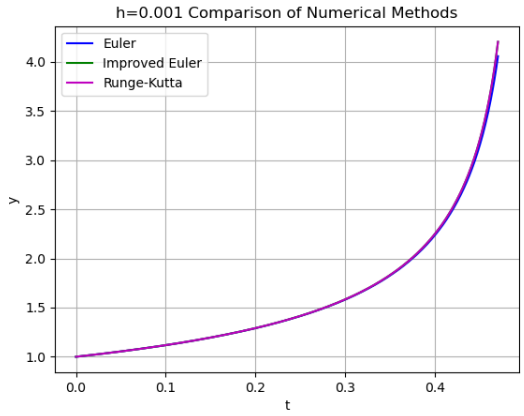
Overall, the Runge-Kutta method seems to be more accurate than the Improved Euler method, which in turn is more accurate than the Euler method.

4 Symmetry Considerations

1. Nullcline

When $y' = 0$, we have $y^3 + t^3 = 0$. In the real domain, this means $y^3 = -t^3$, so $y = -t$.

Therefore, on the line $y = -t$, the slope of the solution curves of the differential equation is zero. This implies that if a solution curve crosses the line $y = -t$ (for $t \neq 0$), it will have a horizontal tangent at that point. This is



(a) h=0.001 graph

h=0. 001

t	Euler_y1	proved_Eul	RungeKutt
0. 2	1. 290652	1. 291478	1. 291479
0. 3	1. 581445	1. 584178	1. 584183
0. 4	2. 238666	2. 252298	2. 252339
0. 42	2. 502658	2. 524381	2. 524456
0. 43	2. 674808	2. 703307	2. 703414
0. 44	2. 887692	2. 926458	2. 926621
0. 45	3. 160282	3. 215682	3. 215949
0. 46	3. 526695	3. 611717	3. 612204
0. 47	4. 056245	4. 202178	4. 203235
0. 48	4. 919279	5. 225773	5. 228955
0. 49	6. 708974	7. 752308	7. 774964

(b) h=0.001 table

an important geometric property.

2. Behavior under Transformations

- The equation is not autonomous because it explicitly depends on t.
- It is not separable.
- It is not a standard homogeneous equation (in the sense $y' = F(y/t)$). If we try the substitution $y = ut$, then $u't + u = (ut)^3 + t^3 = u^3t^3 + t^3$, leading to $u' = t^2(u^3 + 1) - u/t$. This equation is not simpler.
- Consider the transformation: Let $s = -t$ and $u(s) = -y(-s)$. Then $t = -s$, and $y(t) = -u(-t)$. Thus, $y'(t) = \frac{d}{dt}(-u(-t)) = -u'(-t) \cdot (-1) = u'(-t)$. Substituting into the original equation $y' = y^3 + t^3$: $u'(-t) = (-u(-t))^3 + (-s)^3 = -u^3(-t) - s^3$. Let $x = -t$. Since $s = -t$ as well, we can use s. So, $u'(s) = -u^3(s) - (-s)^3 = -u^3(s) + s^3$. Thus, if $y(t)$ is a solution to $y' = y^3 + t^3$, then $u(s) = -y(-s)$ is a solution to the equation $u'(s) = s^3 - u^3(s)$. These two equations are similar in form but not identical.

5 Conclusion

In this lab report, we thoroughly explored the solutions to three initial value problems (IVPs) using both numerical and analytical methods. Specifically, we applied Euler, Improved Euler, Runge-Kutta, Picard-Lindelof iteration, and the Power Series method to solve the IVPs:

$$\begin{aligned} &\text{(IVP1)} \quad y' = t^3 + y^3, \quad y(0) = -1 \\ &\text{(IVP2)} \quad y' = t^3 + y^3, \quad y(0) = 0 \\ &\text{(IVP3)} \quad y' = t^3 + y^3, \quad y(0) = 1 \end{aligned}$$

Finally, we got the following for the solution's domain and behavior based on the initial condition y_0:

- When $y_0 = -1$, the function's domain extends from $-\infty$ to approximately **0.502**. The graph of the solution exhibits a shape similar to a "check mark" (or a hyperbola-like curve with two branches).
- When $y_0 = 0$, the function's domain extends from $-\infty$ to approximately **1.647**. The solution is strictly monotonically decreasing across its entire domain.
- When $y_0 = 1$, the function's domain extends from $-\infty$ to approximately **0.498**. Similar to the $y_0 = -1$ case, the solution's graph resembles a "check mark" shape.

6 Reference

Galaktionov, V. A. (1981). On the blow-up of solutions of the Cauchy problem for quasi-linear degenerate parabolic equations of arbitrary order. Differential Equations, 17(5), 737–743.

Arnold, V. I. (1973). Ordinary differential equations. MIT Press. Jordan, D. W., Smith, P. (2007). Nonlinear ordinary differential equations: An introduction for scientists and engineers (4th ed.). Oxford University Press.

Chipot, M., Weissler, F. B. (1989). Blow-up of solutions for a class of nonlinear ordinary differential equations. Nonlinear Analysis: Theory, Methods Applications, 13(1), 127–142.

Hairer, E., Nørsett, S. P., Wanner, G. (1993). Solving ordinary differential equations I: Nonstiff problems (2nd rev. ed., Springer Series in Computational Mathematics, Vol. 8). Springer.

Dumortier, F., Llibre, J., Artés, J. C. (2006). Qualitative theory of planar differential systems. Springer.

7 Ackonwledgements

Firstly, I'd like to say thank you to Professor Honold for assigning such an interesting task that we have learnt a lot. Then I would like to express my gratitude to all the TAs, especially TA Ren Jiashen for giving us lots' of instructions on the lab report. Thirdly, I have to say thank you to AI the Gemini, as it serves as a good coding helper for us to modify our code.

8 Appendix Coding

8.1 Euler Method

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

#
def f(t, y):
    return t**4 + y**3

# Heun
def heun_step(f, t, y, h):
    k1 = f(t, y)
    k2 = f(t + h, y + h * k1)
    y_new = y + (h / 2) * (k1 + k2)
    return y_new

# t >= 0
def integrate_forward(y0, t_span=(0, 5), h=0.001, blowup_threshold=1e6):
    t0, t_end = t_span
    t_values = [t0]
    y_values = [y0]
    t, y = t0, y0

    while t < t_end:
        y_next = heun_step(f, t, y, h)
        if abs(y_next) > blowup_threshold:
            return np.array(t_values), np.array(y_values), t
        t += h
        t_values.append(t)
        y_values.append(y_next)
        y = y_next
    return np.array(t_values), np.array(y_values), None

# t <= 0
def integrate_backward(y0, t_span=(-5, 0), h=0.001, blowup_threshold=1e6):
    t_start, t0 = t_span
    t_values = [t0]
    y_values = [y0]
    t, y = t0, y0

    while t > t_start:
        y_next = heun_step(f, t, y, -h)  #
        if abs(y_next) > blowup_threshold:
            return np.array(t_values), np.array(y_values), t
        t -= h
        t_values.insert(0, t)
        y_values.insert(0, y_next)
        y = y_next
    return np.array(t_values), np.array(y_values), None

#
y0_list = [-1, 0, 1]
colors = ['green', 'blue', 'orange']
h = 0.001
blowup_threshold = 1e6
```

```
plt.figure(figsize=(10, 6))

for y0, color in zip(y0_list, colors):
    #
    t_fwd, y_fwd, blowup_time_fwd = integrate_forward(y0, t_span=(0, 5), h=h, blowup_threshold=blowup_thres

    #
    t_bwd, y_bwd, blowup_time_bwd = integrate_backward(y0, t_span=(-5, 0), h=h, blowup_threshold=blowup_th

    #
    t_total = np.concatenate((t_bwd, t_fwd))
    y_total = np.concatenate((y_bwd, y_fwd))

    #
    plt.plot(t_total, y_total, label=f"y(0) = {y0}", color=color)

    #  blow-up
    for blowup_time in [blowup_time_bwd, blowup_time_fwd]:
        if blowup_time is not None:
            plt.axvline(x=blowup_time, linestyle='--', color='red', alpha=0.7,
                        label=f"Blow-up est. at t  {blowup_time:.4f} (y0={y0})")

#
plt.title("Solutions of $ y' = t^3 + y^3 $ using Heun's Method")
plt.xlabel("t")
plt.ylabel("y(t)")
plt.legend()
plt.grid(True)
plt.xlim(-5, 5)
plt.ylim(-10, 10)
plt.tight_layout()
plt.show()
```

8.2 Real Value

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

def ode_func(t, y):
    return t**3 + y**3

# Initial conditions
y0_values = [0, 1, -1]

# Store results: {'y0': {'sol_fwd': sol_fwd, 'sol_bwd': sol_bwd, 'a': a, 'b': b}}
results = {}

# Integration parameters
t_span_limit = 20  # Max integration time in each direction (can be adjusted)
rtol = 1e-9        # Relative tolerance
atol = 1e-12       # Absolute tolerance (important for reaching asymptotes accurately)
# Using a large value for y to stop integration is an option,
# but solve_ivp often handles blow-up by itself by reducing step size until failure.
# We can check sol.status and sol.message.
```

```
for y0_val in y0_values:
    print(f"\nProcessing for y(0) = {y0_val}")


    # Forward integration
    # Integrate from t=0 to t_span_limit
    # If it blows up, sol_fwd.t[-1] is an estimate of b
    # We need to make sure t_span is large enough to pass the asymptote if one exists.
    # solve_ivp will stop if it can't proceed (e.g. asymptote)
    sol_fwd = solve_ivp(ode_func, [0, t_span_limit], [y0_val],
                        dense_output=True, rtol=rtol, atol=atol, method='Radau') # Radau is good for stiff

    b_asymptote = np.inf
    if sol_fwd.status == 0 and sol_fwd.t[-1] < t_span_limit : # solver might have stopped before t_span_li
         print(f"  Forward integration for y0={y0_val} finished early at t={sol_fwd.t[-1]} but status is 0
         # If it truly didn't blow up, b is inf. Otherwise, it might be an issue with solver params or t_s
         # For this problem, blow-up is expected if y gets large.
         if abs(sol_fwd.y[0,-1]) < 1e6: # Arbitrary threshold to decide if it's not a blow up
            b_asymptote = np.inf
         else: # Assume it was on its way to blow up but stopped.
             t_last_fwd = sol_fwd.t[-1]
             y_last_fwd = sol_fwd.y[0, -1]
             if abs(y_last_fwd) > 1e3 : # Check if y is large enough for extrapolation
                 b_asymptote = t_last_fwd + 1.0 / (2.0 * y_last_fwd**2) if y_last_fwd !=0 else t_last_fwd =
                 print(f"  Extrapolated forward asymptote b for y0={y0_val}: {b_asymptote:.6f} from t_last=
             else:
                 print(f"  Forward solution for y0={y0_val} ended at t={t_last_fwd:.6f}, y={y_last_fwd:.2e}
                 b_asymptote = t_last_fwd # Best guess is where it stopped.
    elif sol_fwd.t[-1] < t_span_limit:  # Solver stopped early, likely due to stiffness/blow-up (status !=


        t_last_fwd = sol_fwd.t[-1]
        y_last_fwd = sol_fwd.y[0, -1]
        # Extrapolation for right asymptote b: b_approx = t_last + 1/(2*y_last^2)
        # This formula is for y' ~ y^3.
        # Need to be careful if y_last is small or zero.
        if abs(y_last_fwd) > 1e3: # Check if y is large enough for extrapolation
            b_asymptote = t_last_fwd + 1.0 / (2.0 * y_last_fwd**2)
            print(f"  Forward integration for y0={y0_val} stopped at t={t_last_fwd:.6f}, y={y_last_fwd:.2e}
            print(f"  Extrapolated forward asymptote b for y0={y0_val}: {b_asymptote:.6f}")
        else:
            print(f"  Forward integration for y0={y0_val} stopped at t={t_last_fwd:.6f}, y={y_last_fwd:.2e}
            b_asymptote = t_last_fwd # Best guess is where it stopped.
    else: # Reached t_span_limit
        print(f"  Forward integration for y0={y0_val} reached t_span_limit={t_span_limit} without apparent
        b_asymptote = np.inf




    # Backward integration
```

```
    # Integrate from t=0 to -t_span_limit
    sol_bwd = solve_ivp(ode_func, [0, -t_span_limit], [y0_val],
                        dense_output=True, rtol=rtol, atol=atol, method='Radau')

    a_asymptote = -np.inf
    if sol_bwd.status == 0 and sol_bwd.t[-1] > -t_span_limit:
        print(f"  Backward integration for y0={y0_val} finished early at t={sol_bwd.t[-1]} but status is 0
        if abs(sol_bwd.y[0,-1]) < 1e6:
             a_asymptote = -np.inf
        else:
            t_last_bwd = sol_bwd.t[-1]
            y_last_bwd = sol_bwd.y[0, -1]
            if abs(y_last_bwd) > 1e3:
                a_asymptote = t_last_bwd - 1.0 / (2.0 * y_last_bwd**2) if y_last_bwd != 0 else t_last_bwd
                print(f"  Extrapolated backward asymptote a for y0={y0_val}: {a_asymptote:.6f} from t_last
            else:
                print(f"  Backward solution for y0={y0_val} ended at t={t_last_bwd:.6f}, y={y_last_bwd:.2e}
                a_asymptote = t_last_bwd
    elif sol_bwd.t[-1] > -t_span_limit:  # Solver stopped early
        t_last_bwd = sol_bwd.t[-1]
        y_last_bwd = sol_bwd.y[0, -1]
        # Extrapolation for left asymptote a: a_approx = t_last - 1/(2*y_last^2)
        if abs(y_last_bwd) > 1e3:
            a_asymptote = t_last_bwd - 1.0 / (2.0 * y_last_bwd**2)
            print(f"  Backward integration for y0={y0_val} stopped at t={t_last_bwd:.6f}, y={y_last_bwd:.2
            print(f"  Extrapolated backward asymptote a for y0={y0_val}: {a_asymptote:.6f}")
        else:
            print(f"  Backward integration for y0={y0_val} stopped at t={t_last_bwd:.6f}, y={y_last_bwd:.2
            a_asymptote = t_last_bwd
    else: # Reached -t_span_limit
        print(f"  Backward integration for y0={y0_val} reached -t_span_limit={-t_span_limit} without appar
        a_asymptote = -np.inf

    results[y0_val] = {
        'sol_fwd': sol_fwd,
        'sol_bwd': sol_bwd,
        'a': a_asymptote,
        'b': b_asymptote,
        'domain': (a_asymptote, b_asymptote)
    }

    print(f"  For y(0) = {y0_val}, estimated domain: ({a_asymptote:.6f}, {b_asymptote:.6f})")

# Plotting the solutions
plt.figure(figsize=(12, 8))
for y0_val, res in results.items():
    sol_fwd = res['sol_fwd']
    sol_bwd = res['sol_bwd']
    a = res['a']
    b = res['b']

    # Forward solution plot
    t_plot_fwd = np.linspace(0, sol_fwd.t[-1], 200) # ensure plotting up to where solver stopped
    if b != np.inf and b < sol_fwd.t[-1] + 1/(2*sol_fwd.y[0,-1]**2+1e-9): # A bit of a hack to ensure linsp
        # if extrapolated b is very close to sol_fwd.t[-1] or slightly less due to numerical precision in
        # ensure t_plot_fwd doesn't exceed b by much if b is the true limit
```

```
        t_plot_fwd = np.linspace(0, min(sol_fwd.t[-1], b if b >0 else sol_fwd.t[-1] ), 200)


    y_plot_fwd = sol_fwd.sol(t_plot_fwd)[0]

    # Backward solution plot
    t_plot_bwd = np.linspace(sol_bwd.t[-1], 0, 200) # ensure plotting up to where solver stopped
    if a != -np.inf and a > sol_bwd.t[-1] - 1/(2*sol_bwd.y[0,-1]**2+1e-9) :
         t_plot_bwd = np.linspace(max(sol_bwd.t[-1], a if a < 0 else sol_bwd.t[-1]), 0, 200)


    y_plot_bwd = sol_bwd.sol(t_plot_bwd)[0]

    # Combine and remove duplicates, sort by time
    t_combined = np.concatenate((t_plot_bwd[:-1], t_plot_fwd)) # Avoid duplicating t=0
    y_combined = np.concatenate((y_plot_bwd[:-1], y_plot_fwd))

    # Sort by t for correct plotting
    sort_indices = np.argsort(t_combined)
    t_combined = t_combined[sort_indices]
    y_combined = y_combined[sort_indices]

    # Filter out extreme values if they mess up the plot too much before reaching asymptote line
    # y_combined = np.clip(y_combined, -50, 50) # Optional: for better visualization if blowup is too fast

    plt.plot(t_combined, y_combined, label=f'$y(0) = {y0_val}$')

    if b != np.inf:
        plt.axvline(x=b, linestyle='--', color=plt.gca().lines[-1].get_color(), alpha=0.7, label=f'$t={b:.
    if a != -np.inf:
        plt.axvline(x=a, linestyle='--', color=plt.gca().lines[-1].get_color(), alpha=0.7, label=f'$t={a:.


plt.xlabel('$t$')
plt.ylabel('$y(t)$')
plt.title('Solutions to $y\' = t^3 + y^3$')
plt.legend(loc='best')
plt.grid(True)
plt.ylim([-10, 10]) # Adjust ylim for better visualization of behavior near origin; asymptotes make it har
plt.show()

# Print summary of domains
print("\nSummary of Maximal Solution Domains:")
for y0_val, res in results.items():
    print(f"For y(0) = {y0_val}: Domain = ({res['domain'][0]:.6f}, {res['domain'][1]:.6f})")
```

8.3 Comparison Coding

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

def euler(f,y0,t0,tn,h):
    t=np.arange(t0,tn+h,h)
    y=np.zeros_like(t)
    y[0]=y0
    for i in range(len(t)-1):
        y[i+1]=y[i]+h*f(t[i],y[i])
```

```
    return t,y

def improved_euler(f,y0,t0,tn,h):
    t=np.arange(t0,tn+h,h)
    y=np.zeros_like(t)
    y[0]=y0
    for i in range(len(t)-1):
        y_hat = y[i] + h * f(t[i], y[i])

        #
        y[i+1] = y[i] + 0.5 * h * (f(t[i], y[i]) + f(t[i+1], y_hat))
    return t,y

def runge_kutta(f,y0,t0,tn,h):
    t=np.arange(t0,tn+h,h)
    y=np.zeros_like(t)
    y[0]=y0
    for i in range(len(t)-1):
        k1=h*f(t[i],y[i])
        k2=h*f(t[i]+h/2,y[i]+k1/2)
        k3=h*f(t[i]+h/2,y[i]+k2/2)
        k4=h*f(t[i]+h,y[i]+k3)
        y[i+1]=y[i]+(k1+2*k3+2*k3+k4)/6
    return t,y

def f(t, y):
    return y**3+t**3  #  dy/dt = y

t0=0
y0=1
tn=0.5
h=0.001

t,y1=euler(f,y0,t0,tn,h)
t,y2=improved_euler(f,y0,t0,tn,h)
t,y3=runge_kutta(f,y0,t0,tn,h)

# yture=np.exp(t)

plt.plot(t, y1, 'b-', label='Euler')
plt.plot(t, y2, 'g-', label='Improved Euler')
plt.plot(t, y3, 'm-', label='Runge-Kutta')
# plt.plot(t, ytrue, 'r-', label='True')

plt.legend()
plt.xlabel('t')
plt.ylabel('y')
plt.title('Comparison of Numerical Methods')
plt.grid(True)
plt.show()
```

8.4 Picard Iteration

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

```
import math
def picard_lindelof_iteration(f, y0, t, max_iter=100, tol=1e-6):


    #
    y = np.zeros_like(t)
    y_prev = np.zeros_like(t)

    #  y0
    y_prev.fill(y0)

    for iteration in range(max_iter):
        #
        for i in range(len(t)):
            #  t=0
            if i == 0:
                y[i] = y0
            else:
                # y(t) = y0 +  f(s, y_prev(s)) ds
                y[i] = y0 + integrate.quad(lambda s: f(s, y_prev[i-1]), t[0], t[i])[0]

        #  y
        # print(f"Iteration {iteration + 1}: y = {y}")

        #
        if np.max(np.abs(y - y_prev)) < tol:
            print(f"Picard iteration converged in {iteration+1} iterations")
            return y

        #
        y_prev = np.copy(y)

    print(f"Picard iteration did not converge after {max_iter} iterations")
    return y


#
if __name__ == "__main__":
    #  y' = y y(t) = e^t

    def f(t, y):
        return y**3+t**3

    #

    tn=1
    h=0.001


    y0 = 1.0   #  y(0) = 1

    s=math.ceil(tn/h) #

    t = np.linspace(0, tn, s)  #
```

```
#
plt.figure(figsize=(10, 6))
for i in range(6):
    yi = picard_lindelof_iteration(f, y0, t, max_iter=i+1)
    plt.plot(t, yi, label=f'Iteration {i+1}')



#
# exact_solution = np.exp(t)
# plt.plot(t, exact_solution, 'r--', label='Exact Solution')

#
plt.legend()
plt.xlabel('t')
plt.ylabel('y')
plt.title('Picard-Lindelöf Iteration Convergence')
plt.grid(True)
plt.show()
```