ZJU-UIUC INSTITUTE

2023SP MATH285 LAB REPORT

Group Name:

MATH285gg

Group Members:

Jun Liang	321011xxxx
Xinchen Yin	321011xxxx
Ziming Yan	321011xxxx
Xingjian Kang	321011xxxx
Yujie Pan	321011xxxx

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:

1 Introduction

The task for this lab is to consider two ODEs, namely,

$$\frac{\mathrm{d}y}{\mathrm{d}t} = (y - t^2)(y^2 - t), \quad y(0) = 0 \tag{1}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = (y - t^2)(y^2 - t), \quad y(0) = 1 \tag{2}$$

In this report, we will try to solve it in different ways, including some numerical methods and power series methods. We will also use some theories to make some qualitative analysis including the domain, stability, and asymptote. Before we start, let us refer IVP1 and IVP2 to equations (1) and (2) respectively. And we will assume that $f(t, y) = (y - t^2)(y^2 - t)$.

2 Numerical Methods

2.1 Euler Method

Simply describe this method.

2.1.1 Formula and Derivation

For an interval $[0, a], a > 0$, if it is divided by n equal parts, which leads to $0 = t_0 \leq t_1 \leq t_2 \leq ... \leq t_n = a$, and let $h = t_{k+1} - t_k, k = 0, 1, 2, ...n - 1$. The Euler Formula is claimed as

$$y_{k+1} = y_k + h * f(t_k, y_k)$$

where $y_k (k = 1, 2, ...n)$ is the estimated value of the function and y_0 is the given Initial Value $y(0)$. So we can get $y(t_k) \approx y_k$. Then we can use y_k as the estimation. The steps are as follows

$$y(t_0) = y(0)$$

$$y(t_1) \approx y_1 = y_0 + h * f(t_0, y_0) = y(0) + h * f(t_0, y(0))$$

$$y(t_2) \approx y_2 = y_1 + h * f(t_1, y_1)$$

$$\vdots$$

$$y(t_n) \approx y_n = y_{n-1} + h * f(t_{n-1}, y_{n-1})$$

So we can get all estimated values from the process above. And it is easy to see that if we increase n(or decrease h), the precision will increase as well.

2.1.2 Analysis of the IVP

For IVP1, $y(0) = 0$, so if we choose $a = 1$ and $h = 0.1$, then we can estimate the following values

$$y(0.1), y(0.2), ..., y(1)$$

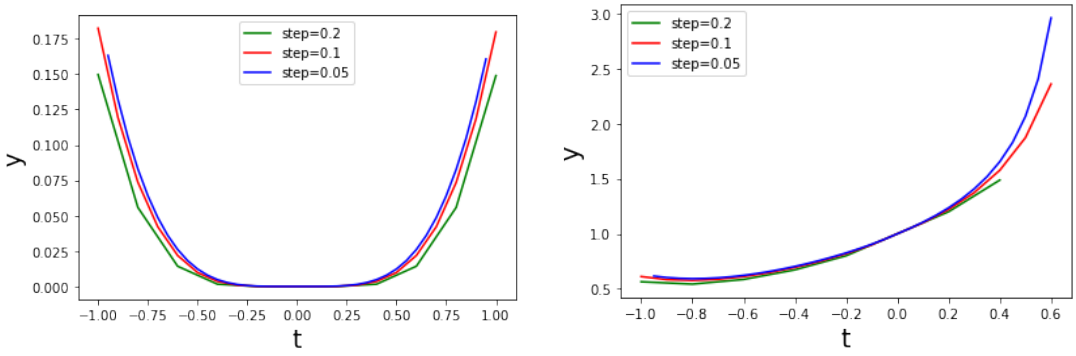
Similarly, for the interval $[-a, 0], a > 0$, we can just take h as -0.1 Then we can estimate the following values in the interval $[-1, 1]$ and plot them(Fig.1(a)).

$$y(-1), y(-0.9), ..., y(0.9), y(1)$$

For IVP2, similarly, if we choose the interval $[-1, 0.6]$, and the step $h = 0.1$, we can get the plot below(Fig.1(b)).

Then, by decreasing the step, we claim that the precision will increase. And we can see it is true in Fig. 2(a) and Fig. 2(b).

Figure 1: Euler Method with steps



(a) IVP1, Euler Method with several steps

(b) IVP2, Euler Method with several steps

2.1.3 Error Analysis

After referring to some reading materials [BDM21], we can learn that the local truncation error of Euler method on an interval $[a, b]$ is given by

$$|e_n| \leq \frac{1}{2}Mh^2$$

where M is the maximum of y'' on the interval $[a, b]$. And moreover, we can obtain the global truncation error E_n is

$$E_n \leq Kh$$

for some constant K, so we can say that the global truncation error for Euler method is $O(h)$.

2.2 Improved Euler Method

2.2.1 Formula and Derivation

If we look at Euler method in a geometric way, it is easy to see its essence is to estimate the integral by using the area of a rectangle. More accurately, however, we can use the area of a trapezoid to simulate integration. Namely, by replacing the growth rate of y_n, we can easily get the relation below

$$y_{k+1} = y_k + \frac{h}{2} * f(t_k, y_k) + f(t_{k+1}, y_{k+1})$$

where $k = 0, 1, 2, ...n - 1$, and other notations also have the same meanings as above. The formula is called Improve Euler Formula. And we can just see that it is based on the Euler method. In other words, it uses Euler Method to calculate y_k at first and then improve it. The improvement will be shown in the next section.

2.2.2 Analysis of the IVP

The analysis of the two IVPs is just similar to that in 2.1.2. For IVP1 and IVP2, by using several steps $h = 0.1, 0.05, 0.01$ and the interval $[-1, 1]$ and $[-1, 0.6]$ separately, we can obtain the following graphs(Fig.3 to Fig.5) compared with what we get by using Euler method.

It is easy to see that as the step decreases, the difference between the two methods decreases, which meets our expectations.

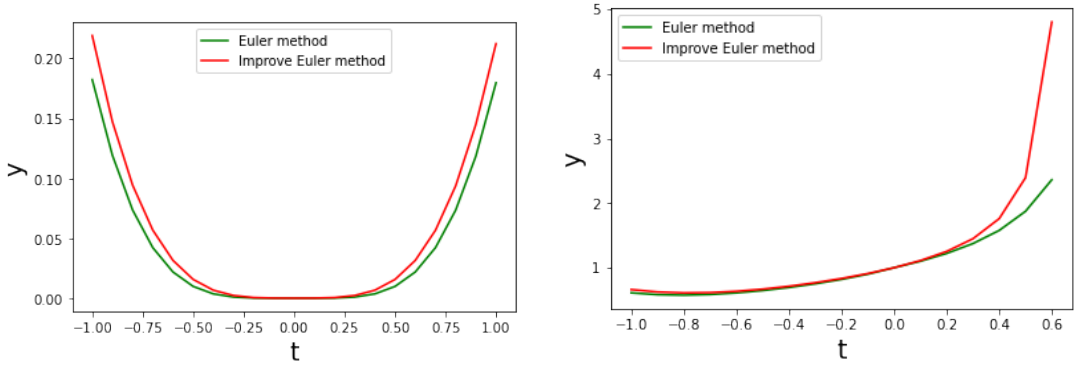
2.2.3 Error Analysis

Similar to Euler Method, the global truncation error E_n of the improved Euler method can be calculated theoretically as

$$E_n \leq Kh^2$$

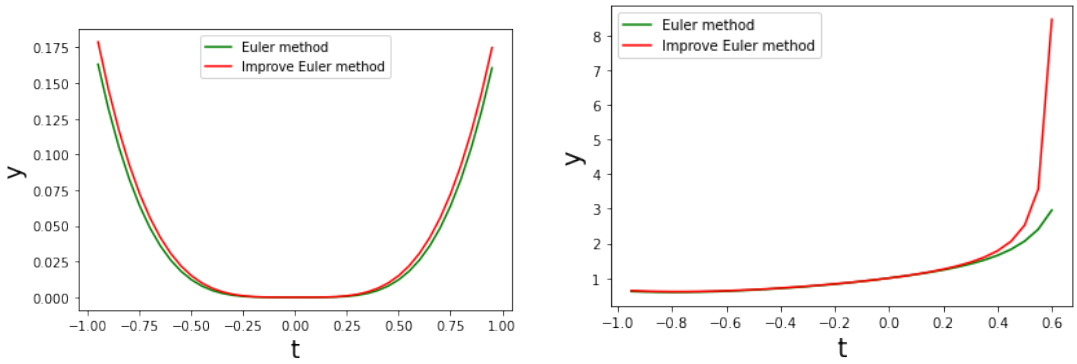
for some constant K, so we can say that the global truncation error of improved Euler method is $O(h^2)$, which is more accurate than Euler method.

Figure 2: Comparations between Euler and improve Euler, with step=0.1



(a) IVP1 (b) IVP2

Figure 3: Comparations between Euler and improve Euler, with step=0.05



(a) IVP1 (b) IVP2

2.3 Runge-Kutta Method

2.3.1 Basic Idea

The basic idea of Runge-Kutta method is that by Mean Value Theorem,

$$y(t_{n+1}) = y(t_n) + hy'(\epsilon), \quad t_n < \epsilon < t_{n+1}$$

where h is step length. Using $y'(t_n) = f(t_n, y_n)$ we have

$$y(t_{n+1}) = y(t_n) + hf(\epsilon, y(\epsilon))$$

and $f(\epsilon, y(\epsilon))$(noted as k) represents the average slope in interval (x_n, x_{n+1}). Thus the goal of Runge-Kutta method is to gain k as accurately as possible, and Euler can be regarded as first-order Runge-Kutta, improved Euler can be considered as second-order Runge-Kutta.

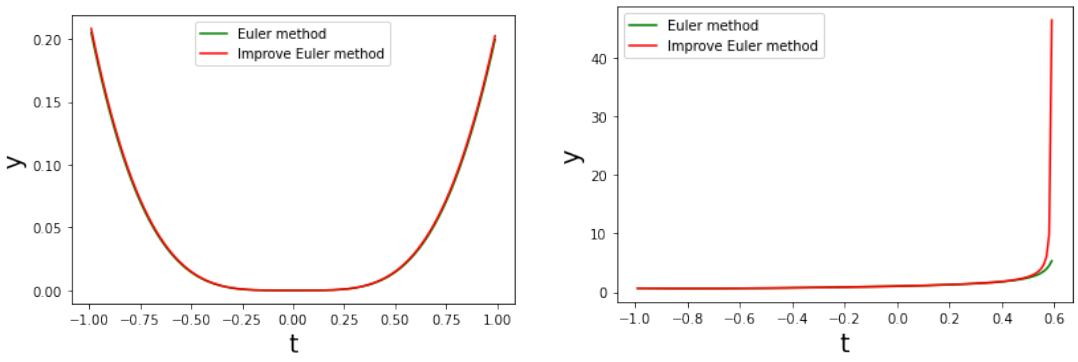
From Euler to improved Euler, we reduce the error by using an extra point at x_{n+1}, and in Runge-Kutta method, we will use more slopes of points in the range of (t_n, t_{n+1}) and take the average of them to reduce error.

2.3.2 Formula

The most widely known member of the Runge–Kutta family is generally referred to as "RK4". It's stated as follows

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

Figure 4: Comparations between Euler and improve Euler, with step=0.01



(a) IVP1 (b) IVP2

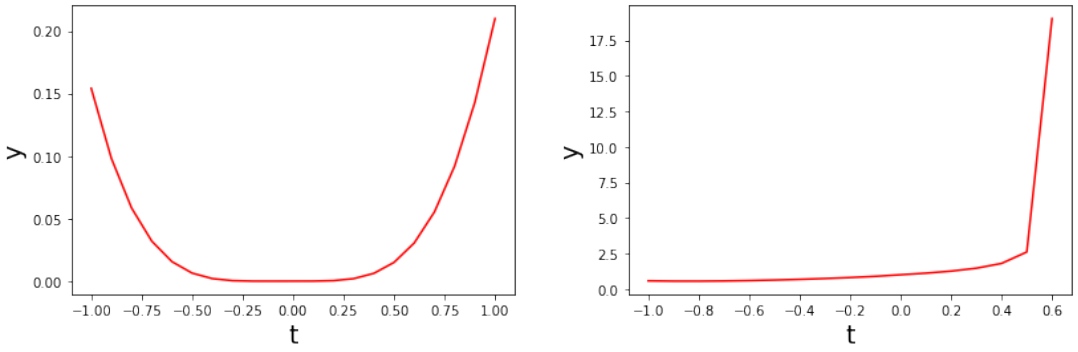
where

$$
\begin{aligned}
k_1 &= f(t_n, y_n) \\
k_2 &= f(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}) \\
k_3 &= f(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}) \\
k_4 &= f(t_n + h, y_n + hk_3)
\end{aligned}
$$

2.3.3 IVP Analysis

For IVP1 and IVP2, the fitting curve is as follow.

Figure 5: Runge-Kutta curve for IVP1 and IVP2



(a) IVP1 (b) IVP2

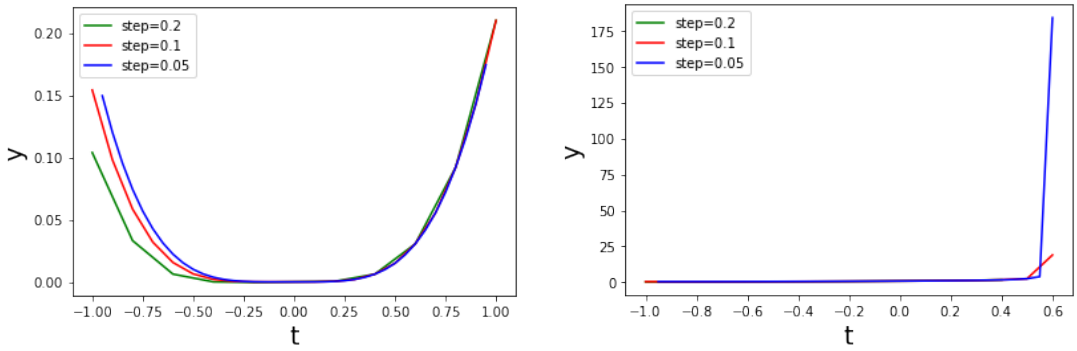
Then comparing the different step length, we can find that step length 0.1 is a good choice. Also, we can find the upper bound of IVP2 is at about 0.6, as at 0.6 it's about to increase very fast to infinite.

By changing the lower bound and upper of plotting, we can also find the lower bound of accurate plotting of IVP1 is about -2.4, the upper bound of IVP1 is about 3.5.(Fig.7)
Also the lower bound for IVP2 is about -2.4, the upper bound is about 0.6.(Fig.8)

2.3.4 Error Analysis

As the Derivation of Runge-Kutta method is based on Taylor series, thus the error is $O(h^5)$

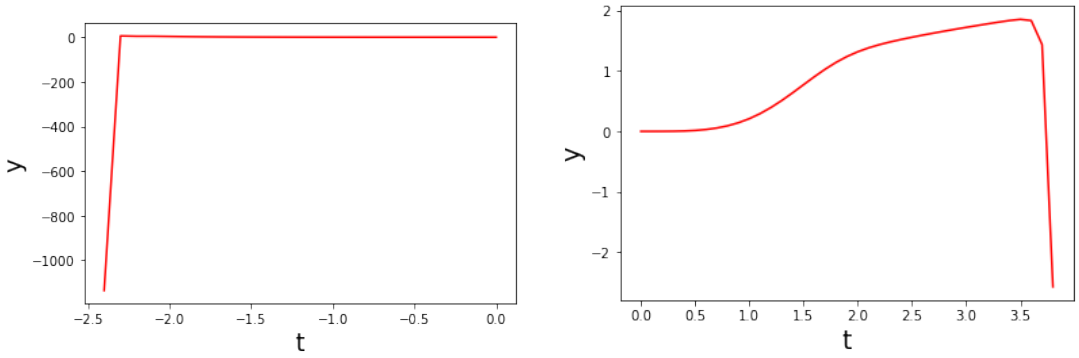
Figure 6: Runge-Kutta step length comparation for IVP1 and IVP2



(a) IVP1

(b) IVP2

Figure 7: Bounds using Runge-Kutta for IVP1



(a) IVP1 lower bound

(b) IVP1 upper bound

3 Linear Multistep Methods

In a step-by-step approach to solving, a series of approximations to y_{n+1} has in fact been derived prior to the calculation of $y_0, y_1 ... y_n$. If the information from the preceding multiple parts is fully utilised to predict y_{n+1}, a high degree of accuracy can be expected. This is the basic idea behind the construction of the resulting linear multi-step method.

3.1 Formula and Derivation

The formula of Linear Multistep Method generally can be presented as

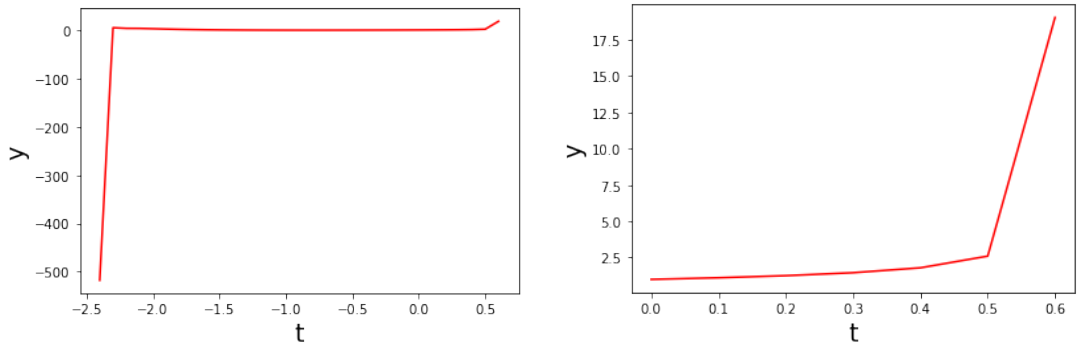
$$y_{n+k} = \sum_{i=0}^{k-1} a_i y_i + h \sum i = 0^k b_i f_{n+i} (i = 0, 1..., k-2)$$

y_{n+1} is the approximate number of $y(x_{n+1})$, $f_{n+i} = f(t_{n+i}, y_{n+i})$, $t_{n+i} = t_n + ih$, a,b are constants, a_0, b_0 are not all zeros. And this is called Linear k-step method. h is the length of each step. By giving the previous K approximations $y_0, y_1 ..., y_{k-1}$, we can use them to compute $y_k, y_{k+1}, ...$

3.2 Analysis

There are three specific ways to solve IVP,Milne-Simpson method, Adams-Bashforth-Moulton method and Hamming method. For this IVP, we need to use linear 4th-step method. Therefore, we need to get the first 4 points by Runge-Kutta method.

Figure 8: Bounds using Runge-Kutta for IVP2



(a) IVP2 lower bound

(b) IVP1 upper bound

3.3 Adams-Bashforth-Moulton method

By using Lagrange polynomials, based on

$$(t_{k-3}, f_{k-3}), (t_{k-2}, f_{k-2}), (t_{k-1}, f_{k-1}), (t_k, f_k)$$

we can get

$$y'_{k+1} = y_k + \int_{t_{k+1}}^{t_k} f(t,y) = y_k + \frac{h}{24}(55f_k - 59f_{k-1} + 37f_{k-2} - 9f_{k-3}$$

With this, we find the point $(t_{k+1}, f(t_{k+1}, y'_{k+1}))$ and we can use it to find y_{k+1}, by using the integration on $[t_k, t_{k+1}]$. That is

$$y_{k+1} = y_k + \int_{t_{k+1}}^{t_k} f(t,y) = y_k + \frac{h}{24}(9f_{k+1} - 19f_{k-1} - 5f_{k-1} + f_{k-2}$$

3.4 Milne-Simpson method

Similar to Adams-Bashforth-Moulton method, The estimation is based on the calculation on$[t_{k-3}, t_{k+1}]$.By using Lagrange polynomials, based on$(t_{k-3}, f_{k-3}), (t_{k-2}, f_{k-2}), (t_{k-1}, f_{k-1}), (t_k, f_k)$we can get

$$y'_{k+1} = y_{k-3} + \int_{t_{k+1}}^{k_{k-3}} f(t,y) = y_{k-3} + \frac{4h}{3}(2f_{k-2} - fk - 1 + 2f_k$$

With this, we find the point $(t_{k+1}, f(t_{k+1}, y'_{k+1}))$ and we can use it to find y_{k+1}, by using the integration on $[t_{k-3}, t_{k+1}]$. That is

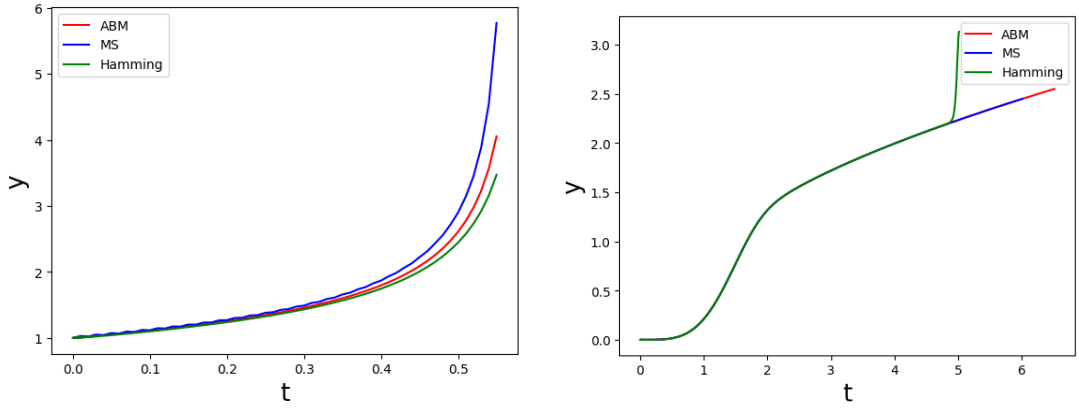
$$y_{k+1} = y_{k-1} + \int_{t_{k+1}}^{k_{k-3}} f(t,y) = y_{k-1} + \frac{h}{3}(f_{k-1} + 4f_k + f_{k+1})$$

3.5 Hamming method

Hamming method is also similar, and the formula is

$$y'_{k+1} = y_{k-3} + \frac{4h}{3}(2f_{k-2} - f_{k-1} + 2f_k)$$
$$y_{k+1} = \frac{-1}{8}y_{k-2} + \frac{9}{8}y_k + \frac{3h}{8}(-f_{k-1} + 2f_k + f_{k+1})$$

Figure 9: Graphs of three ways linear multistep method, with step=0.01



(a) IVP1 (b) IVP2

3.6 Error and Optimize

The reason why errors occur is that when we use linear multistep method, we need to predict the $y'k+1$ no matter by which kind of method. And the predictions are not accurate.

For Adams-Bashforth-Moulton, the error is about $\frac{28}{90}h^5y^{(iv)}(\xi)$
For Milne-Simpson method, the error is about $\frac{251}{720}h^5y^{(iv)}(\xi)$

In order to optimize prediction, we need to use methods to amend predictions, like Adams Moulton and Simpson. Actually, the method of Adams-Bashforth-Moulton is named for using the Adams-bashforth way to predict y'_{k+1}, then using Adams Moulton to amends, so as Milne-Simpson method (Milne formulate to predict and Simpson method to optimize). And Adam-Bashforth-Moulton method has a larger error term than Milne Simpson.

4 Extra

4.1 Power Series Method

4.1.1 Analysis of the IVP

Use power series to represent y

$$y(t) = \Sigma_{n=0}^{\infty} a_n (t - t_0)^n$$

Because in both cases $t_0 = 0$, so for simple,

$$y = \Sigma_{n=0}^{\infty} a_n t^n$$

Then we can get

$$\begin{aligned} \frac{\mathrm{d}y}{\mathrm{d}t} &= \Sigma_{n=1}^{\infty} n a_n t^{n-1} \\ &= \Sigma_{n=0}^{\infty} (n+1) a_{n+1} t^n \end{aligned}$$

$$\begin{aligned} y^2 &= (\Sigma_{n=0}^{\infty} a_n t^n)^2 = \Sigma_{n=0}^{\infty} (\Sigma_{m=0}^{\infty} a_m t^m) \cdot a_n t^n \\ &= \Sigma_{n=0}^{\infty} (\Sigma_{i=0}^{n} a_i a_{n-i}) \cdot t^n \end{aligned}$$

$$\begin{aligned} y^3 &= (\Sigma_{n=0}^{\infty} a_n t^n)^3 \\ &= \Sigma_{n=0}^{\infty} (\Sigma_{i=0}^{n} (\Sigma_{j=0}^{i} a_{i-j} a_j) \cdot a_{n-i}) \cdot t^n \end{aligned}$$

Let

$$b_n = \Sigma_{i=0}^{n} a_i a_{n-i} \ , \quad c_n = \Sigma_{i=0}^{n} (\Sigma_{j=0}^{i} a_{i-j} a_j) \cdot a_{n-i}$$

So that

$$y^2 = \Sigma_{n=0}^{\infty} b_n t^n \ , \quad y^3 = \Sigma_{n=0}^{\infty} c_n t^n$$

Substitute these power series into $\frac{\mathrm{d}y}{\mathrm{d}t} = (y - t^2)(y^2 - t) = y^3 - t^2 y^2 - ty + t^3$, we get

$$\begin{aligned} \Sigma_{n=0}^{\infty}(n+1)a_{n+1}t^n &= \Sigma_{n=0}^{\infty} c_n t^n - t^2 \cdot \Sigma_{n=0}^{\infty} b_n t^n - t \cdot \Sigma_{n=0}^{\infty} a_n t^n + t^3 \\ &= \Sigma_{n=0}^{\infty} c_n t^n - \Sigma_{n=2}^{\infty} b_{n-2} t^n - \Sigma_{n=1}^{\infty} a_{n-1} t^n + t^3 \end{aligned}$$

Let us assume for $n < 0$, $b_n = c_n = 0$ So we can get the recursion formulas of a_n, b_n, c_n

$$(n+1)a_{n+1} = \begin{cases} c_n - b_{n-2} - a_{n-1}, & n \neq 3 \\ c_3 - b_1 - a_2 + 1, & n = 3 \end{cases}$$

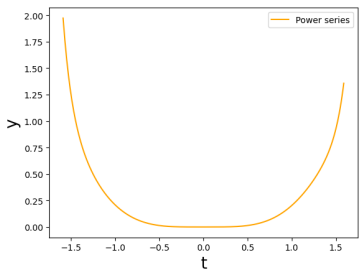
Thus we can get a_n and then calculate $y(t)$
For IVP1, $y(0) = 0$

$$y(t) = 0.25t^4 - 0.041667t^6 + 0.000521t^8 - 0.000521t^{10} \ldots$$

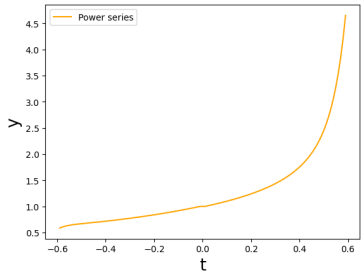
For IVP2, $y(0) = 1$

$$y(t) = 1 + t + t^2 + 1.333333t^3 + 2.25t^4 + 3.283333t^5 + 5.238889t^6 \ldots$$

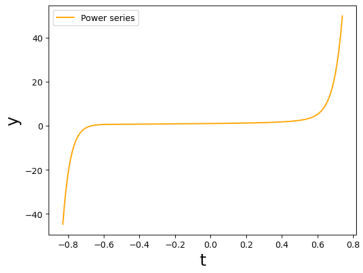
Figure 10: Power series method



(a) Power series method for IVP1



(b) Power series method for IVP2



(c) Power series method for IVP2, out-range

4.1.2 Error Analysis

The two graphs looks good in their region. The radius of convergence is also good. Take IVP2 as example,

$$\rho = 1/(\lim_{n \to \infty} \mid a_{n+1} \mid / \mid a_n \mid) = 0.582869$$

which is quite close to the asymptote.
However, if we plot the graph in a wider region We will see it has large error. So the power series method only apply in a small region close to $t = 0$.

5 Qualitative Analysis

5.1 Asymptote Analysis

As we can see from Fig.6(b), the value of y becomes extremely large when $t \to 0.6$, so we guess there is an asymptote around $t = 0.6$, say $t = t_0$To find it, let us look back:

$$Euqation \quad (2) \Rightarrow \frac{y'}{y^3} = 1 - \frac{t^2}{y} - \frac{t}{y^2} + \frac{t^3}{y^3}$$

By taking the limit of both sides and considering that y becomes extremely large, we obtain

$$\lim_{t\to t_0}\frac{y'}{y^3}=\lim_{t\to t_0}(1-\frac{t^2}{y}-\frac{t}{y^2}+\frac{t^3}{y^3})=1$$

so we get $\frac{\mathrm{d}y}{\mathrm{d}t}\approx y^3 \quad (*)$ as $t\to t_0$. And we can easily solve $(*)$, which is $y=\frac{1}{\sqrt{2C-2t}}$ for some constant C. By taking some value that has been proved to be as accurate as possible near $t=t_0$, such as $(t,y)=(0.58,13.87)$ from RK method, we can get $C\approx 0.5826$, which is quite close to the radius of convergence that we get in 4.1.2. So the asymptote can be estimated as

$$t=0.5826$$

In addition, we assert that there is no other asymptote for IVP2, and IVP1 Neither, which will be shown in detail in the next part.

5.2 Domain Analysis

For IVP1, we assert that the domain is $\mathbf{R}$. This can be shown intuitively through Fig.11 as on both sides the value of y keeps increasing as $|t|$ increases. And it could be verified by the following analysis. From the graph, it is easy to see that $y\approx t^2$ as t keeps decreasing below $t=0$. So it is reasonable to guess that $y=\alpha t^2$ and $\alpha\to 1$as $t\to-\infty$. By taking $y=\alpha t^2$ into (2) and take the limit, we get

$$\lim_{t\to-\infty}\frac{\alpha-1}{2\alpha}=\lim_{t\to-\infty}\frac{1}{\alpha^2t^5-t^2}=0$$

, so we get $\alpha=1$ as $t\to-\infty$, which meets our expectation. Similarly, we can get $y=\sqrt{t}$ as $t\to+\infty$. So we can get the conclusion that for IVP1, there is no asymptote at all, which indicates the domain is $\mathbf{R}$.

For IVP2, similarly, we get $y=t^2$ as $t\to-\infty$, so there is no other asymptote at the left side. And because our estimation has been interrupted by the asymptote $t=0.5826$, there is no way for us to move right forward. So the domain for IVP2 is $(-\infty,t_0]$(with t_0 estimated as 0.5826).

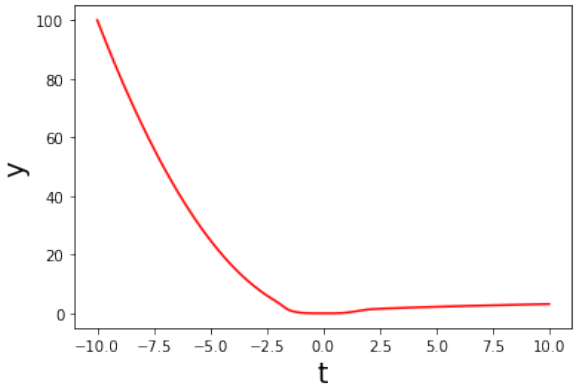


Figure 11: IVP1, for a wide range of t

5.3 Special Case Approximation

From 5.2, we see that we can use some simple functions to estimate y as $|t|\to\infty$. Now we try to use the numerical methods above to see if this fact is reasonable, as illustrated in Table 1. Note that we use the value computed by RK method as the accurate y value.

From Tabel 1 we can see that for IVP1, when $|t|>2$, the value of y can be estimated by two simple functions $y=t^2$ and $y=\sqrt{t}$ respectively with the percentage error less than 1%. And for $|t|<2$(or more accurately, $[-2,1]$), it can be estimated by $y=\frac{1}{4}t^4$. So the answer for IVP1 can be estimated by a segmented function consisting of three simple functions, the formula can be written as (4) and the result is plotted at Fig.13

$$y = \begin{cases} t^2, & t \leq -2 \\ \frac{1}{4}t^4, & -2 < t < 1.5 \\ \sqrt{t}, & t \geq 0.5 \end{cases} \tag{3}$$

Table 1: IVP1, differences between accurate value and $y = t^2$ or $y = \sqrt{t}$

t	t^2	$\sqrt{t}$	accurate y	Δ	error%
-3	8.9994	x	8.9259	0.07347	0.823%
-4	15.9992	x	15.9682	0.03097	0.194%
-5	24.9990	x	24.9830	0.01590	0.063%
-6	35.9999	x	35.9907	0.00922	0.0256%
-7	48.9995	x	48.9941	0.00581	0.011%
-8	63.9999	x	63.9961	0.00389	0.006%
-9	80.9999	x	80.9972	0.00274	0.003%
3	x	1.7320	1.7199	0.01206	0.701%
4	x	1.9999	1.9954	0.00453	0.227%
5	x	2.2360	2.2338	0.00221	0.098%
6	x	2.4494	2.4482	0.00124	0.05%
7	x	2.6455	2.6449	0.00077	0.029%
8	x	2.8284	2.8279	0.00051	0.018%
9	x	2.9999	2.9996	0.00035	0.011%

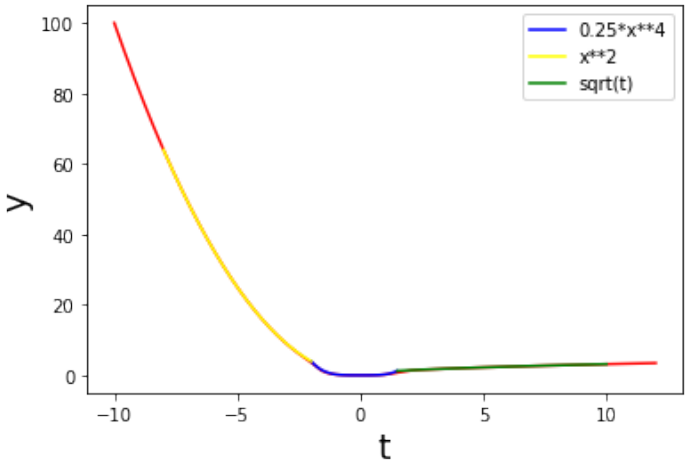


Figure 12: IVP1, $y = \frac{1}{4}t^4$ or $\sqrt{t}$ or t^2 VS $y(t)$

5.4 Further observation

Finally, we choose different initial values to draw the vector field and the graphs(Fig.14). We can see that for different IVPs, the graphs are just as similar as IVP1 or IVP2, and follows the guess in section 5.3, i.e., when $|t|$ is large, $y(t)$ can be estimated by $y = t^2$ and $y = \sqrt{t}$, respectively.

6 Conclusion

By doing this lab exercise, all of the group members learned some numerical methods to solve ODEs, including the comparison among methods, which will be helpful for our future study.

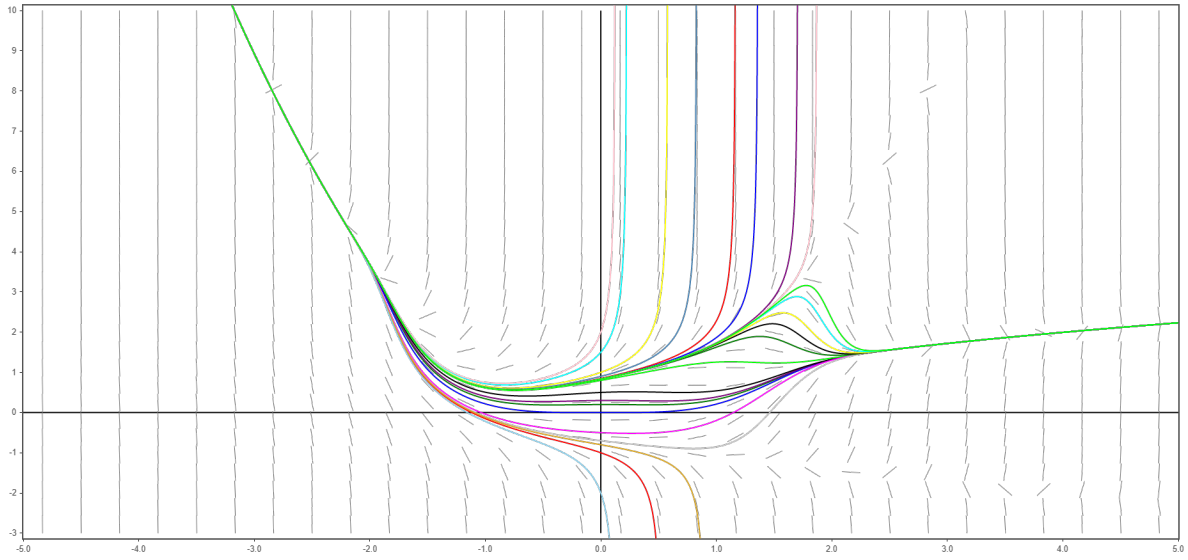


Figure 13: Vevtor field and more graphs for different initial values

References

[BDM21] William E Boyce, Richard C DiPrima, and Douglas B Meade. *Elementary differential equations and boundary value problems*. John Wiley & Sons, 2021.

MATH285 Lab Code

Group Name: MATH285gg

Group Member and student ID:

In [ ]:
```
import math
from matplotlib import pyplot as plt
import numpy as np

import sympy as sp
```

In [ ]:
```
def func(t, y):
    return (y - t ** 2) * (y ** 2 - t)
    # return y              # for test
    # return math.cos(t)    # for test
```

In [ ]:
```
class NumericalSols:
    def __init__(self, y_0, step, lower_bound, upper_bound, method_num):
        self.t_0 = 0
        self.y_0 = y_0
        assert(lower_bound <= 0 and upper_bound >= 0)
        self.lower_bound = lower_bound
        self.upper_bound = upper_bound
        self.step = step
        self.t_all = []
        self.y_all = []
        self.method_num = method_num
        if method_num == 0:
            self.eulerMethod()
        elif method_num == 1:
            self.impEuler()
        elif method_num == 2:
            self.runge4thMethod()
        elif method_num == 3:
            self.ABM()
        elif method_num == 4:
            self.MS()
        elif method_num == 5:
            self.Hamming()
        elif method_num == 6:
            self.powerSeries()
        else:
            raise NotImplementedError

    # method_num: 0
    def eulerMethod(self):
        # less than 0
        i = 1
        y_n = t_n = 0
        while t_n >= self.lower_bound:
            if i == 1:
```

```
                i = 0
                y_n = self.y_0
                t_n = self.t_0
            self.y_all.insert(0, y_n)
            self.t_all.insert(0, t_n)
            y_n = y_n + func(t_n, y_n) * (-self.step)
            t_n = t_n + (-self.step)

        # larger than 0
        i = 1
        y_n = t_n = 0
        while t_n <= self.upper_bound:
            if i == 1:
                i = 0
                y_n = self.y_0
                t_n = self.t_0
            self.y_all.append(y_n)
            self.t_all.append(t_n)

            y_n = y_n + func(t_n, y_n) * self.step
            t_n = t_n + self.step

    # 1
    def impEuler(self):
        i = 1
        y_n = t_n = 0
        while t_n >= self.lower_bound:
            if i == 1:
                i = 0
                y_n = self.y_0
                t_n = self.t_0
            self.y_all.insert(0, y_n)
            self.t_all.insert(0, t_n)
            y_n_euler = y_n + func(t_n, y_n) * (-self.step)    # improved from Eule
            y_n = y_n + 0.5 * (-self.step) * (func(t_n, y_n) + func(t_n + (-self.st
            t_n = t_n + (-self.step)

        i = 1
        y_n = t_n = 0
        while t_n <= self.upper_bound:
            if i == 1:
                i = 0
                y_n = self.y_0
                t_n = self.t_0
            self.y_all.append(y_n)
            self.t_all.append(t_n)

            y_n_euler = y_n + func(t_n, y_n) * self.step    # improved from Euler Me
            y_n = y_n + 0.5 * self.step * (func(t_n, y_n) + func(t_n + self.step, y
            t_n = t_n + self.step

    # 2
    def runge4thMethod(self):
        i = 1
        y_n = t_n = 0
        while t_n >= self.lower_bound:
            if i == 1:
                i = 0
                y_n = self.y_0
                t_n = self.t_0
            self.y_all.insert(0, y_n)
            self.t_all.insert(0, t_n)
            k1 = self.step * func(y_n, t_n)
            k2 = self.step * func(y_n + 0.5 * k1, t_n + 0.5 * self.step)
```

```
            k3 = self.step * func(y_n + 0.5 * k2, t_n + 0.5 * self.step)
            k4 = self.step * func(y_n + k3, t_n + self.step)
            y_n = y_n - (k1 + 2 * k2 + 2 * k3 + k4) / 6.0
            t_n = t_n + (-self.step)

        i = 1
        y_n = t_n = 0
        while t_n <= self.upper_bound:
            if i == 1:
                i = 0
                y_n = self.y_0
                t_n = self.t_0
            self.y_all.append(y_n)
            self.t_all.append(t_n)

            k1 = self.step * func(y_n, t_n)
            k2 = self.step * func(y_n + 0.5 * k1, t_n + 0.5 * self.step)
            k3 = self.step * func(y_n + 0.5 * k2, t_n + 0.5 * self.step)
            k4 = self.step * func(y_n + k3, t_n + self.step)
            y_n = y_n + (k1 + 2 * k2 + 2 * k3 + k4) / 6.0
            t_n = t_n + self.step

    ### method 3, 4, 5 are linear multiple step methods, and the code below just imp
    # 3
    def ABM(self):
        y_n = self.y_0
        t_n = self.t_0
        i=0
        while i<4:
            self.y_all.insert(0, y_n)
            self.t_all.insert(0, t_n)
            k1 = self.step * func(y_n, t_n)
            k2 = self.step * func(y_n + 0.5 * k1, t_n + 0.5 * self.step)
            k3 = self.step * func(y_n + 0.5 * k2, t_n + 0.5 * self.step)
            k4 = self.step * func(y_n + k3, t_n + self.step)
            y_n = y_n + (k1 + 2 * k2 + 2 * k3 + k4) / 6.0
            t_n = t_n + self.step
            i=i+1
        k=3
        while t_n <= self.upper_bound:
            t_0=self.t_all[k-3]
            t_1=self.t_all[k-2]
            t_2=self.t_all[k-1]
            t_3=self.t_all[k]
            y_0=self.y_all[k-3]
            y_1=self.y_all[k-2]
            y_2=self.y_all[k-1]
            y_3=self.y_all[k]
            dy=y_3+(self.step/24)*(-9*func(t_0,y_0)+37*func(t_1,y_1)-59*func(t_2,y_
            t_n=t_3+self.step
            y_n=y_3+(self.step/24)*(func(t_1,y_1)-5*func(t_2,y_2)+19*func(t_3,y_3)+
            if(abs(y_n)>5): break
            self.t_all.append(t_n)
            self.y_all.append(y_n)
            k=k+1

    # 4
    def MS(self):
        y_n = self.y_0
        t_n = self.t_0
        i=0
        while i<4:
            self.y_all.insert(0, y_n)
            self.t_all.insert(0, t_n)
```

```
            k1 = self.step * func(y_n, t_n)
            k2 = self.step * func(y_n + 0.5 * k1, t_n + 0.5 * self.step)
            k3 = self.step * func(y_n + 0.5 * k2, t_n + 0.5 * self.step)
            k4 = self.step * func(y_n + k3, t_n + self.step)
            y_n = y_n + (k1 + 2 * k2 + 2 * k3 + k4) / 6.0
            t_n = t_n + self.step
            i=i+1
        k=3
        while (t_n <= self.upper_bound):
            t_0=self.t_all[k-3]
            t_1=self.t_all[k-2]
            t_2=self.t_all[k-1]
            t_3=self.t_all[k]
            y_0=self.y_all[k-3]
            y_1=self.y_all[k-2]
            y_2=self.y_all[k-1]
            y_3=self.y_all[k]
            dy=y_0+4*(self.step/3)*(2*func(t_1,y_1)-func(t_2,y_2)+2*func(t_3,y_3))
            t_n=t_3+self.step
            y_n=y_2+(self.step/3)*(func(t_2,y_2)+4*func(t_3,y_3)+func(t_n,dy))
            if(abs(y_n)>5): break
            self.t_all.append(t_n)
            self.y_all.append(y_n)
            k=k+1

    # 5
    def Hamming(self):
        y_n = self.y_0
        t_n = self.t_0
        i=0
        while i<4:
            self.y_all.insert(0, y_n)
            self.t_all.insert(0, t_n)
            k1 = self.step * func(y_n, t_n)
            k2 = self.step * func(y_n + 0.5 * k1, t_n + 0.5 * self.step)
            k3 = self.step * func(y_n + 0.5 * k2, t_n + 0.5 * self.step)
            k4 = self.step * func(y_n + k3, t_n + self.step)
            y_n = y_n + (k1 + 2 * k2 + 2 * k3 + k4) / 6.0
            t_n = t_n + self.step
            i=i+1
        k=3
        while (t_n <= self.upper_bound):
            t_0=self.t_all[k-3]
            t_1=self.t_all[k-2]
            t_2=self.t_all[k-1]
            t_3=self.t_all[k]
            y_0=self.y_all[k-3]
            y_1=self.y_all[k-2]
            y_2=self.y_all[k-1]
            y_3=self.y_all[k]
            dy=y_0+4*(self.step/3)*(2*func(t_1,y_1)-func(t_2,y_2)+2*func(t_3,y_3))
            t_n=t_3+self.step
            y_n=-y_1/8+9*y_3/8+3*(self.step/8)*(-func(t_2,y_2)+2*func(t_3,y_3)+fun
            if(abs(y_n)>5): break
            self.t_all.append(t_n)
            self.y_all.append(y_n)
            k=k+1

    # 6
    def powerSeries(self):
        # pass
        a = np.zeros(10000, dtype=float)
        b = np.zeros(10000, dtype=float)
        c = np.zeros(10000, dtype=float)
```

```
        n_pwr = 10
        # manully calculate
        a[0] = self.y_0
        self.help_bc(a, b, c, 0)
        a[1] = c[0]
        self.help_bc(a, b, c, 1)
        a[2] = (c[1]-a[0])/2
        self.help_bc(a, b, c, 2)
        a[3] = (c[2]-b[0]-a[1])/3
        self.help_bc(a, b, c, 3)
        a[4] = (c[3]-b[1]-a[2]+1)/4

        for i in range(4,n_pwr):
            self.help_bc(a, b, c, i)
            a[i+1] = (c[i]-b[i-2]-a[i-1])/(i+1)

        for i in range(n_pwr):
            print(i, "a=", a[i], "b=", b[i], "c=", c[i])
        self.help_radius(a, n_pwr)

        self.help_PwrSrs(a, b, c, -1, n_pwr)
        self.help_PwrSrs(a, b, c, 1, n_pwr)

    def help_bc(self, a, b, c, i):
        for j in range(i+1):
            b[i] += a[j]*a[i-j]
        for j in range(i+1):
            c[i] += a[j]*b[i-j]

    def help_PwrSrs(self, a, b, c, dir, n):
        i = 1
        y_n = t_n = 0
        while (t_n>=self.lower_bound and t_n<=self.upper_bound):
            if i == 1:
                i = 0
                y_n = self.y_0
                t_n = self.t_0

            if(dir==1):
                self.y_all.append(y_n)
                self.t_all.append(t_n)
            elif(dir==-1):
                self.y_all.insert(0, y_n)
                self.t_all.insert(0, t_n)

            y_n = 0
            for i in range(n):
                y_n += a[i] * (t_n ** i)
            if(abs(y_n)>50): break

            t_n = t_n + dir* self.step

    def help_radius(self, a, n):
        for i in range(n-1):
            # print(i, "r=", abs(a[i]/a[i+1]))
            pass

    # Draw
    def draw(self):
        plt.xlabel('t', fontsize=19)
        plt.ylabel('y', fontsize=19)
        plt.plot(self.t_all, self.y_all, c="red")
```

```
    def drawlinear(self,n):
        plt.xlabel('t', fontsize=19)
        plt.ylabel('y', fontsize=19)
        if (n==3):
            plt.plot(self.t_all, self.y_all, c="red",label="ABM")
        if (n==4):
            plt.plot(self.t_all, self.y_all, c="blue", label="MS")
        if (n==5):
            plt.plot(self.t_all, self.y_all, c="green", label="Hamming")
```

In [ ]:
```
def testDiffMethods(y_0, step, lower_bound, upper_bound):
    """
    Test different methods according to the given parameters by drawing plots.
    """
    # eulerMethod
    euler_meth = NumericalSols(y_0, step, lower_bound, upper_bound, 0)
    plt.plot(euler_meth.t_all, euler_meth.y_all, c="green", label="Euler method")

    # impEuler
    im_eu_meth = NumericalSols(y_0, step, lower_bound, upper_bound, 1)
    plt.plot(im_eu_meth.t_all, im_eu_meth.y_all, c="red", label="Improve Euler metho

    # runge4thMethod
    runge_meth = NumericalSols(y_0, step, lower_bound, upper_bound, 2)
    plt.plot(runge_meth.t_all, runge_meth.y_all, c="blue", label="Runge 4th order")

    # ABM
    mulstep_meth = NumericalSols(y_0, step, lower_bound, upper_bound, 3)
    plt.plot(mulstep_meth.t_all, mulstep_meth.y_all, c="brown", label="ABM")

    # MS
    mulstep_meth = NumericalSols(y_0, step, lower_bound, upper_bound, 4)
    plt.plot(mulstep_meth.t_all, mulstep_meth.y_all, c="black", label="MS")

    # Hamming
    mulstep_meth = NumericalSols(y_0, step, lower_bound, upper_bound, 5)
    plt.plot(mulstep_meth.t_all, mulstep_meth.y_all, c="purple", label="Hamming")


    # powerSeries method
    power_meth = NumericalSols(y_0, step, lower_bound, upper_bound, 6)
    plt.plot(power_meth.t_all, power_meth.y_all, c="orange", label="Power series")
```

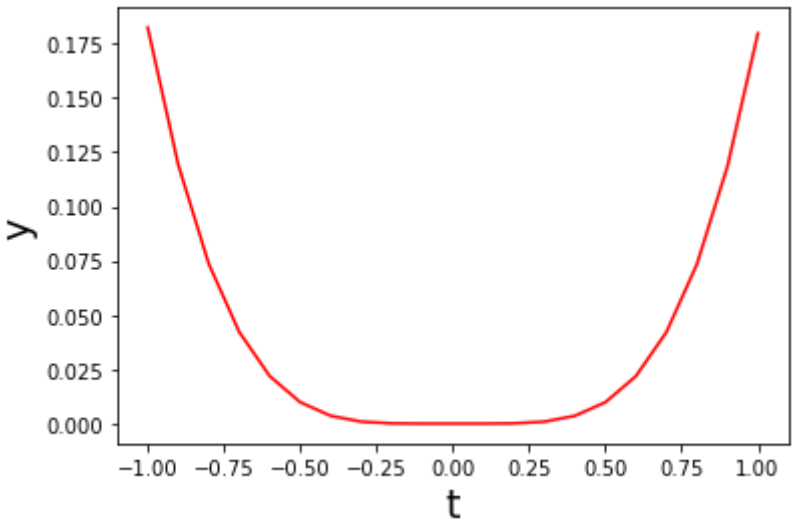
In [ ]:
```
def testDiffSteps(y_0, lower_bound, upper_bound, method_num):
    """
    Test one method with different steps by drawing plots.
    """
    # step == 0.2
    meth1 = NumericalSols(y_0, 0.2, lower_bound, upper_bound, method_num)
    plt.plot(meth1.t_all, meth1.y_all, c="green", label="step=0.2")

    # step == 0.1
    meth2 = NumericalSols(y_0, 0.1, lower_bound, upper_bound, method_num)
    plt.plot(meth2.t_all, meth2.y_all, c="red", label="step=0.1")

    # step == 0.05
    meth3 = NumericalSols(y_0, 0.05, lower_bound, upper_bound, method_num)
    plt.plot(meth3.t_all, meth3.y_all, c="blue", label="step=0.05")
```

In [ ]:
```
def plotDiffMethods(y_0, step, lower_bound, upper_bound):
    # fig=plt.figure(num=1,figsize=(8,6))
```

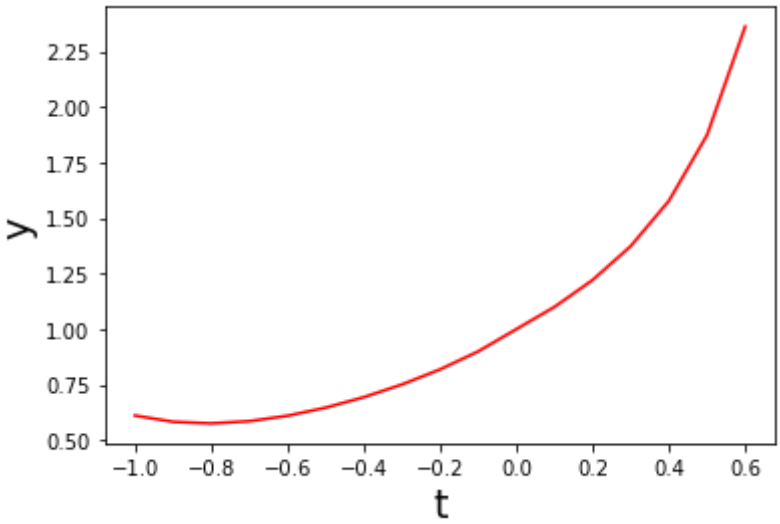
```
    plt.xlabel('t', fontsize=19)
    plt.ylabel('y', fontsize=19)

    # ax1 = fig.add_subplot(311)
    # ax1.set_title("311")
    testDiffMethods(y_0, step, lower_bound, upper_bound)

    plt.legend()
    plt.show()
```

In [ ]:
```
def plotDiffSteps(y_0, lower_bound, upper_bound, method_num):
    plt.xlabel('t', fontsize=19)
    plt.ylabel('y', fontsize=19)

    testDiffSteps(y_0, lower_bound, upper_bound, method_num)

    plt.legend()
    plt.show()
    pass
```

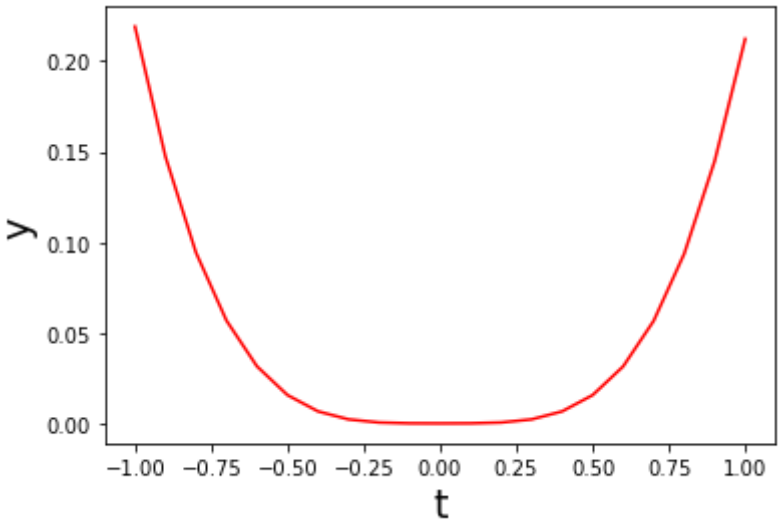
In [ ]:
```
# Euler Method with step = 0.1, IVP1
euler_1 = NumericalSols(0, 0.1, -1, 1, 0)
euler_1.draw()
```



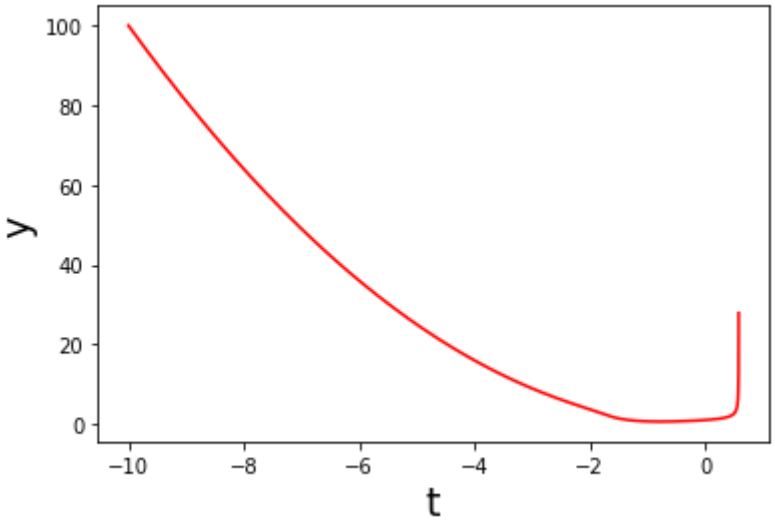
In [ ]:
```
# Euler Method with step = 0.1, IVP2
euler_2 = NumericalSols(1, 0.1, -1, 0.6, 0)
euler_2.draw()
```



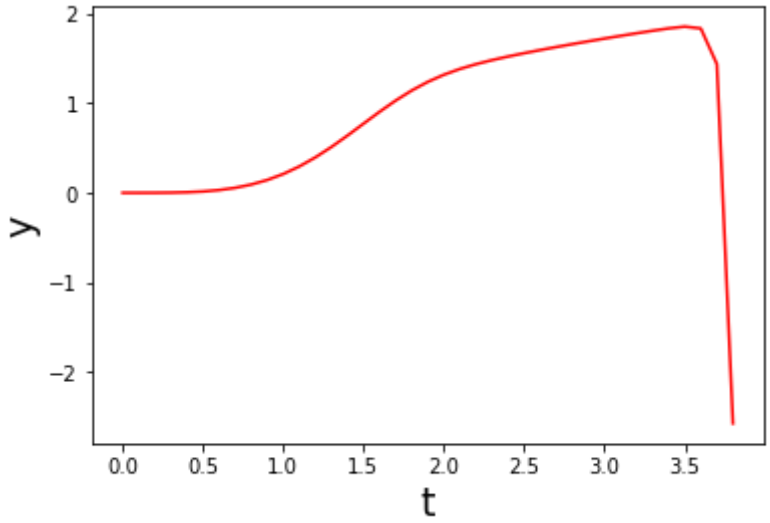
In [ ]:
```
# improve Euler Method with step = 0.1, IVP1
plt.xlabel('t', fontsize=19)
plt.ylabel('y', fontsize=19)
im_euler_1 = NumericalSols(0, 0.1, -1, 1, 1)
im_euler_1.draw()
```



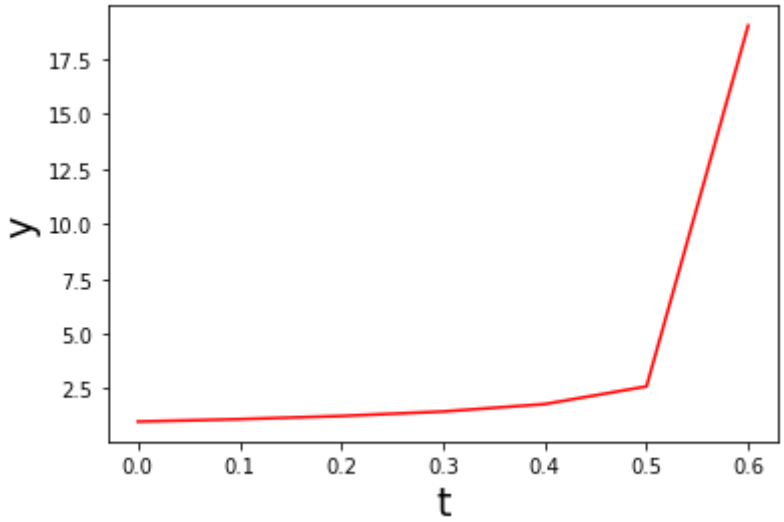
In [ ]:
```
# improve Euler Method with step = 0.1, IVP2
plt.xlabel('t', fontsize=19)
plt.ylabel('y', fontsize=19)
im_euler_2 = NumericalSols(1, 0.0001, -10, 0.582, 1)
im_euler_2.draw()
```



In [ ]:
```
plt.xlabel('t', fontsize=19)
plt.ylabel('y', fontsize=19)
Runge = NumericalSols(0, 0.1, 0, 3.9, 2)
Runge.draw()
```

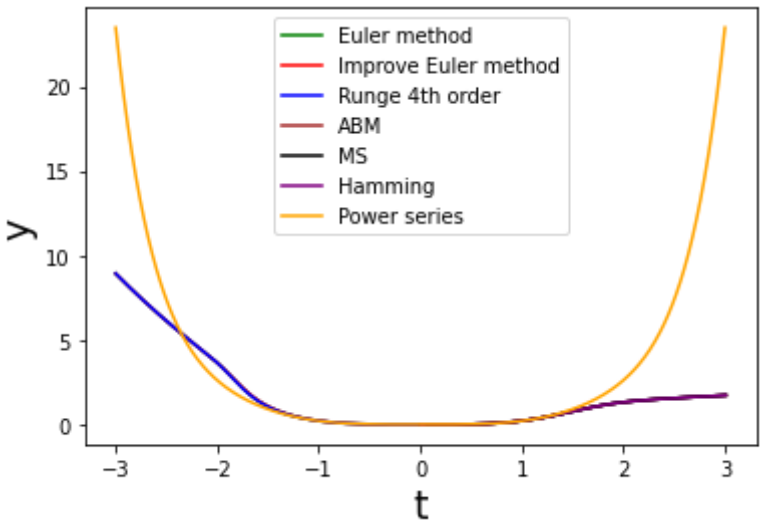


In [ ]:
```
plt.xlabel('t', fontsize=19)
plt.ylabel('y', fontsize=19)
Runge = NumericalSols(1, 0.1, 0, 0.6, 2)
Runge.draw()
```



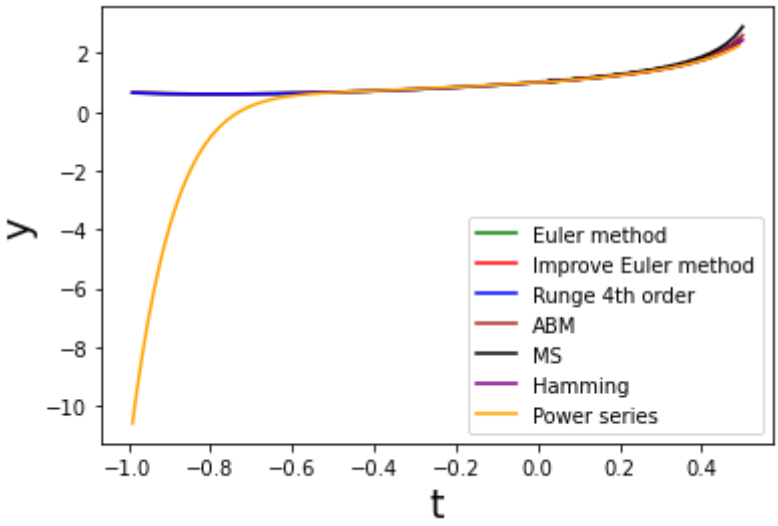
In [ ]:
```
# IVP1, compare methods with different steps
l_bound = -3
r_bound = 3
# plotDiffMethods(0, 0.1, l_bound, r_bound)  # step == 0.1
#plotDiffMethods(0, 0.05, l_bound, r_bound) # step == 0.05
plotDiffMethods(0, 0.01, l_bound, r_bound) # step == 0.01
```

```
0 a= 0.0 b= 0.0 c= 0.0
1 a= 0.0 b= 0.0 c= 0.0
2 a= 0.0 b= 0.0 c= 0.0
3 a= 0.0 b= 0.0 c= 0.0
4 a= 0.25 b= 0.0 c= 0.0
5 a= 0.0 b= 0.0 c= 0.0
6 a= -0.041666666666666664 b= 0.0 c= 0.0
7 a= 0.0 b= 0.0 c= 0.0
8 a= 0.005208333333333333 b= 0.0625 c= 0.0
9 a= 0.0 b= 0.0 c= 0.0
```

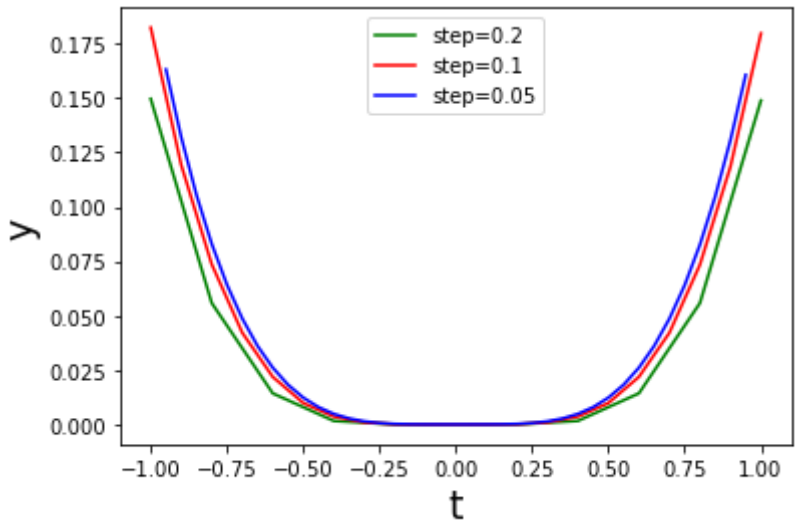


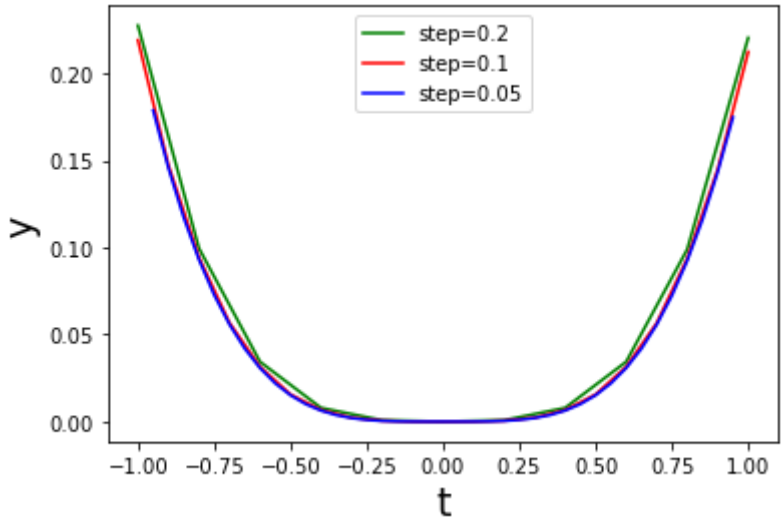
In [ ]:
```
# IVP2, compare methods with different steps
l_bound = -1
r_bound = 0.5
# plotDiffMethods(1, 0.1, l_bound, r_bound)  # step == 0.1
# plotDiffMethods(1, 0.05, l_bound, r_bound) # step == 0.05
plotDiffMethods(1, 0.01, l_bound, r_bound) # step == 0.01
```

```
0 a= 1.0 b= 1.0 c= 1.0
1 a= 1.0 b= 2.0 c= 3.0
2 a= 1.0 b= 3.0 c= 6.0
3 a= 1.3333333333333333 b= 4.666666666666666 c= 11.0
4 a= 2.25 b= 8.166666666666666 c= 20.75
5 a= 3.2833333333333337 b= 13.733333333333333 c= 38.349999999999994
6 a= 5.238888888888888 b= 23.322222222222223 c= 70.0
7 a= 8.364285714285716 b= 39.77301587301588 c= 126.90952380952382
8 a= 13.492162698412699 b= 68.00873015873016 c= 229.0497023809524
9 a= 21.929243827160494 b= 116.31675485008819 c= 411.36326058201064
```

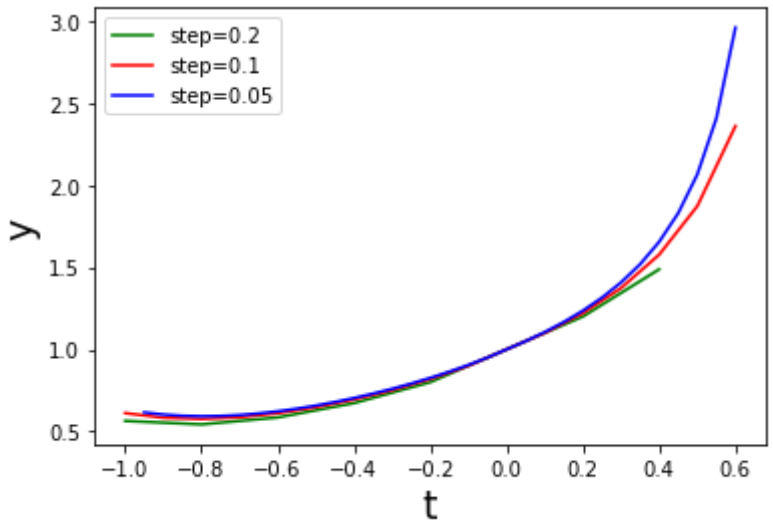


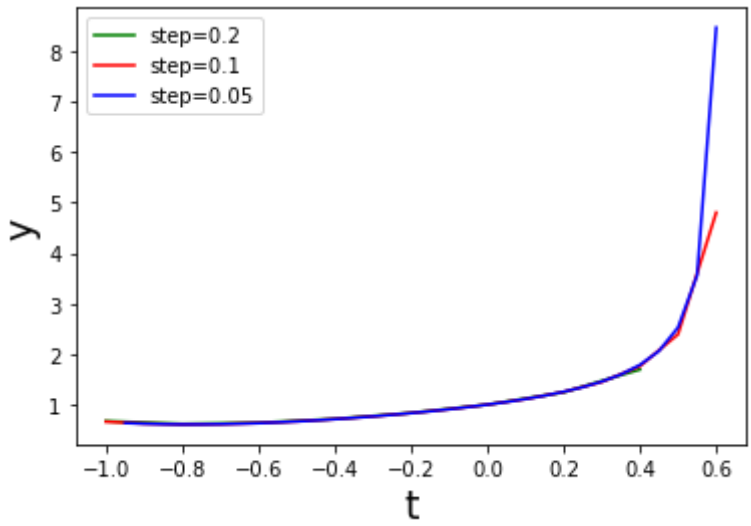
In [ ]:
```
# IVP1, compare one method with different steps
plotDiffSteps(0, -1, 1, 0) # Euler
plotDiffSteps(0, -1, 1, 1) # improve Euler
                           # others
```



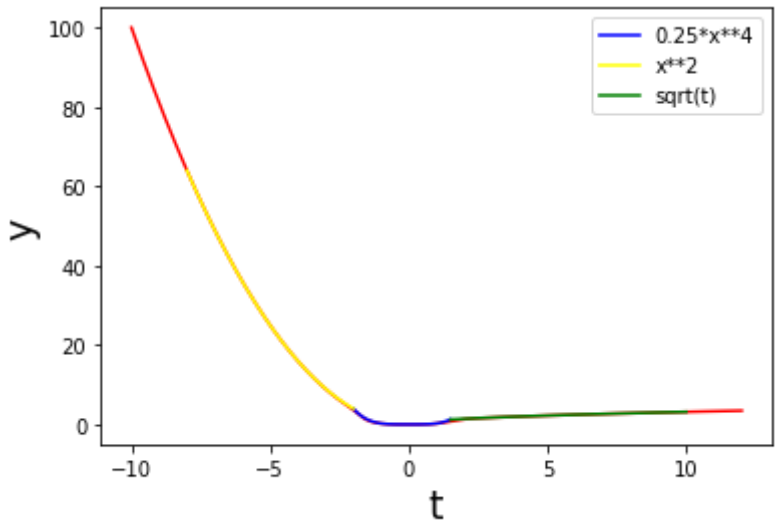


In [ ]:
```
# IVP2, compare one method with different steps
plotDiffSteps(1, -1, 0.6, 0) # Euler
plotDiffSteps(1, -1, 0.6, 1) # improve Euler
                             # others
```





In [ ]:
```
# IVP1 domain test

plt.xlabel('t', fontsize=19)
plt.ylabel('y', fontsize=19)
domain_test = NumericalSols(0, 0.0001, -10, 12, 1)
domain_test.draw()
# y1_list = [t**2 for t in domain_test.t_all]
# plt.plot(domain_test.t_all, y1_list, c="green")
t_c = np.linspace(-2, 1.5, 100)
yc_list = [0.25*t**4 for t in t_c]
plt.plot(t_c, yc_list, c="blue",label="0.25*x**4")
t_l = np.linspace(-8, -2, 100)
y1_list = [t**2 for t in t_l]
plt.plot(t_l, y1_list, c="yellow",label="x**2")
t_r = np.linspace(1.5, 10, 100)
yr_list = [np.sqrt(t) for t in t_r]
plt.plot(t_r, yr_list, c="green", label="sqrt(t)")
plt.legend()
plt.show()
# delta_y = [abs(y1_list[i] - domain_test.y_all[i]) for i in range(len(y1_list))]
# plt.plot(linear_mul.t_all, delta_y, c="blue")
# print(j,'&', 'x',y1_list[-1],'&', domain_test.y_all[-1],'&','&', delta_y[-1], '&',
```



MATH285 LAB PRESENTATION

TEAM: MATH285gg

Jun Liang

Xinchen Yin

Ziming Yan

Xingjian Kang

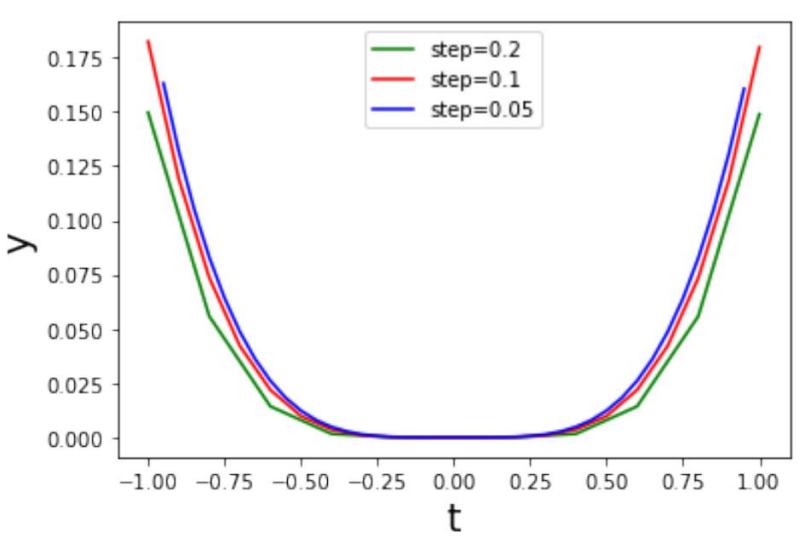
Yujie Pan

Outline

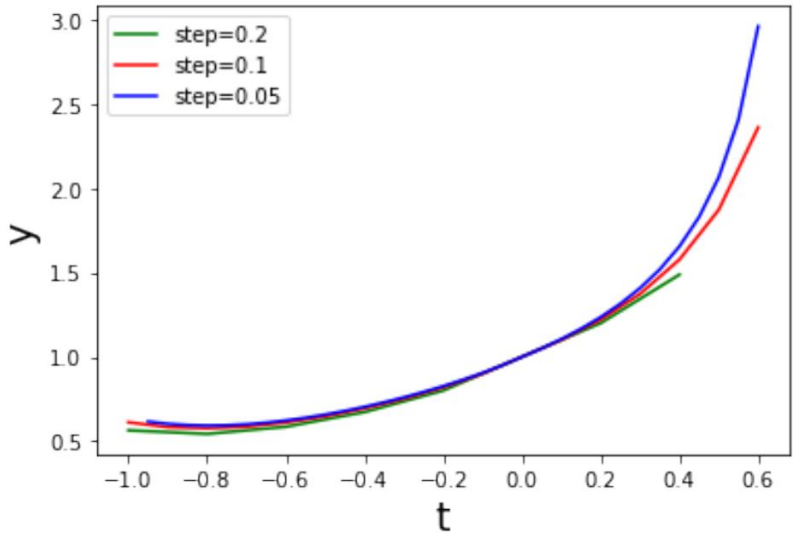
- Introduction
- Numerical Methods
  - Euler Method
  - Improved Euler Method
  - Runge-Kutta Method
  - Linear Multistep Methods
- Extra
  - Power Series Method
- Qualitative Analysis and further observation

Euler Method

- Fomula: $y_{k+1} = y_k + h * f(t_k, y_k)$ —> $y(t_n) \approx y_n = y_{n-1} + h * f(t_{n-1}, y_{n-1})$



(a) IVP1, Euler Method with several steps



(b) IVP2, Euler Method with several steps

- Error: the local truncation error is $|e_n| \leq \frac{1}{2}Mh^2$, *M* is the maximum of *y''* on the interval [*a, b*].
  So the global truncation error En is $E_n \leq Kh$, K is a constant.
  The global truncation error for Euler method is $\mathrm{O(h)}$.

Improved Euler Method

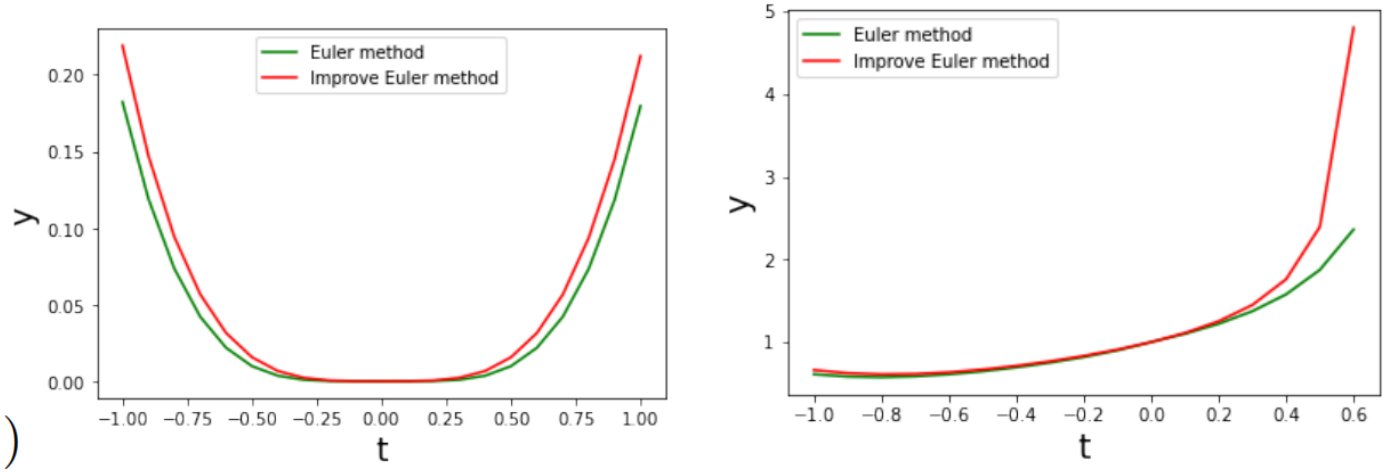
- Fomula:

$$y_{k+1} = y_k + \frac{h}{2} * f(t_k, y_k) + f(t_{k+1}, y_{k+1})$$

- Error: Similar to Euler Method, the global truncation error En of the improved Euler method is: $E_n \leq Kh^2$

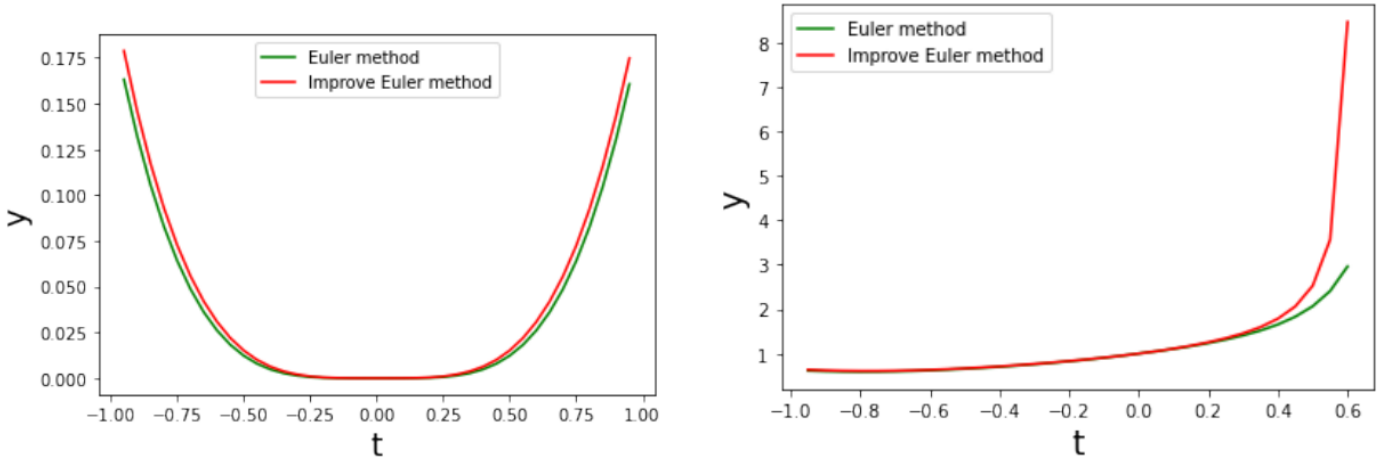
Where K is a constant. So, the global truncation error of improved Euler method is O(h^2), which is more accurate than Euler method.

Figure 2: Comparations between Euler and improve Euler, with step=0.1



(a) IVP1 (b) IVP2

Figure 3: Comparations between Euler and improve Euler, with step=0.05

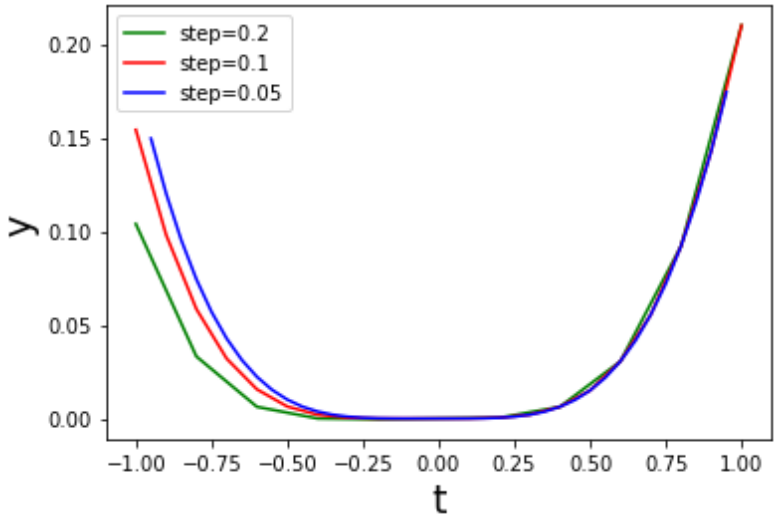


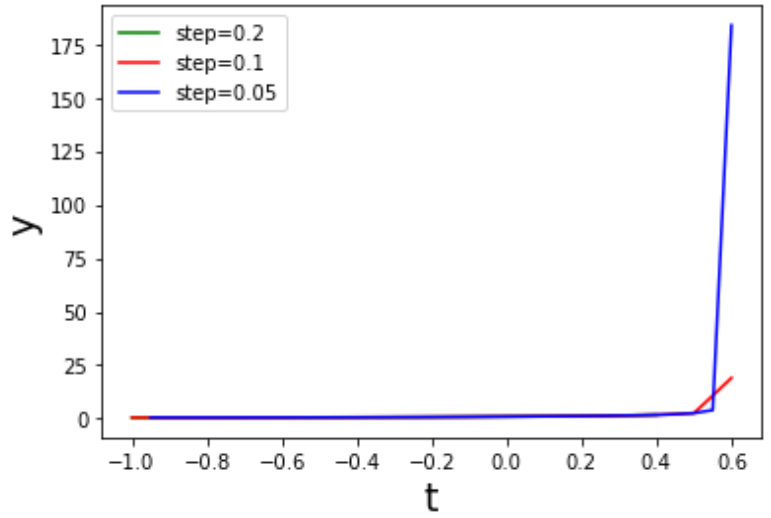
(a) IVP1 (b) IVP2

Runge-Kutta Method

- The most widely known member of the Runge–Kutta family is generally referred to as ”RK4”.
- Below is formula and the plot using different step length.

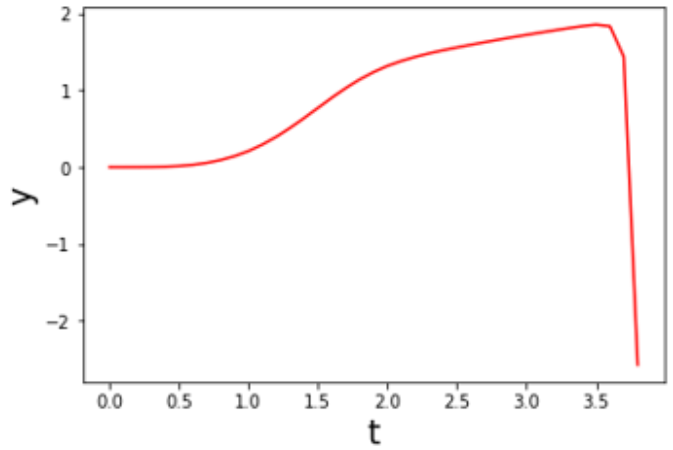
$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \ t_{n+1} = t_n + h$$

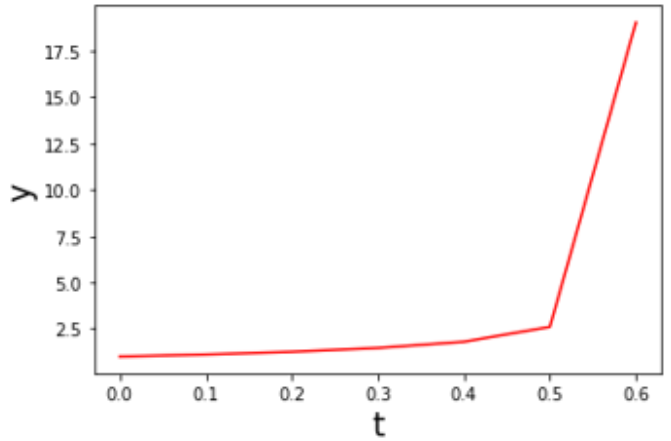


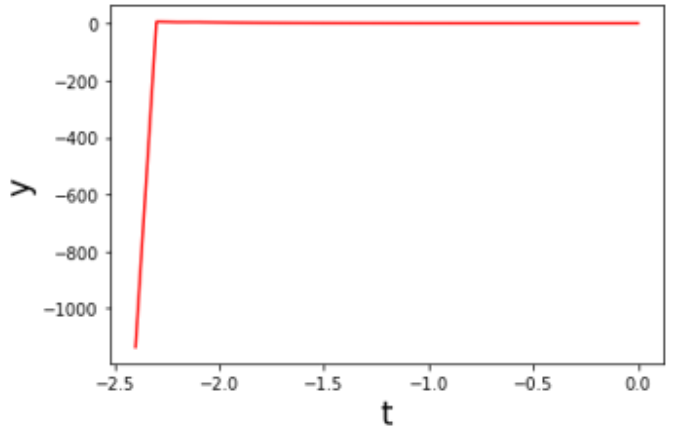


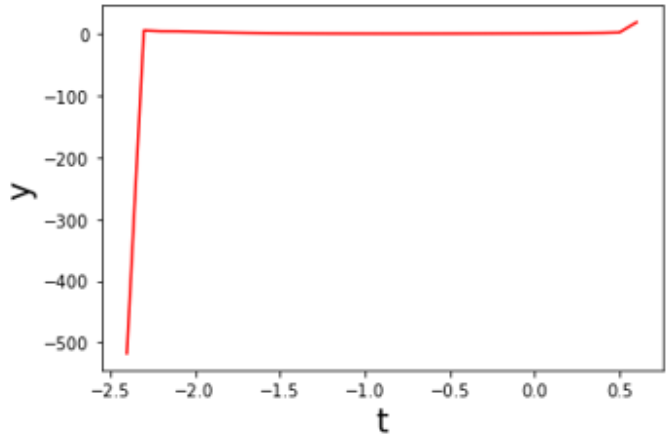
Runge-Kutta Method

- Left depicts the upper and lower limits plots using Runge-Kutta with step length 0.1 for IVP1, right are for IVP2.









Linear Multistep Methods

In a step-by-step approach to solving, a series of approximations to yn+1 has in fact been derived prior to the calculation of y0, y1...yn. If the information from the preceding multiple parts is fully utilised to predict yn+1, a high degree of accuracy can be expected. This is the basic idea behind the construction of the resulting linear multi-step method.

General formula is

$$y_{n+k} = \sum_{i=0}^{k-1} a_i y_i + h \sum i = 0^k b_i f_{n+i} (i = 0, 1..., k-2)$$

As shown, we need to find first k points to comput k+1 point. To get more accurate data, I select to use linear 4th-step method. And there are three formulars to accomplish it:

1.Adams-Bashforth-Moulton method

2.Milne-Simpson method

3.Hamming method

Formulate

- Predictors
- Adams-bashforth

$$y'_{k+1} = y_k + \int_{t_{k+1}}^{t_k} f(t,y) = y_k + \frac{h}{24}(55f_k - 59f_{k-1} + 37f_{k-2} - 9f_{k-3}$$

- Milne

$$y'_{k+1} = y_{k-3} + \int_{t_{k+1}}^{k_{k-3}} f(t,y) = y_{k-3} + \frac{4h}{3}(2f_{k-2} - fk - 1 + 2f_k$$

- Hamming

$$y'_{k+1} = y_{k-3} + \frac{4h}{3}(2f_{k-2} - f_{k-1} + 2f_k)$$

- Baces
- Adams-Moulton

$$y_{k+1} = y_k + \int_{t_{k+1}}^{t_k} f(t,y) = y_k + \frac{h}{24}(9f_{k+1} - 19f_{k-1} - 5f_{k-1} + f_{k-2}$$

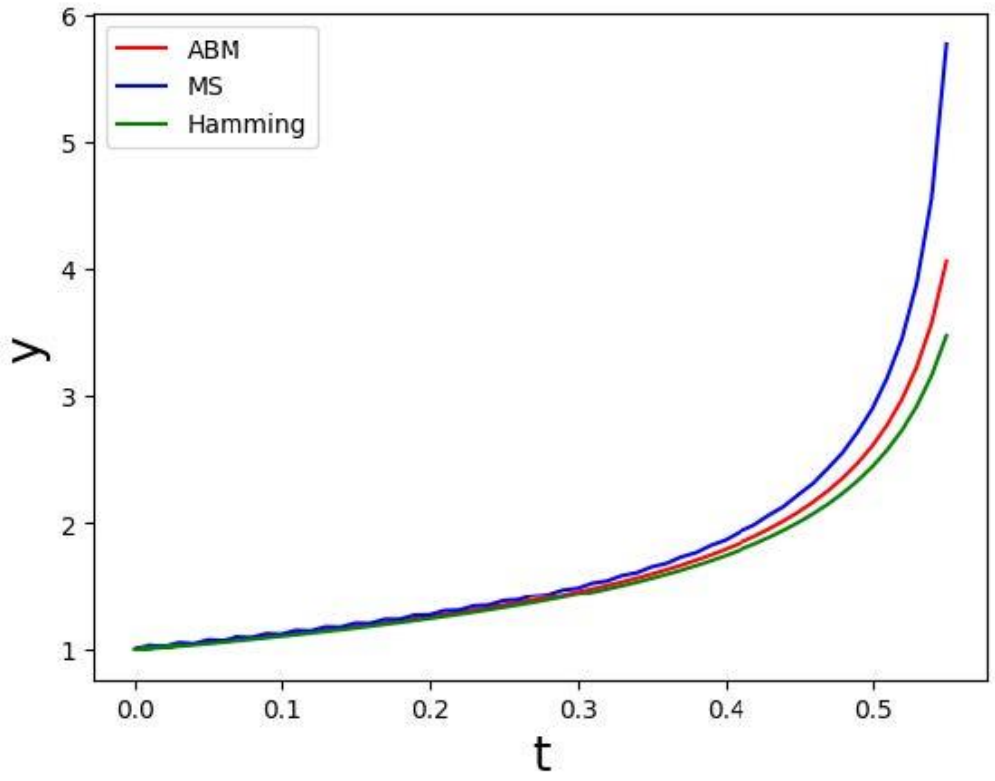
- Simpson

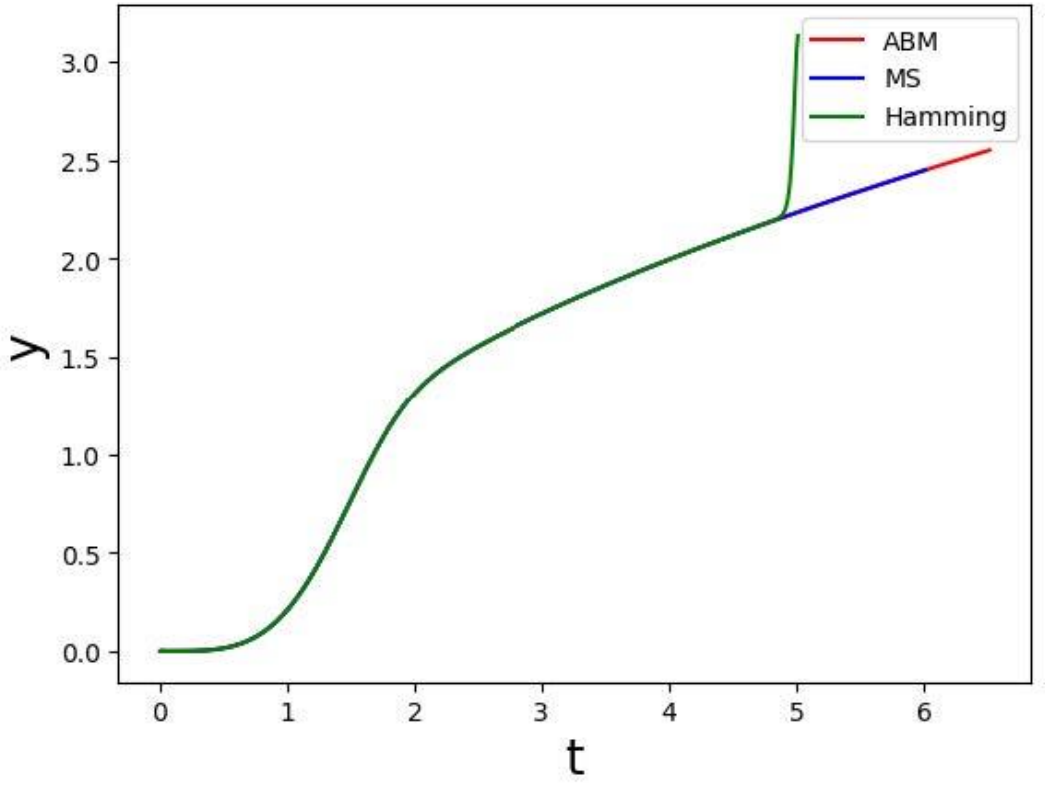
$$y_{k+1} = y_{k-1} + \int_{t_{k+1}}^{k_{k-3}} f(t,y) = y_{k-1} + \frac{h}{3}(f_{k-1} + 4f_k + f_{k+1})$$

- Hamming

$$y_{k+1} = \frac{-1}{8}y_{k-2} + \frac{9}{8}y_k + \frac{3h}{8}(-f_{k-1} + 2f_k + f_{k+1})$$

Consequence graph and comparation





For Adams-Bashforth-Moulton, the error is about $\frac{28}{90}h^5y^{(iv)}(\xi)$

For Milne-Simpson method, the error is about $\frac{251}{720}h^5y^{(iv)}(\xi)$

Power Series Method

$$y(t) = \Sigma_{n=0}^{\infty} a_n (t - t_0)^n$$

$$t_0 = 0$$

$$y = \Sigma_{n=0}^{\infty} a_n t^n$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = \Sigma_{n=1}^{\infty} n a_n t^{n-1}$$
$$= \Sigma_{n=0}^{\infty} (n+1) a_{n+1} t^n$$

$$y^2 = (\Sigma_{n=0}^{\infty} a_n t^n)^2 = \Sigma_{n=0}^{\infty} (\Sigma_{m=0}^{\infty} a_m t^m) \cdot a_n t^n$$
$$= \Sigma_{n=0}^{\infty} (\Sigma_{i=0}^{n} a_i a_{n-i}) \cdot t^n$$

$$y^3 = (\Sigma_{n=0}^{\infty} a_n t^n)^3$$
$$= \Sigma_{n=0}^{\infty} (\Sigma_{i=0}^{n} (\Sigma_{j=0}^{i} a_{i-j} a_j) \cdot a_{n-i}) \cdot t^n$$

Power Series Method

$$b_n = \Sigma_{i=0}^{n} a_i a_{n-i} \ , \quad c_n = \Sigma_{i=0}^{n} (\Sigma_{j=0}^{i} a_i$$

$$y^2 = \Sigma_{n=0}^{\infty} b_n t^n \ , \quad y^3 = \Sigma_{n=0}^{\infty} c$$

ute these power series into $\frac{\mathrm{d}y}{\mathrm{d}t} = (y - t^2)(y^2 - t) = y^3 - t$

$$\begin{aligned} \Sigma_{n=0}^{\infty}(n+1)a_{n+1}t^n &= \Sigma_{n=0}^{\infty} c_n t^n - t^2 \cdot \Sigma_{n=0}^{\infty} b_n t^r \\ &= \Sigma_{n=0}^{\infty} c_n t^n - \Sigma_{n=2}^{\infty} b_{n-2} t^n \end{aligned}$$

ssume for $n < 0$, $b_n = c_n = 0$ So we can get the recursi

$$(n+1)a_{n+1} = \begin{cases} c_n - b_{n-2} - a_{n-1}, \\ c_3 - b_1 - a_2 + 1, \ n = 3 \end{cases}$$

$$\begin{aligned} \frac{\mathrm{d}y}{\mathrm{d}t} &= \Sigma_{n=1}^{\infty} n a_n t^{n-1} \\ &= \Sigma_{n=0}^{\infty}(n+1)a_{n+1}t^n \end{aligned}$$

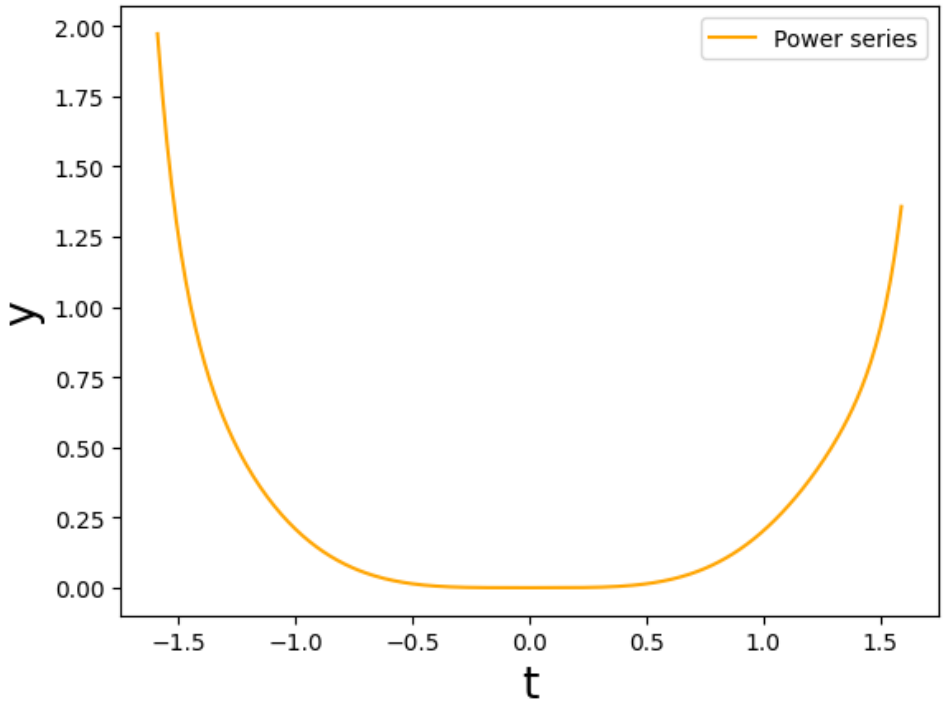
$$\begin{aligned} y^2 &= (\Sigma_{n=0}^{\infty} a_n t^n)^2 = \Sigma_{n=0}^{\infty}(\Sigma_{m=0}^{\infty} a_m t^m) \cdot a_n t^n \\ &= \Sigma_{n=0}^{\infty}(\Sigma_{i=0}^{n} a_i a_{n-i}) \cdot t^n \end{aligned}$$

$$\begin{aligned} y^3 &= (\Sigma_{n=0}^{\infty} a_n t^n)^3 \\ &= \Sigma_{n=0}^{\infty}(\Sigma_{i=0}^{n}(\Sigma_{j=0}^{i} a_{i-j} a_j) \cdot a_{n-i}) \cdot t^n \end{aligned}$$

Power Series Method

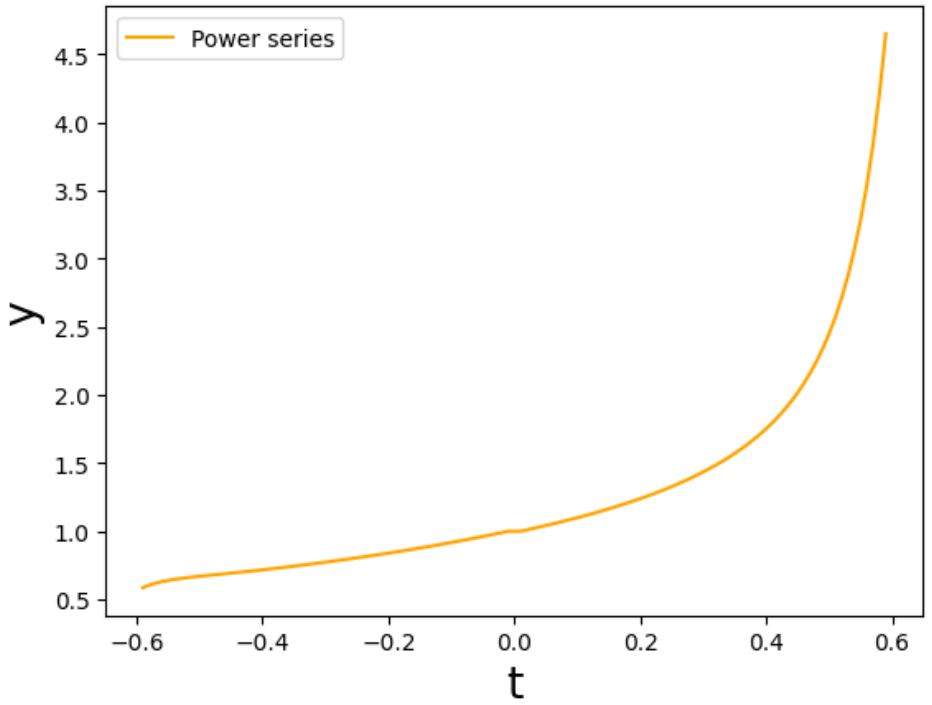
For IVP1, $y(0) = 0$

$$y(t) = 0.25t^4 - 0.041667t^6 + 0.000521t^8 - 0.000521t^{10} \ldots$$

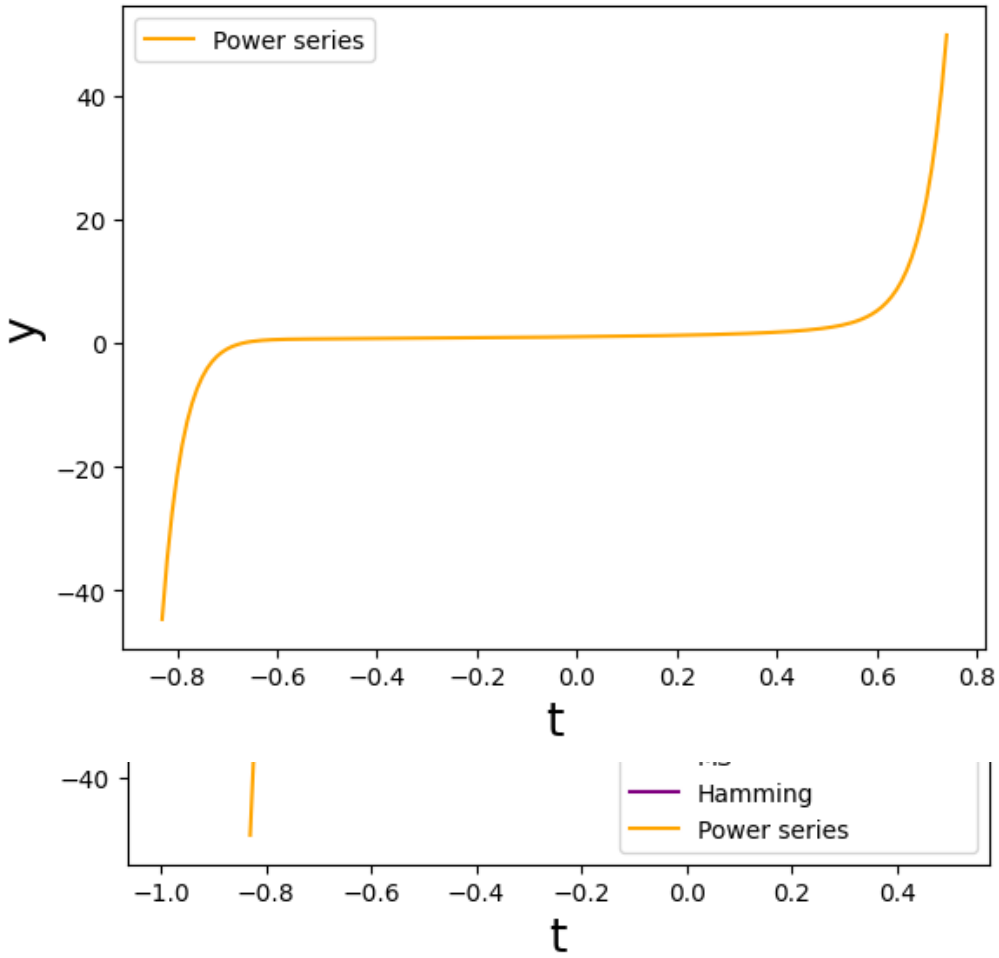


For IVP2, $y(0) = 1$

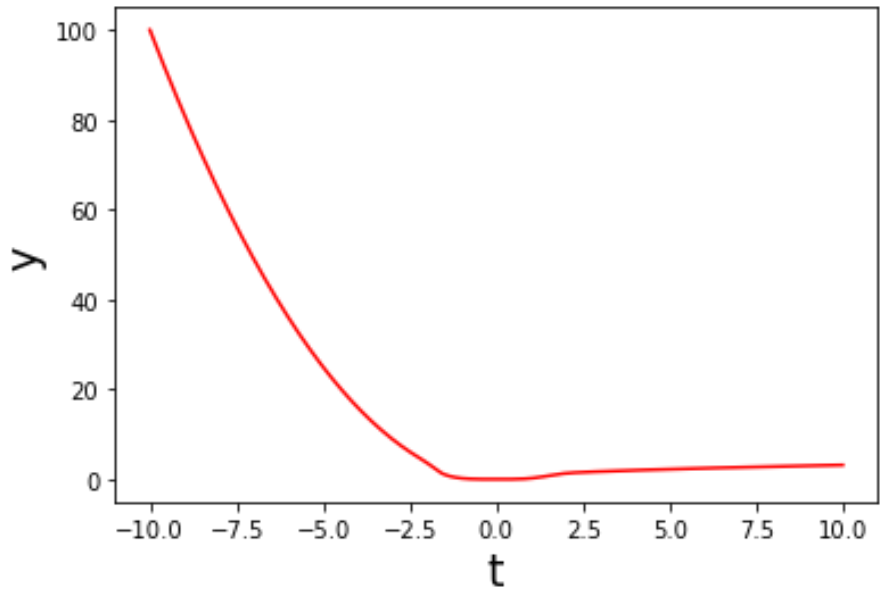
$$y(t) = 1 + t + t^2 + 1.333333t^3 + 2.25t^4 + 3.283333t^5 + 5.238889t^6 \ldots$$



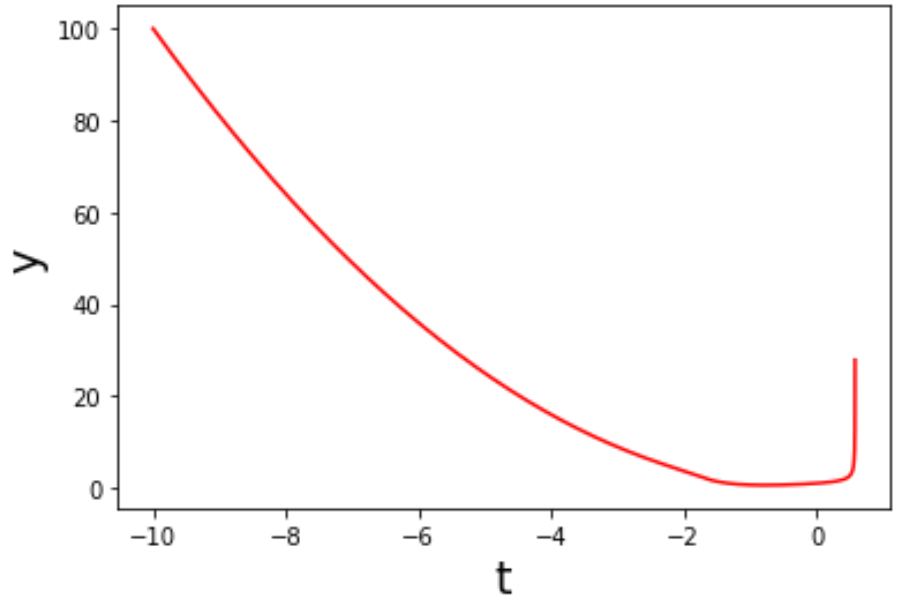
Power Series Method - Error



Qualitative Analysis--Domain And Asymptote



IVP1 with wide range



IVP2 with wide range

We guess that the domain of IVP1 is **R** and there is an asymptote for IVP2

Qualitative Analysis--Domain And Asymptote

For IVP1:

$$\frac{dy}{dt} = (y - t^2)(y^2 - t), \quad y(0) = 0 \tag{1}$$

$$\frac{dy}{dt} = (y - t^2)(y^2 - t), \quad y(0) = 1 \tag{2}$$

From the graph, it is easy to see that $y \approx t^2$ as t keeps decreasing below $t = 0$. So it is reasonable to guess that $y = \alpha t^2$ and $\alpha \to 1$as $t \to -\infty$. By taking $y = \alpha t^2$ into (2) and take the limit, we get

$$\lim_{t \to -\infty} \frac{\alpha - 1}{2\alpha} = \lim_{t \to -\infty} \frac{1}{\alpha^2 t^5 - t^2} = 0$$

, so we get $\alpha = 1$ as $t \to -\infty$, which meets our expectation. Similarly, we can get $y = \sqrt{t}$ as $t \to +\infty$. So we can get the conclusion that for IVP1, there is no asymptote at all, which indicates the domain is $\mathbf{R}$.

Qualitative Analysis--Domain And Asymptote

$$\frac{dy}{dt} = (y - t^2)(y^2 - t), \quad y(0) = 0 \tag{1}$$

$$\frac{dy}{dt} = (y - t^2)(y^2 - t), \quad y(0) = 1 \tag{2}$$

For IVP2:

As we can see from Fig.7(b), the value of y becomes extremely large when $t \to 0.6$, so we guess there is an asymptote around $t = 0.6$, say $t = t_0$To find it, let us look back:

$$Euqation \quad (2) \Rightarrow \frac{y'}{y^3} = 1 - \frac{t^2}{y} - \frac{t}{y^2} + \frac{t^3}{y^3}$$

By taking the limit of both sides and considering that y becomes extremely large, we obtain

$$\lim_{t \to t_0} \frac{y'}{y^3} = \lim_{t \to t_0} (1 - \frac{t^2}{y} - \frac{t}{y^2} + \frac{t^3}{y^3}) = 1$$

so we get $\frac{dy}{dt} \approx y^3 \quad (*)$ as $t \to t_0$. And we can easily solve $(*)$, which is $y = \frac{1}{\sqrt{2C - 2t}}$ for some constant C. By taking some value that has been proved to be as accurate as possible near $t = t_0$, such as $(t, y) = (0.58, 13.87)$ from RK method, we can get $C \approx 0.5826$, which is quite close to the radius of convergence that we get in 4.1.2. So the asymptote can be estimated as

$$t = 0.5826$$

Approximately equal to the radius of convergence from power series method of IVP2

Special Case Approximation

$$y = \begin{cases} t^2, & t \leq -2 \\ \frac{1}{4}t^4, & -2 < t < 1.5 \\ \sqrt{t}, & t \geq 0.5 \end{cases}$$

Table 1: IVP1, differences between accurate value and $y = t^2$ or $y = \sqrt{t}$

t	t^2	$\sqrt{t}$	accurate y	Δ	error%
-3	8.9994	x	8.9259	0.07347	0.823%
-4	15.9992	x	15.9682	0.03097	0.194%
-5	24.9990	x	24.9830	0.01590	0.063%
-6	35.9999	x	35.9907	0.00922	0.0256%
-7	48.9995	x	48.9941	0.00581	0.011%
-8	63.9999	x	63.9961	0.00389	0.006%
-9	80.9999	x	80.9972	0.00274	0.003%
3	x	1.7320	1.7199	0.01206	0.701%
4	x	1.9999	1.9954	0.00453	0.227%
5	x	2.2360	2.2338	0.00221	0.098%
6	x	2.4494	2.4482	0.00124	0.05%
7	x	2.6455	2.6449	0.00077	0.029%
8	x	2.8284	2.8279	0.00051	0.018%
9	x	2.9999	2.9996	0.00035	0.011%

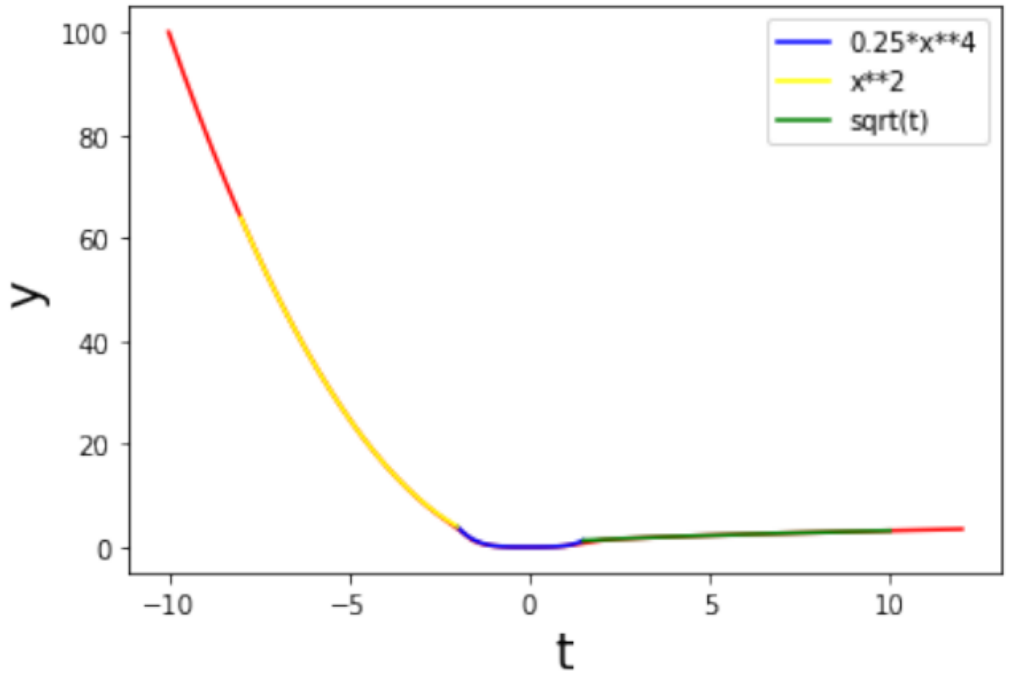
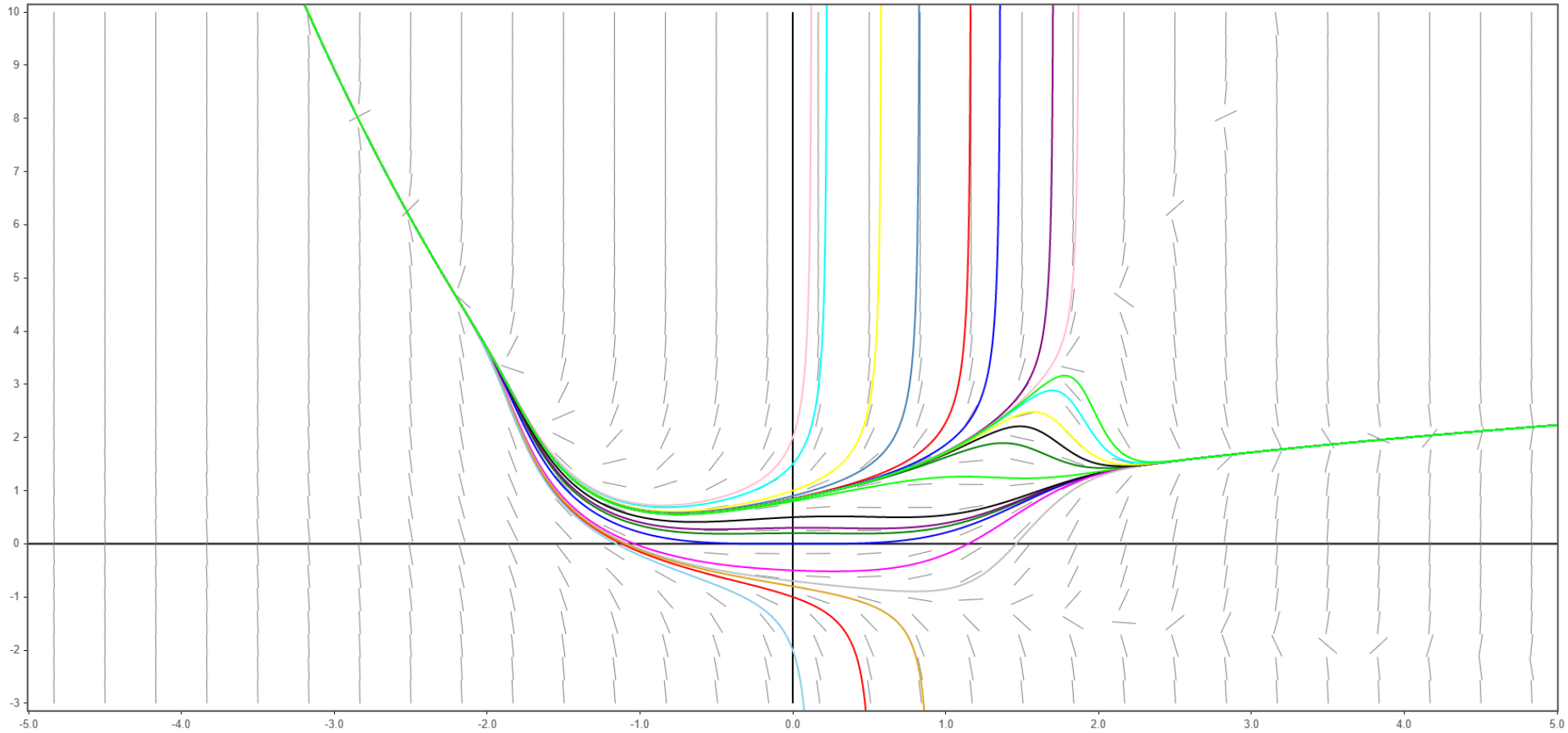


Figure 12: IVP1, $y = \frac{1}{4}t^4$ or $\sqrt{t}$ or t^2 VS $y(t)$

Further observation

More ODEs with different initial value for $y' = (y - t^2)(y^2 - t), \quad y(0) = y_0$



THANKS!