# MATH 285 LAB REPORT

Instructor: Thomas Honold

Group TA: Ouyang Yichen

Group Leader: Yao Xinchen

Group Members: Yan Kaibiao

Huang Haoyu

Li Yuxuan

He Xuhong

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.

Signs:

# 1. Three basic numerical methods.

## 1.1 Introduction

For the equation:

$$y' = f(t, y), y(t_0) = y_0$$

### 1.1.1 Its Euler Method is:

$$y(t + h) = y(t) + f\big(t, y(t)\big) * h$$

This method is the most intuitive method. However, its accuracy is not very high.

### 1.1.2 Its Improved Euler Method is:

$$y_{predicted}(t + h) = y(t) + f\big(t, y(t)\big) * h$$

$$y(t + h) = y(t) + f\big(t, y(t)\big) * \frac{h}{2} + f(t + h, y_{predicted}(t + h))$$

This is based on the equation:

$$y(t_1) = y(t_0) + \int_{t_0}^{t_1} f\big(t, y(t)\big)dt$$

$$\approx y(t_0) + \Big(f\big(t_0, y(t_0)\big) + f\big(t_1, y(t_1)\big)\Big) * \frac{h}{2}$$

### 1.1.3 Its Runge-Kutta Method is:

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

$$k_2 = f\left(t_n + \frac{1}{3}h, y_n + \frac{1}{3}hk_1\right)$$

$$k_3 = f\left(t_n + \frac{2}{3}h, y_n - \frac{1}{3}hk_1 + hk_2\right)$$

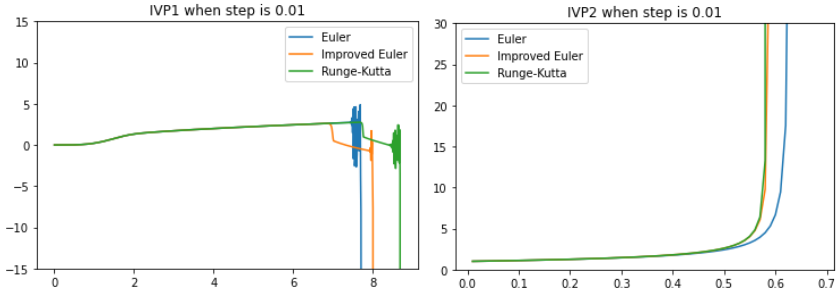
$$k_4 = f(t_n + h, y_n + hk_1 - hk_2 + hk_3)$$

$$y_{n+1} = y_n + \frac{h}{8}(k_1 + 3k_2 + 3k_3 + k_4)$$

## 1.2 Comparison among different methods.

### 1.2.1. Comparison when step size is 0.01.

Using three numerical methods with the same step size to calculate each IVP.

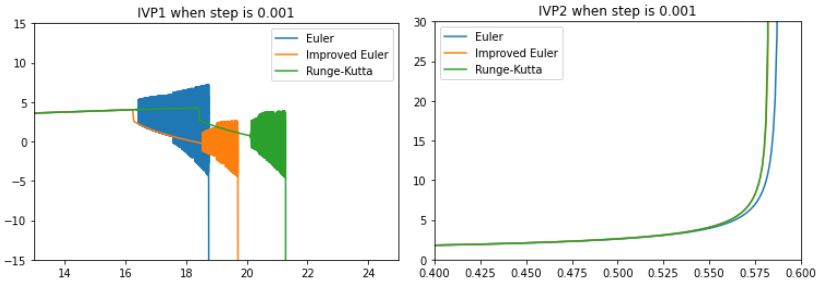


In the IVP1, we can find Euler Method fails to predict the solution when x=7.45, y=2.64. When using Improved Euler Method, the prediction starts to fail when x=6.95, y=2.34, whose range is smaller than Euler Method. However, it is still stable before x=7.93, unlike Euler method start to explode around 7.45. Similarly, Runge-Kutta Method fails when x=7.72, y=2.64, but is stable until x=8.45.

In the IVP2, both Improved Euler Method and Runge-Kutta get a vertical asymptote at around x=0.58. The Euler Method, however, has a vertical asymptote around x=0.62.

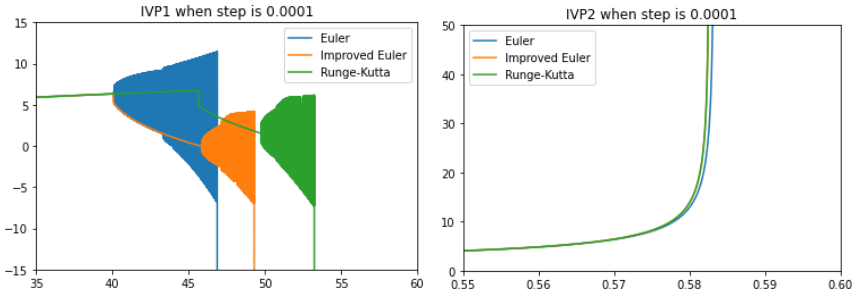
1.2.2. Comparison when step size is 0.001.

When the step size is changed to 0.001, we can get the images below.




Note the unit length of each axis has changed.

1.2.3. Comparison when step size is 0.0001.




Note the unit length of each axis has changed.

## 1.2.4. Conclusion of comparison.

1.2.4.1. IVP1, which has no vertical asymptotes.

We can conclude that the curves first undergo a sudden change, then start to jump up and down, and finally explode. Let's call them shifting point, jumping point, and explosion point. Note that Euler method doesn't have shifting point or have one that coincide with jumping point.

The comparisons of different step sizes show the same result. Euler method Euler Method has the earliest shifting point, followed by Improved Euler Method, and then Runge-Kutta. Similarly, Euler Method has the earliest jumping point, followed by Improved Euler Method, and then Runge-Kutta. It's worth noting that the shift point of Improved Euler is approximately equal to or even earlier than Euler's jumping point.

Runge-Kutta is obviously the best solution. Euler and Improved Euler have their pros and cons. Improved Euler Method fail to predict relatively early. However, it can remain stable for a longer time.

1.2.4.2. IVP2, which has a vertical asymptote.

In this case, Improved Euler Method and Runge-Kutta are very close. They both predict the asymptote well. In contrast, Euler Method has a relatively later asymptote.

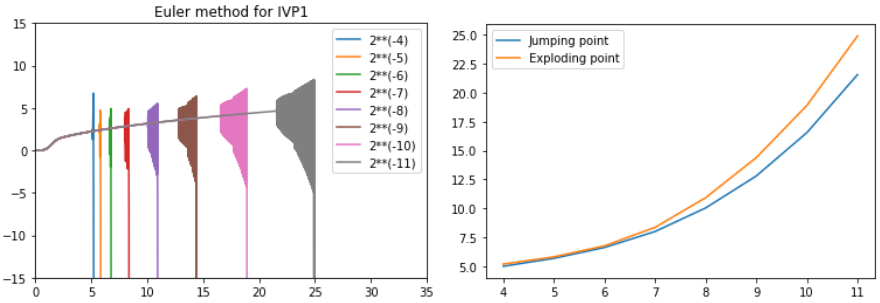
## 1.3. Comparison among different step sizes.

For this part, we use $2^{-n}$ for steps.

1.3.1. IVP1.

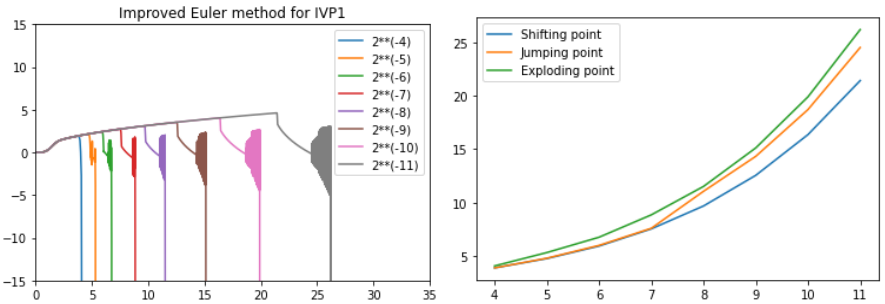
From the analysis in 2.4.1, we can find there are three points that can indicate how well a method can predict the curve of a solution. In later analysis, shifting points are marked as S(-n), jumping points are marked as J(-n), and exploding points are marked as E(-n).

1.3.1.1. Euler method for IVP1.

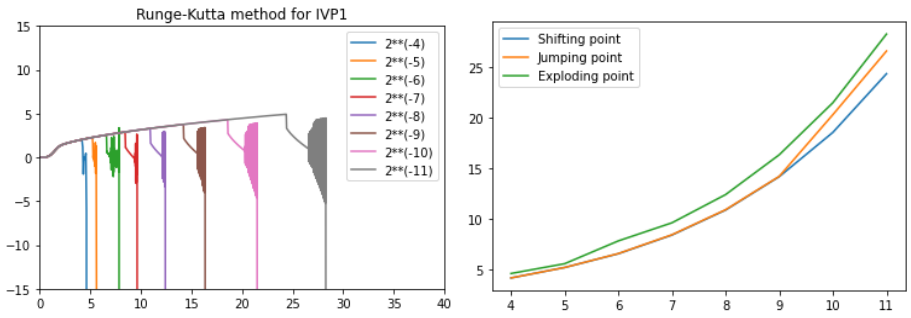



We can see from the graph that as step become smaller exponentially, the two points also increase increasingly faster. The data can be plotted as above.

1.3.1.2. Improved Euler Method for IVP1



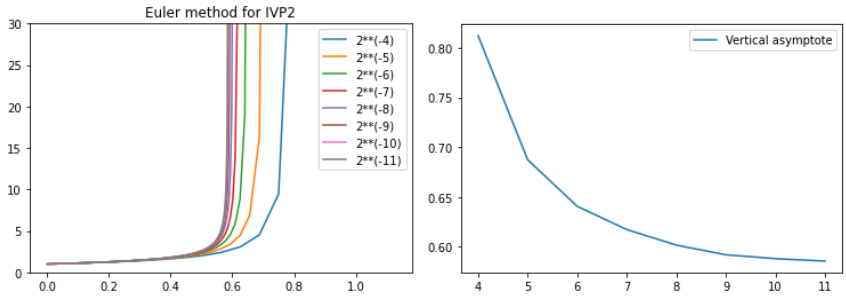

1.3.1.3. Runge-Kutta for IVP1




1.3.2. IVP2.

From the analysis in 2.4.2, there is only one major factor determining how the methods are, the value of vertical asymptotes. We mark it as V(-n).

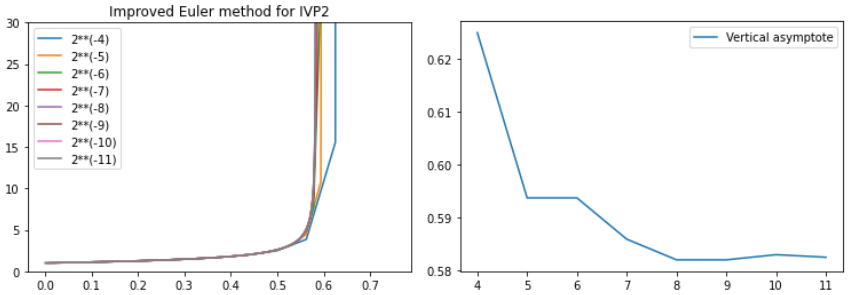
1.3.2.1 Euler Method




From the graph, we can see the curve is increasingly close to a vertical asymptote at around 0.6.

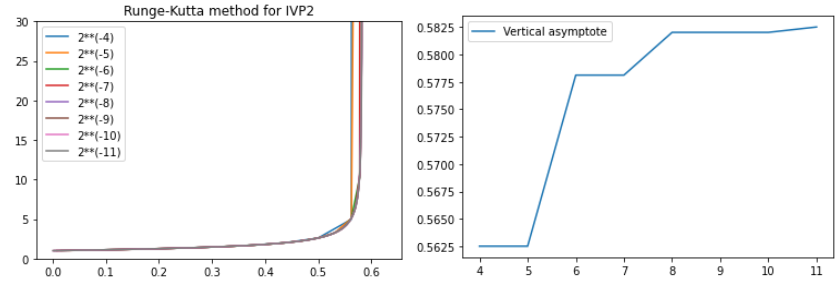
1.3.2.2. Improved Euler Method

Similar to the methods mentioned above, we can obtain its graph.




1.3.2.3. Runge-Kutta

Similar to the methods mentioned above, we can obtain its graph.

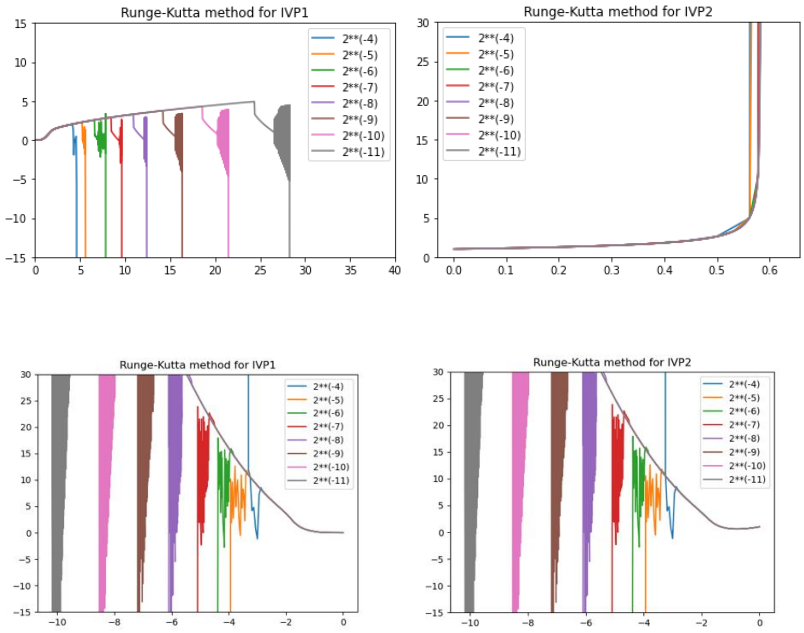


## 2 Asymptote analysis:

$$y' = (y - t^2)(y^2 - t), \quad y(0) = 0\ (IVP1) \text{ or } y(0) = 1\ (IVP2)$$

Two methods of find vertical asymptote:

2.1 According to our plot, we could easily find when step = 2**(-11), y(0.583008) = 3.195e+9， when step = 2**(-12), y(0.583008) = 1.497e+121,when step = 2**(-13), y(0.582764) = 3.858e+21, all the points with y approaches infinite. That we could find the negative and positive vertical asymptote, and the positive vertical asymptote is about t=0.583.




Here IVP1 no vertical asymptote and IVP2 have only positive vertical asymptote. Then we could decide the accurate domain. IVP1's domain is (-∞, ∞), and IVP2's domain is (-∞,0.583).

2.1 For IVP2, we could compute the date.

| h | y(0.5) | y(0.6) |
|---|---|---|
| $2^{-4}$ | 2.606622 | 285.520 |
| $2^{-5}$ | 2.607693 | 7.39e12 |
| $2^{-6}$ | 2.607767 | 8.97e210 |

| | | |
|---|---|---|
| $2^{-7}$ | 2.607772 | $\infty$ |

The values at $t = 0.5$ are reasonable, and we might well believe that the solution has a value of about 2.61 at $t = 0.5$. However, it is not clear what is happening between $t = 0.5$ and $t = 0.6$. Thus, the solution of the problem has a vertical asymptote for some t in $0.5 \leq t \leq 0.6$. According to the precise point (0.583, 3194905643.584661), we find the asymptote is about x = 0.583.

# 3 Power Series

## 3.1 Introduction

An analytic solution (or power series solution) of a scalar ODE is a "power series function". We can use the following "Power Series" to approach the solution of an ODE in a certain interval *I*.

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

## 3.2 Process & Method

Since $f(t, y) = (y - t^2)(y^2 - t)$ is analytic, its solution must be analytic. Therefore, choosing 0 as $t_0$, we can assume that $y(t) = \sum_{n=0}^{\infty} a_n t^n$. For (IVP1), $y(0) = a_0 = 0$. For (IVP2), $y(0) = a_0 = 1$. We can write the ODE into $f(t, y) = y^3 - ty - t^2 y^2 + t^3$. Since $y(t) = \sum_{n=0}^{\infty} a_n t^n$, we obtain $y'(t) = \sum_{n=1}^{\infty} n a_n t^{n-1}$, $y^2(t) = \sum_{n=0}^{\infty} b_n t^n$, $y^3(t) = \sum_{n=0}^{\infty} \sum_{k=0}^{n} (b_k a_{n-k}) t^n$, where $b_n = \sum_{k=0}^{n} a_k a_{n-k}$. Therefore, the (IVP) can be written as:

$$\sum_{n=1}^{\infty} n a_n t^{n-1} = \sum_{n=0}^{\infty} (\sum_{k=0}^{n} b_k a_{n-k}) t^n - t \sum_{n=0}^{\infty} a_n t^n - t^2 \sum_{n=0}^{\infty} b_n t^n + t^3$$

$$\Rightarrow \sum_{n=0}^{\infty} (n+1) a_{n+1} t^n = \sum_{n=0}^{\infty} (\sum_{k=0}^{n} b_k a_{n-k}) t^n - \sum_{n=1}^{\infty} a_{n-1} t^n - \sum_{n=2}^{\infty} b_{n-2} t^n + t^3$$

$$\Rightarrow a_1 + 2a_2 t + \sum_{n=2}^{\infty} (n+1) a_{n+1} t^n =$$

$$a_0 b_0 + (a_1 b_0 + a_0 b_1) t - a_0 t + \sum_{n=2}^{\infty} (\sum_{k=0}^{n} b_k a_{n-k}) t^n - \sum_{n=2}^{\infty} a_{n-1} t^n - \sum_{n=2}^{\infty} b_{n-2} t^n + t^3$$

Finally, we can obtain the following recursion relation:

$$(n+1)a_{n+1} = \begin{cases} \left(\sum_{k=0}^{n} b_k a_{n-k}\right) - a_{n-1} - b_{n-2}, & n \neq 3 \\ \left(\sum_{k=0}^{n} b_k a_{n-k}\right) - a_{n-1} - b_{n-2} + 1, & n = 3 \end{cases}$$

Based on the above relation, for (IVP1), we can obtain the solution,

$$y(t) = 0.25t^4 - 0.041667t^6 + 0.005208t^8 - 0.000521t^{10} - 0.005682t^{11} + \cdots$$

and its radius of convergence,

$$\rho = \frac{1}{L} = \frac{1}{\lim_{n\to\infty} sup \sqrt[n]{a_n}} = 1.4196$$

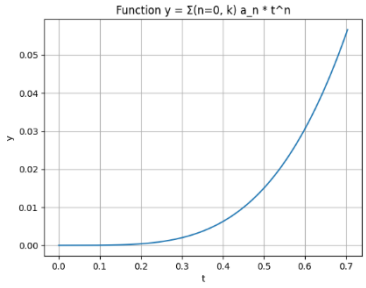
for (IVP1), we can obtain

$$y(t) = 1 + t + t^2 + 1.333333t^3 + 2.25t^4 + 3.283333t^5 + 5.238889t^6 + 8.364286t^7 + \cdots$$

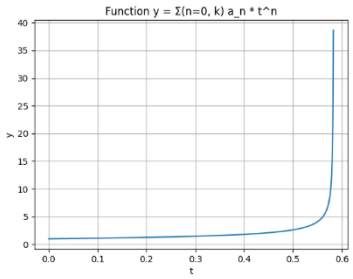
and its radius of convergence,

$$\rho = \frac{1}{L} = \frac{1}{\lim_{n\to\infty} \frac{|a_{n+1}|}{|a_n|}} = 0.5829$$

Hence, we can draw the graph of (IVP1) and (IVP2):




IVP1




IVP2

# 4 Linear Multi-step

We already introduce some numerical methods including Euler, improved Euler and Runge-Kutta. We introduce linear multistep method, including Milne-Simpson method, Adams-Bashforth-Moulton method to solve these problems. Each method has to solve 4 history point and uses 4 previous points to get next one ($4^{th}$order for example). The first formular is to solve the derivative of $y_{k+1}$ value, and the next step we will use this value to get a new $y_{k+1}$ value, which is more precise. However, in these two problems, we only have one initial point, which means we need at least three more points to use the method. So, we use improved Euler or Runge-Kutta method to get three more points. (The main code is attached in the appendix)

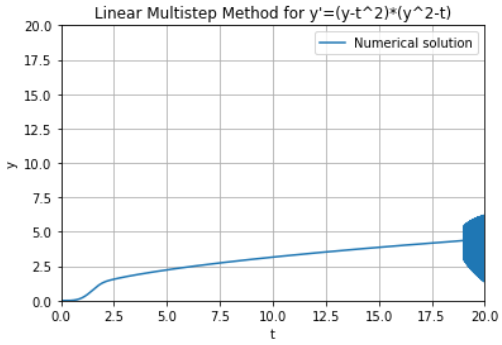
## 4.1 Adams-Bashforth-Moulton method

The estimation is based on the calculation on $[t_k,\ t_{k+1}]$. By using improve Euler function we can get the history solution. Eg. I use improved Euler function to get y1, y2, y3.

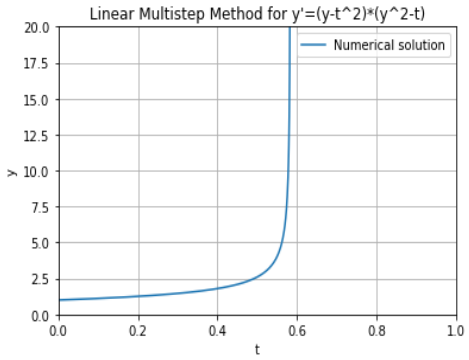
$$y'_k = y_k + \int_{t_k}^{t_{k+1}} f(t,y) = y_k + \frac{h}{24}(-9f_{k-3}+37f_{k-2} - 59f_{k-1} + 55f_k)$$

This means we know a new point $(t_{k+1}$,f $(t_{k+1}, y'_{k-1})$ ), which can be used to find $y_{k+1}$again, by calculate the integration on $[t_k,\ t_{k+1}]$, $y_{k+1} = y_k + \int_{t_k}^{t_{k+1}} f(t,y) = y_k + \frac{h}{24}(f_{k-2}-5f_{k-1} + 19f_k + 9f_{k+1})$. (The main code is attached in the appendix)

IVP1: Step:0.0001 times: 1000000




IVP2: Step: 0.001 times: 1000




## 4.2 Milne-Simpson Method

The Milne-Simpson method is one of the prediction-correction methods. Similar to Adams-Bash forth-Moulton method, it uses two formulas: a prediction formula to estimate the next point based on existing data, and a correction formula to improve this estimate.

For Milne-Simpson method, the Milne formula is used as prediction formula and Simpson formula is used as correction formula. A fourth-order method as it is, it requires four initial values of y to start, while the main error term of both the predictor and the corrector contain $h^5$.

Integrating $\frac{dy}{dt} = f(t,y)$ on $(t_{k-3}, t_{k+1})$, which is $y(t_{k+1}) - y(t_{k-3}) = \int_{t_{k-3}}^{t_{k+1}} f(t,y(t))dx$

obtain the Milne formula:

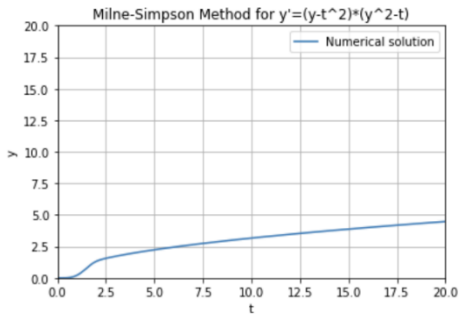
$$y_{k+1} = y_{k-3} + \frac{4}{3}h(2f_k - f_{k-1} + 2f_{k-2})$$

Integrating $\frac{dy}{dt} = f(t,y)$ on $(t_{k-1}, t_{k+1})$, which is $y(t_{k+1}) - y(t_{k-1}) = \int_{t_{k-1}}^{t_{k+1}} f(t,y(t))dx$ and obtain Simpson formula:

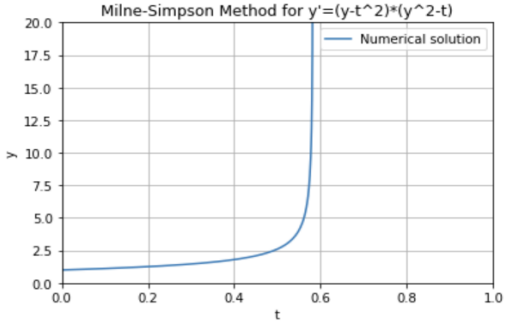
$$y_{k+1} = y_{k-1} + \frac{1}{3}h(f_{k+1} + 4f_k + f_{k-1})$$

*The plotted results are shown below, while the code attached in the appendix.*

*(Graph 1, the graph for IVP y(0)=0)*




*(Graph 2, the graph for IVP y(0)=1)*




# 5 Error Analysis

(1) Euler Method

As it is mentioned above, the Euler method comes from the Taylor approximation:

$$y(t+h) = y(t) + y'(t) * h + \frac{1}{2} * y''(c) * h^2$$

where $c \in [t, t+h]$. For a small h, we can estimate $h^2 \approx h$, then the global error can be estimate as $\frac{1}{2}\sum_{k=1}^{M} y^{(2)}(c_k)h^2 \approx \frac{1}{2}(b-a)y^{(2)}(c)h$, Thus the global error will be within **O(h).**

(2) Improved Euler Method

Since the Improved Euler Method is based on Euler method and estimated using the area of trapezoid, we also estimate the error by using the approximation of integral by area of trapezoid,

which is $-\sum_{k=1}^{M} y^{(2)} c_k \frac{h^3}{12} \approx \frac{b-a}{12} y^{(2)} c h^2$, Thus the global error will be within **O($h^2$).**

(3) Runge-Kutta method

Due to the relation of order and highest accuracy by Butcher, it can be concluded that:

For order 2 formula, highest accuracy is $O(h^2)$;

For order 3 formula, highest accuracy is $O(h^3)$;

For order 4 formula, highest accuracy is $O(h^4)$;

For order 5 formula, highest accuracy is $O(h^4)$;

For order 6 formula, highest accuracy is $O(h^5)$;

For order 7 formula, highest accuracy is $O(h^6)$;

For formula that has an order $n \geq 8$, highest accuracy is $O(h^{n-2})$.

APPENDIX:

Code for 3 basic methods:

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

def dy(t, y):
    return (y - t**2) * (y**2 - t)

# euler
def EM(y0, step, x_range):
    y = y0
    t = 0
    ys = []
    ts = []
    while t < x_range:
        try:
            y += step * dy(t, y)
        except OverflowError:
            y = np.nan
        t += step
        ys.append(y)
        ts.append(t)
    return np.array(ys), np.array(ts)

# improved euler
def IEM(y0, step, x_range):
    y = y0
    t = 0
    ys = []
    ts = []
    while t < x_range:
        try:
            y += step * (dy(t, y) + dy(t + step, y + step * dy(t, y))) /
2
        except OverflowError:
            y = np.nan
        t += step
        ys.append(y)
        ts.append(t)
    return np.array(ys), np.array(ts)

# Runge-Kutta
def RK(y0, step, x_range):
    y = y0
```

```
    t = 0
    ys = []
    ts = []
    while t < x_range:
        try:
            k1 = dy(t, y)
            k2 = dy(t + step / 2, y + step * k1 / 2)
            k3 = dy(t + step / 2, y + step * k2 / 2)
            k4 = dy(t + step, y + step * k3)
            y += step * (k1 + 2 * k2 + 2 * k3 + k4) / 6
        except OverflowError:
            y = np.nan
        t += step
        ys.append(y)
        ts.append(t)
    return np.array(ys), np.array(ts)

x_range = 10
plt.ylim(top=15)
plt.ylim(bottom=-15)

# euler method for IVP1
Y_E, T = EM(0, 0.01, x_range)
plt.plot(T, Y_E, label='Euler')

# improved euler method for IVP1
Y_I, T = IEM(0, 0.01, x_range)
plt.plot(T, Y_I, label='Improved Euler')

# Runge-Kutta for IVP1
Y_R, T = RK(0, 0.01, x_range)
plt.plot(T, Y_R, label='Runge-Kutta')

# plot
plt.title('IVP1 when step is 0.01')
plt.legend()
plt.show()

x_range = 10
plt.ylim(top=30)
plt.ylim(bottom=0)

# euler method for IVP2
Y_E, T = EM(1, 0.01, x_range)
```

```
plt.plot(T, Y_E, label='Euler')

# improved euler method for IVP2
Y_I, T = IEM(1, 0.01, x_range)
plt.plot(T, Y_I, label='Improved Euler')

# Runge-Kutta for IVP2
Y_R, T = RK(1, 0.01, x_range)
plt.plot(T, Y_R, label='Runge-Kutta')

# plot
plt.title('IVP2 when step is 0.01')
plt.legend()
plt.show()
```

Code for Power Series:

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

def compute_an1(n):
    a = [0] * (n + 1)
    b = [0] * (n + 1)

    a[0] = 0
    a[1] = 0
    a[2] = 0
    b[0] = 0
    b[1] = 0
    b[2] = 0

    for i in range(2, n - 1):
        if i != 3:
            a[i + 1] = ( sum(b[k] * a[i - k] for k in range(i + 1)) -
a[i - 1] - b[i - 2] ) / (i + 1)
        else:
            a[i + 1] = ( sum(b[k] * a[i - k] for k in range(i + 1)) -
a[i - 1] - b[i - 2] + 1 ) / (i + 1)
        b[i + 1] = sum(a[k] * a[i + 1 - k] for k in range(i + 2))

    return a
```

```
def compute_an2(n):
    a = [0] * (n + 1)
    b = [0] * (n + 1)

    a[0] = 1
    a[1] = 1
    a[2] = 1
    b[0] = 1
    b[1] = 2
    b[2] = 3

    for i in range(2, n - 1):
        if i != 3:
            a[i + 1] = ( sum(b[k] * a[i - k] for k in range(i + 1)) -
a[i - 1] - b[i - 2] ) / (i + 1)
        else:
            a[i + 1] = ( sum(b[k] * a[i - k] for k in range(i + 1)) -
a[i - 1] - b[i - 2] + 1 ) / (i + 1)
        b[i + 1] = sum(a[k] * a[i + 1 - k] for k in range(i + 2))

    return a

n = 1000        #IVP1:2000,  IVP2:1000
result = compute_an2(n)

print(result)

# print("IVP1: radius of convergence:", result[1999] ** (1/2000) )
print("IVP2: radius of convergence:", result[998] / result[999] )


def compute_function(t, coefficients):
    result = 0
    for n, a in enumerate(coefficients):
        result += a * t**n
    return result

coefficients = result

t_min = 0
t_max = 0.5829             #IVP1:0.7044, IVP2:0.5829
num_points = 1000
t_values = np.linspace(t_min, t_max, num_points)
```

```
y_values = [compute_function(t, coefficients) for t in t_values]

plt.plot(t_values, y_values)
plt.xlabel('t')
plt.ylabel('y')
plt.title('Function y = Σ(n=0, k) a_n * t^n')
plt.grid(True)
plt.show()
```

Code for Multi-step methods:

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

#  dy/dt = (y - t^2)(y^2 - t)
def f(t, y):
    return (y - t**2) * (y**2 - t)

# def improve Eurler
def improved_euler_step(t, y, h):
    k1 = h * f(t, y)
    k2 = h * f(t + h, y + k1)
    return y + 0.5 * (k1 + k2)

# Adams-Bashforth
def adams_bashforth_step(t, y, h, history):
    f_values = [f(t_i, y_i) for t_i, y_i in history]
    return y + h * (55 * f_values[-1] - 59 * f_values[-2] + 37 *
f_values[-3] - 9 * f_values[-4]) / 24

# initial condition
t0 = 0  # time
y0 = 1  #  y(t0) -------------------IVP2
h = 0.001  # step
num_steps = 1000  # times

# initial time
t_values = np.zeros(num_steps + 1)
y_values = np.zeros(num_steps + 1)

# condition
t_values[0] = t0
y_values[0] = y0
```

```
# euler condition
for i in range(3):
    t = t_values[i]
    y = y_values[i]
    y_next = improved_euler_step(t, y, h)
    t_values[i+1] = t + h
    y_values[i+1] = y_next

# Adams-Bashforth
for i in range(3, num_steps):
    t = t_values[i]
    y = y_values[i]
    history = [(t_values[j], y_values[j]) for j in range(i-3, i+1)]
    y_next = adams_bashforth_step(t, y, h, history)
    t_values[i+1] = t + h
    y_values[i+1] = y_next

# plot
plt.plot(t_values, y_values, label='Numerical solution')
plt.xlabel('t')
plt.ylabel('y')
plt.title("Linear Multistep Method for y'=(y-t^2)*(y^2-t)")
plt.legend()
plt.grid(True)
plt.xlim(0, 1)  # x
plt.ylim(0, 20)  # y
plt.show()


import numpy as np
import matplotlib.pyplot as plt

# dy/dt = (y - t^2)(y^2 - t)
def f(t, y):
    return (y - t**2) * (y**2 - t)

# Milne-Simpson method
def milne_simpson_step(t, y, h):
    f1 = f(t, y)
    f2 = f(t + h/2, y + (h/2) * f1)
    f3 = f(t + h/2, y + (h/2) * f2)
    f4 = f(t + h, y + h * f3)
    return y + (h / 6) * (f1 + 4*f3 + f4)
```

```
# Exact solution
def exact_solution(t):
    return (1 + t + t**3) / (1 - t + t**2)

# initial condition
t0 = 0  # time
y0 = 1  # y(t0)
h = 0.001  # step
num_steps = 1000  # times

# initial time
t_values = np.zeros(num_steps + 1)
y_values = np.zeros(num_steps + 1)
# = np.zeros(num_steps + 1)

# condition
t_values[0] = t0
y_values[0] = y0
#exact_values[0] = y0

# Milne-Simpson method
for i in range(num_steps):
    t = t_values[i]
    y = y_values[i]
    y_next = milne_simpson_step(t, y, h)
    t_values[i+1] = t + h
    y_values[i+1] = y_next
    #exact_values[i+1] = exact_solution(t + h)

# plot
plt.plot(t_values, y_values, label='Numerical solution')
#plt.plot(t_values, exact_values, label='Exact solution')
plt.xlabel('t')
plt.ylabel('y')
plt.title("Milne-Simpson Method for y'=(y-t^2)*(y^2-t)")
plt.legend()
plt.grid(True)
plt.xlim(0, 1)
plt.ylim(0, 20)
plt.show()
```

Reference:

[1] TecrayC. “常(偏)微分方程的数值求解”. https://zhuanlan.zhihu.com/p/435769998
[2] 深渊潜航. “预测-修正法(Milne-Simpson 和 Adams-Bashforth-Moulton)解常微分方程”. https://blog.csdn.net/seventonight/article/details/117194468