MATH286 LAB

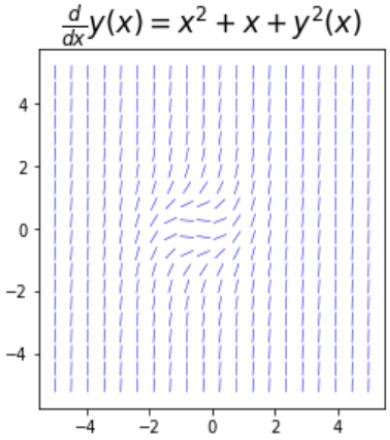
Zheng Fang 3190110722, Bo Pang 3190110669, Yuxin Qu 3190110414, Yuan Xu 3190110839, Jiayu Zhang 319

May 2021

We declare that this report is our own original work, and every work group member has a fair share in this work. For the preparation of the report we have not used any other resources than the Math 286 lecture material and the references cited in the report.

1 Problem1 $y' = y^2 + t^2 + t$

1.1 Direction Field



1.2 Vertical Asymptotes

First, we consider the first differential equation. Since this differential equation is nonlinear, the exist and uniqueness theorem guarantees only that there is a solution in some interval about $t = 0$. So, we use Runge-Kutta method and Euler method to compute approximate solution on the interval between $0 \leq t \leq 1$,and $-1 \leq t \leq 0$.

We found that when t approaches 1 and -3 the value becomes infinite. And when using different method the values about $t = 1$ and $t = -3$ are quite different see figure below.

Table 1: Euler Method

h	t=0.9	t=1	t=-2.9	t=-2.8
0.1	7.21903901	12.63049143	-21.87736782	-10.43579211
0.05	10.53944376	29.38549348	-5.52770048e+01	-1.42395650e+01
0.01	26.98469231	9.56223083e+16	-2.48632937e+28	-3.26613218e+01
0.001	72.54585927	INF	-INF	-80.06050893

Table 2: improved Euler Method

h	t=0.9	t=1	t=-2.9	t=-2.8
0.1	33.33297537	1135.16390732	-2176743.36597152	-251.80450454
0.05	52.93023524	80506569.97752264	-2.39575111e+18	-279.8641361
0.01	100.81838691	INF	-INF	-220.74014195
0.001	102.65706881	INF	-INF	-117.97003763

Table 3: Runge-Kutta Method

h	t=0.9	t=1	t=-2.9	t=-2.8
0.1	341.59592594	2.20671354e+21	-9.48313569e+120	-5.92879509e+08
0.05	322.98613266	3.03931386e+242	-INF	-2.31050560e+07
0.01	143.78553654	INF	-INF	-9.14494577e+02
0.001	103.16361989	INF	-INF	-118.720058

The values at $t = 0.9$ are reasonable, and when we use smaller h we can find that all values of Euler Method and Runge-Kutta Method are approaching 100. So, we might well believe that the solution has a value of about $99.8704(h = 0.000001)$. It is same for $t = -2.8$ has value about -106.576$(h = 0.000001)$.

But it is not clear what happen between $t = 0.9$ and $t = 1$. To help clarify this, we turn to some analytical approximations to the solution of the IVP. Note that $0 \leq t \leq 1$.

$$y^2 \leq y^2 + t^2 + t \leq y^2 + 2 \tag{1}$$

This suggests that the solution $y = \phi_1(t)$ of

$$y' = y^2 + 2, y(0) = 1 \tag{2}$$

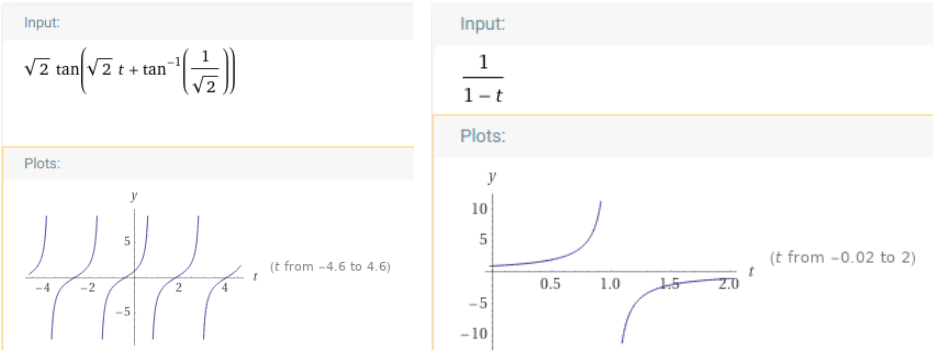
and the solution $y = \phi_2(t)$ of

$$y' = y^2, y(0) = 1 \tag{3}$$

are upper and lower bounds, respectively for the solution of the original problem, since all these solutions pass through the same initial point.$\phi_1(t) \leq \phi(t) \leq \phi_2(t)$ Indeed, it can be shown that as long as these functions exist. Luckily, we can solve (2) and (3).

$$\phi_1(t) = \sqrt{2}\tan\left(\sqrt{2}t + \tan^{-1}\left(\frac{1}{\sqrt{2}}\right)\right), \phi_2(t) = \frac{1}{1-t} \tag{4}$$

ϕ_2(t) $\to \infty$, as t $\to 1$, and $\phi_1(t) \to \infty, t \to \frac{\left(\frac{\Pi}{2} - \tan^{-1}\left(\frac{1}{\sqrt{2}}\right)\right)}{\sqrt{2}} \cong 0.6755$



These calculations show that the solution of the original IVP exist at least for$0 \leq t < 0.675$
The solution has vertical asymptote for some $0.6755 \leq t <$ 1about $t = 0.955$.
The same calculation can also be applied $-2.9 < t < 0$.

$$y^2 - \frac{1}{4} \leq y^2 + t^2 + t \leq y^2, y(0) = 1$$

$$\phi_3(t) \le \phi(t) \le \phi_1(t)$$

$$\phi_3(t) = \frac{e^t + 3}{6 - 2e^t}$$

The vertical asymptote of the third function is $t = ln(3)$. However, it does not inside the interval $-2.9 < t < 0$. So we only know that vertical asymptote appear about $-2.9 < t < -2.8$.

From above, we can conclude that the first IVP has vertical asymptote for some $0.6755 \le t1$ about $t = 0.955$ and for some $-2.9t - 2.8$.

1.3 Power series analysis

for $t^0 : a_1 = a_0^2 = 1$

for $t^1 : a_2 = \frac{1}{2}(2a_0a_1 + 1) = 1.5$

for $t^2 : a_3 = \frac{1}{3}(a_0a_2 + a_1^2 + a_2a_0 + 1) = \frac{4}{3}$

for $t^n \ n \ge 3 : a_n = \frac{1}{n}\sum_{i=0}^{n-1} a_i a_{n-1-i} = \frac{17}{12}$

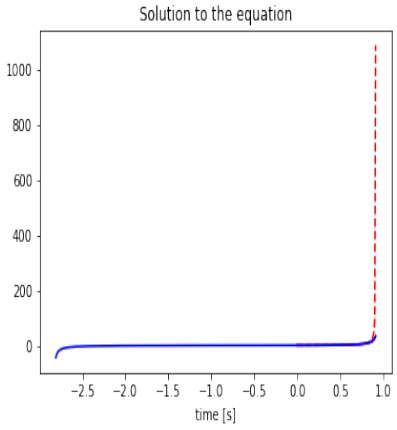
The recursion of this value of $a_n(n \ge 3)$ can be computed iteratively by code. By using the python code, we caculate a_0 to a_1000

```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
an = np.zeros(1000)
an[0] = 1
an[1] = 1
an[2] = 3/2
an[3] = 5/3
def multi(i,n):
    return an[i]*an[n-i]

def sum(n):
    num = 0
    for i in range(n):
        num = num+  multi(i,n-1)
    return num/n
def multi(i,n):

    return an[i]*an[n-i]
for k in range(996):
    an[k+4] = sum(k+4)
```

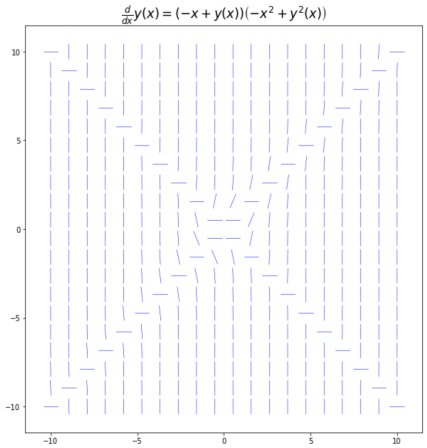
```
[1.00000000e+00 1.00000000e+00 1.50000000e+00 1.66666667e+00
 1.58333333e+00 1.75000000e+00 1.94444444e+00 2.13095238e+00
 2.33482143e+00 2.56723986e+00 2.82201279e+00 3.10055014e+00
 3.40692958e+00 3.74394329e+00 4.11418646e+00 4.52098319e+00
 4.96803448e+00 5.45930345e+00 5.99914401e+00 6.59236491e+00
 7.24424803e+00 7.96059250e+00 8.74777194e+00 9.61279127e+00
 1.05633477e+01 1.16078995e+01 1.27557413e+01 1.40170869e+01
 1.54031601e+01 1.69262945e+01 1.86000433e+01 2.04393000e+01
 2.24604307e+01 2.46814201e+01 2.71220309e+01 2.98039805e+01
 3.27511333e+01 3.59897141e+01 3.95485403e+01 4.34592795e+01
 4.77567303e+01 5.24791326e+01 5.76685075e+01 6.33710312e+01
 6.96374464e+01 7.65235132e+01 8.40905056e+01 9.24057565e+01
 1.01543257e+02 1.11584315e+02 1.22618279e+02 1.34743331e+02
 1.48067363e+02 1.62708936e+02 1.78798334e+02 1.96478725e+02
 2.15907433e+02 2.37257340e+02 2.60718423e+02 2.86499444e+02
 3.14829809e+02 3.45961610e+02 3.80171864e+02 4.17764984e+02
 4.59075482e+02 5.04470949e+02 5.54355326e+02 6.09172497e+02
 6.69410239e+02 7.35604562e+02 8.08344480e+02 8.88277251e+02
 9.76114138e+02 1.07263673e+03 1.17870392e+03 1.29525952e+03
```



Here is the power series solution plot with the Euler solution.

2 Problem2 $y' = (y - t)(y^2 - t^2)$

2.1 Direction Field



2.2 Vertical Asymptotes

For the second IVP, we also apply the same method.

$$(y-1)\left(y^2-1\right) \leq (y-t)\left(y^2-t^2\right) \leq y^3$$

$$\phi_1(t) \leq \phi(t) \leq \phi_2(t)$$

$$\phi_1(t) = \frac{0.5(t-1.41421\sqrt{2-t}-2)}{t-2}$$

$$\phi_2(t) = \frac{1}{\sqrt{1-2t}}$$

The upper bound function has vertical asymptote $t = 0.5$. The lower bound function has vertical asymptote $t = 2$. So vertical asymptote is between $(0.5, 2)$. From the figure that it has vertical asymptote about $t = 0.62(h = 0.0001)$.

Table 4: Euler Method

h	t=0.5	t=0.6
0.1	1.62705	1.16401285
0.05	1.82368	2.02655013
0.01	2.15686	2.30981477
0.001	2.29765	5.52039219

Table 5: improved Euler Method

h	t=0.5	t=0.6
0.1	1.78099364	2.22789579
0.05	2.00829847	2.97029575
0.01	2.24918991	4.83400909
0.001	2.30979475	5.99059255

Table 6: Runge-Kutta Method

h	t=0.5	t=0.6
0.1	1.81007256	2.33609969
0.05	2.02655013	3.12733931
0.01	2.25084321	4.92924419
0.001	2.30981477	5.99379866

2.3 Left Asymptote(A Failed Attempt)

From the numerical analysis, we can assume the left asymptote is$y = -t$. So we need find two function to squeeze the second ODE to prove its left Asymptotic line is $y = -t$.Obviously, one of the bounded function is $y = -t$.But it's difficult to get the second function. We assume the ODE solution is $y = -t + f(t)$, substitute in to the ODE:

$$f(t)' = f(t)(f(t)-2t)^2 + 1 \leq a(t)$$

$$\lim_{t \to -\infty} a(t) = 0$$

It can prove the Left Asymptote is $y = -t$.But we can't find the appropriate $a(t)$.

2.4 Power series analysis

for $t^0 : a_1 = a_0^3 = 1$

for $t^1 : a_2 = \frac{1}{2}(3a_0^2 a_1 - a_0^2) = 1$

for $t^2 : a_3 = \frac{1}{3}(3a_0^2 a_2 + 3a_1^2 a_0 - 2a_0 a_1 - a_0) = 1$

for $t^3 : a_4 = \frac{1}{4}\sum_{i=0}^{3}\sum_{j=0}^{3-i} a_i a_j a_{3-i-j})$ - $\sum_{k=0}^{2} a_k a_{2-k} - a_{n-3} + 1 = \frac{7}{4}$

for $t^n\ n \geq 4 : a_n = \frac{1}{n}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1-i} a_i a_j a_{n-1-i-j})$ - $\sum_{k=0}^{n-2} a_k a_{n-2-k} - a_{n-3}$

The recursion of this value of $a_n (n \geq 4)$ can be computed iteratively by code. By using the python code, we caculate a_0 to $a_4 00$

Here is the power series solution plot with the Euler solution.

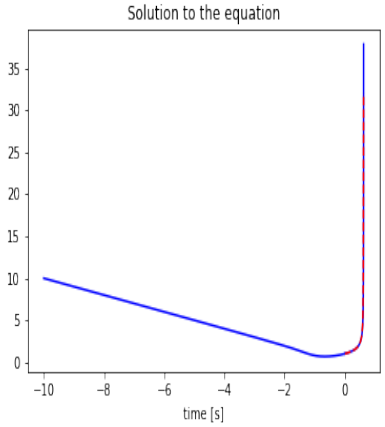
```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
an = np.zeros(400)
an[0] = 1
an[1] = 1
an[2] = 1
an[3] = 1
an[4] = 7/4
def sum(n):
    num = 0
    for i in range(n):
        for j in range(n-i):
            num = num+  multi(i,j,n-1)
    for i in range(n-1):
        num=num - multi2(i,n-1)
    num = num-an[n-3]
    return num/n

def multi(i,j,n):
    return an[i]*an[j]*an[n-i-j]

def multi2(i,n):
    return an[i]*an[n-1-i]

sum(5)
for k in range(395):
    an[k+5] = sum(k+5)
```

```
[1.00000000e+00 1.00000000e+00 1.00000000e+00 1.00000000e+00
 1.75000000e+00 2.45000000e+00 3.72500000e+00 5.63928571e+00
 8.62723214e+00 1.33029266e+01 2.06003423e+01 3.20592257e+01
 5.00672023e+01 7.84435274e+01 1.23236088e+02 1.94064598e+02
 3.06236539e+02 4.84131554e+02 7.66617743e+02 1.21570712e+03
 1.93041535e+03 3.06895191e+03 4.88427472e+03 7.78108198e+03
 1.24072087e+04 1.98002611e+04 3.16229687e+04 5.05410763e+04
 8.08303276e+04 1.29351765e+05 2.07119353e+05 3.31820468e+05
 5.31869639e+05 8.52930152e+05 1.36840875e+06 2.19634843e+06
 3.52662664e+06 5.66475637e+06 9.10244133e+06 1.46312521e+07
 2.35258165e+07 3.78390920e+07 6.08783922e+07 9.79729586e+07
 1.57711866e+08 2.53940806e+08 4.08983526e+08 6.58839836e+08
 1.06157432e+09 1.71085629e+09 2.75781752e+09 4.44634242e+09
 7.17005324e+09 1.15643400e+10 1.86550058e+10 3.00983873e+10
 4.85692889e+10 7.83877973e+10 1.26532189e+11 2.04275929e+11
 3.29833570e+11 5.32637806e+11 8.60253947e+11 1.38955929e+12
 2.24481973e+12 3.62692168e+12 5.86064772e+12 9.47114035e+12
 1.53075856e+13 2.47432931e+13 3.99993893e+13 6.46685328e+13
 1.04562323e+14 1.69082569e+14 2.73440435e+14 4.42247963e+14
 7.15331121e+14 1.15713913e+15 1.87197596e+15 3.02865821e+15
```



3 Error Analysis

There are three fundamental sources of error in numerical approximations. First, the algorithm used in calculations is an approximate one. Second, except for the first step, the input data in calculation in each step is not the exact value. Third, the computer used for the calculations has finite precision. It will create round-off error.

Smaller step size h can obtain more accurate value in algorithm. However, if h is very small, a great many steps will be required to cover a fixed interval, and the global round-off error will increase. Therefore, there exist a suitable h for numerical methods. Since different computers have different representation bits and we don't know the exact solution for these two IVPs. We decide to analysis the error created by algorithm, which can compare the accuracy of different algorithms.

3.1 Euler Method

Algorithm:

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1})$$

Calculation for local truncation error:

Assume that $y_k = y(x_k)$,then $y_{k+1} = y(x_k) + hf(x_k, y(x_k)) = y(x_k) + hy'(x_k)$

The second order Taylor expansion is performed:

$y(x_{k+1}) = y(x_k) + hy'(x_k) + \frac{h^2}{2}y''(\xi) \quad \xi \in (x_k, x_{k+1})$

then,$T_{k+1} = y(x_{k+1}) - y_{k+1} = \frac{h^2}{2}y''(\xi) \Rightarrow T_{k+1} = O(h^2)$

Calculation for global truncation error:

$\bar{y}_{n+1} = y(x_n) + hf(x_n, y(x_n))$, $\varepsilon_{n+1} = y(x_{n+1}) - y_{n+1}$

$|\varepsilon_{n+1}| = |y(x_{n+1}) - y_{n+1}| = |y(x_{n+1}) - \bar{y}_{n+1} + \bar{y}_{n+1} - y_{n+1}| \leq |T_{n+1}| + |\bar{y}_{n+1} - y_{n+1}|$

And,if $f(x, y)$ satisfies Lipschitz condition.

$|\bar{y}_{n+1} - y_{n+1}| \leq |y(x_n) - y_n| + h|f(x_n, y(x_n)) - f(x_n, y_n)| \leq |y(x_n) - y_n| + hL|y(x_n) - y_n| = (1 + hL)|\varepsilon_n|$

Therefore, $|\varepsilon_{n+1}| \leq \frac{Mh^2}{2} + (1 + hL)|\varepsilon_n|$. Since the initial value is accurate, $\varepsilon_0 = 0$. By recursion, we get $|\varepsilon_n| \leq \frac{Mh}{2L}((1 + hL)^n - 1) \leq \frac{Mh}{2L}(e^{nhL} - 1)$

The local truncation error e_n is $O(h^2)$. e_n : $\frac{y''(c)h^2}{2!}$, $|e_n| \leq \frac{1}{2}Mh^2$, where M is the maximum of $y''(c)$.
The global truncation error E_n is $O(h)$.

3.2 Improved Euler Method

Algorithm:

$$\begin{cases} y_{n+1} = y_n + \frac{h}{2}(K_1 + K_2) \\ K_1 = f(x_n, y_n) \\ K_2 = f(x_n + h, y_n + K_1) \end{cases}$$

Calculation for local truncation error: $y_{k+1} = y(x_k) + \frac{h}{2}[y'(x_k) + y'(x_{k+1})]$

$y(x_{k+1}) = y(x_k) + hy'(x_k) + \frac{h^2}{2}f''(x_k) + \frac{h^3}{6}f'''(x_k) + O\left(h^4\right)$

$y'(x_{k+1}) = y'(x_k) + hy''(x_k) + \frac{h^2}{2}f'''(x_k) + O\left(h^3\right)$

$T_{k+1} = -\frac{h^3}{12}f'''(x_k) + O\left(h^4\right) = O\left(h^3\right)$

The local truncation error e_n : $O(h^3)$.

The global truncation error E_n is $O(h^2)$.

3.3 The Runge-Kutta Method

$$\begin{cases} y_{n+1} = y_n + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4) \\ K_1 = f(x_n, y_n) \\ K_2 = f(x_n + \frac{h}{2}, y_n + \frac{h}{2}K_1) \\ K_3 = f(x_n + \frac{h}{2}, y_n + \frac{h}{2}K_2) \\ K_4 = f(x_n + h, y_n + hK_3) \end{cases}$$

The local truncation error e_n : $O(h^5)$.

The global truncation error E_n is $O(h^4)$.

Forth-order Runge-Kutta method: Local truncation error:

$$IVP1:$$

Because the equation is too complex to simplify, I use the value calculated by Forth-order Runge-Kutta method in which $y_n = 2.30443312$, $t = 0.5$ and different h to calculate the difference Diff between the Forth-order Runge-Kutta method and the Sixth Talor equation.

$$h = 0.1, Diff1 = 1.1464433787189421 * 10^{-5}$$

$$h = 0.01, Diff2 = 5.434075411869799 * 10^{-10}$$

$$h = 0.001, Diff3 = 4.440892098500626 * 10^{-15}$$

There is no item without h in Diff. And y and t don't change. So the differences between Diff1, Diff2 and Diff3 are caused by h. For the $item I_{first}$ with the smallest power of h, $I_{first} = L1 * h^a$, $a = 5$ if $10L_n > L_{n+1}$. Because when h is divided by 10, power of 10 of Diff decreases 5. And if $10L_n > L_{n+1}$, $L_n * h^a > L_{n+1} * h^{a+1}$. So the main change in Diff is caused by I_{first}. However, $10L_n > L_{n+1}$ is possible to meet.But for the difference between Diff1 and Diff3, the difference in power of 10 is 10. For the item $I_{first} = L1 * h^a$, $a = 5$ if $100L_n > L_{n+1}$. According to the rough equation of Diff, $100L_n > L_{n+1}$ is easy to meet. So $I_{first} = L1 * h^5$. The local truncation error of Forth-order Runge-Kutta method is $O(h^5)$.

$$IVP2:$$

$$y_n = 2.31591594$$

$$t = 0.5$$

$$h = 0.1, Diff1 = -6.223411617567685 * 10^{-1}$$

$$h = 0.01, Diff2 = -9.323121125781597 * 10^{-6}$$

$$h = 0.001, Diff3 = -8.418914454466631 * 10^{-10}$$

The same reason as before, the local truncation error of Forth-order Runge-Kutta method is $O(h^5)$.

Global truncation error:

First I calculate the difference of Forth-order Runge-Kutta method between ry'_{n+1} which is calculated by the real value ry_n at t, and y_{n+1} which is calculated by y_n.

$$u_n = ry'_n - y_n$$

$$A_n = ry_n - y_n(realerror)$$

$$ry'_{n+1} - y_{n+1} = (ry_n + h * f(t, ry'_n)) - (y_n + h * f(t, y_n)) = ry_n - y_n + h * f(t, ry_n) - h * f(t, y_n)$$

$$u_{n+1} = A_n + h * (f(t, ry_n) - f(t, y_n))$$

There are two kinds of items in $f(t, ry_n) - f(t, y_n)$, one is $L * y^a * h^b * t^c (a \neq 0)$, the other is $L * h^b * t^c$. Because t is the same in $f(t, ry_n)$ and $f(t, y_n)$, $L * h^b * t^c = 0$ in $f(t, ry_n) - f(t, y_n)$. So $f(t, ry_n) - f(t, y_n)$ only has items in the form of $L * h^b * t^c * (ry_n^a - y_n^a)$.And $ry_n^a - y_n^a = (ry_n - y_n) * (...)$. So

$$u_{n+1} = A_n + h * (ry_n - y_n) * (...) < A_n + h * (ry_n - y_n) * L = A_n + h * A_n * L$$

$$u_{n+1} = A_n * (1 + h * L)$$

Because ry'_{n+1} is calculated by Forth-order Runge-Kutta method, real value $ry_{n+1} = ry'_{n+1} + localtruncationerror = ry'_{n+1} + L * h^5$

$$A_{n+1} = ry_{n+1} - y_{n+1} = L * h^5 + u_{n+1} = L * h^5 + A_n * (1 + h * L)$$

$$A_{n+1} <= L * h^5 + L * h^5 * (1 + h * L) + L * h^5 * (1 + h * L)^2 + L * h^5 * (1 + h * L)^3 + ... + L * h^5 * (1 + h * L)^n$$

$$n = \frac{length - of - domain}{h} = \frac{L}{h}$$

$$A_{n+1} < L * h^5 * n = \frac{L * h^5 * L}{h} = L * h^4$$

Global truncation error of Forth-order Runge-Kutta method is O(h^4).

3.4 Error in Particular IVP

Since the value we got from power series is relatively close to the real value, we regard these value as real value. And then analysis the error in different algorithms. We choose to compare the y value when $t = 0.5$. And the steps in all algorithms are 0.01.

Table 7: IVP1

	t=0.5	Absolute Error	Relative Error
Power Series	2.30425965	≈ 0	≈ 0
Euler	2.21805115	0.08620850	3.741%
Improved Euler	2.26132714	0.04293251	1.863%
Runge-Kutta	2.26165698	0.04260267	1.849%

Table 8: IVP2

	t=0.5	Absolute Error	Relative Error
Power Series	2.31659622	≈ 0	≈ 0
Euler	2.09124794	0.22534828	9.728%
Improved Euler	2.17046596	0.14613026	6.308%
Runge-Kutta	2.17181843	0.14477779	6.250%

Even if they are just particular values, we can easily draw the conclution that accuracy from high to low is: Runge-Kutta, Improved Euler, Euler, which according to our theoretical calculation.

4 Appendix

```python
#The first problem (Euler)
import numpy as np
import matplotlib.pyplot as plt
# the given function
def fx(y, x):
    return y**2+x**2+x
def ode_euler_right(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)
    y = np.empty((ts.size, y0.size))
    y[0, :] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):
        y[i + 1, :] = y[i, :] + h * f(y[i, :], t)
    return y, ts
def ode_euler_left(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)
    y = np.empty((ts.size, y0.size))
    y[0,:] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):
        y[i+1,:] = y[i,:] + h * f(y[i,:], t)
    return y, ts
def Euler():
    range_right = 0.91
    range_left = -2.81
    step = 0.01
    y, ts = ode_euler_right(fx, 1, range_right, step)
    y2,ts2 = ode_euler_left(fx,1,range_left,-step)
    plt.figure()
    plt.plot(ts, y,"-b",ts2, y2,"-b")
    plt.xlabel('time [s]')
    plt.title('Solution to the equation')
    plt.show()
Euler()
```

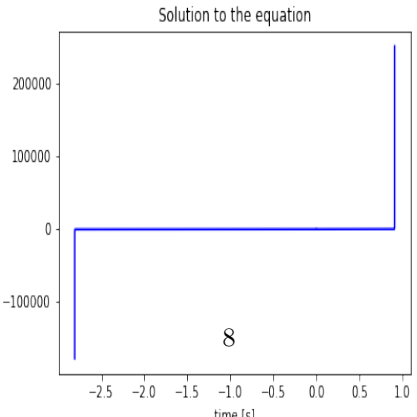
Figure 1: Euler for IVP1

```python
#The first problem (improved Euler)
import numpy as np
import matplotlib.pyplot as plt
# the given function
def fx(y, x):
    return y**2+x**2+x
def ode_euler_right(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)    #0---tf step=h
    y = np.empty((ts.size, y0.size))
    y[0, :] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):    #i
        y[i + 1, :] = y[i, :] + h * f(y[i, :], t)
        k1 = f(y[i,:],t)
        k2 = f(y[i+1,:],t+h)
        k_use =  1/2*(k1+k2)
        y[i + 1, :] = y[i, :] + h * k_use
    return y, ts
def ode_euler_left(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)    #0---tf step=h
    y = np.empty((ts.size, y0.size))
    y[0] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):    #i
        y[i + 1, :] = y[i, :] + h * f(y[i, :], t)
        k1 = f(y[i,:],t)
        k2 = f(y[i+1,:],t+h)
        k_use =  1/2*(k1+k2)
        y[i + 1, :] = y[i, :] + h * k_use
    return y, ts
def improved_Euler():
    range_right = 0.92
    range_left = -2.81
    step = 0.01
    y, ts = ode_euler_right(fx, 1, range_right, step)
    y2,ts2 = ode_euler_left(fx,1,range_left,-step)
    plt.figure()
    plt.plot(ts, y,"-b",ts2, y2,"-b")
    plt.xlabel('time [s]')
    plt.title('Solution to the equation')
    plt.show()
improved_Euler()
```

Figure 2: Improved Euler for IVP1

```python
#The first problem (Runge-Kutta)
import numpy as np
import matplotlib.pyplot as plt
# the given function
def fx(y, x):
    return y**2+x**2+x
def ode_euler_right(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)    #0---tf step=h
    y = np.empty((ts.size, y0.size))
    y[0, :] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):    #i
        k1 = f(y[i,:],t)
        #y_temp1 = y[i, :] + h/2 * k1
        k2 = f(y[i, :] + h/2 * k1,t+h/2)
        k3 = f(y[i, :]+h/2*k2,t+h/2)
        k4 = f(y[i, :]+h*k3,t+h)
        y[i + 1, :] = y[i, :] + (h/6) * (k1+2*k2+2*k3+k4)
    return y, ts
def f(t,y):
    return 2*y**3+2*y*t**2+2*y*t+2*t+1
def ode_euler_left(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)    #0---tf step=h
    y = np.empty((ts.size, y0.size))
    y[0] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):    #i
        k1 = f(y[i,:],t)
        k2 = f(y[i, :] + h/2 * k1,t+h/2)
        k3 = f(y[i, :]+h/2*k2,t+h/2)
        k4 = f(y[i, :]+h*k3,t+h)
        y[i + 1, :] = y[i, :] + (h/6) * (k1+2*k2+2*k3+k4)
    return y, ts
def Runge_Kutta():
    range_right = 0.910
    range_left = -2.80925
    step = 0.0001
    y, ts = ode_euler_right(fx, 1, range_right, step)
    y2,ts2 = ode_euler_left(fx,1,range_left,-step)
    plt.figure()
    plt.plot(ts, y,"-b",ts2, y2,"-b")
    plt.xlabel('time [s]')
    plt.title('Solution to the equation')
    plt.show()
Runge_Kutta()
```

Figure 3: Runge-Kutta for IVP1



```python
#The second problem (Euler)
import numpy as np
import matplotlib.pyplot as plt
# the given function
def fx(y, x):
    return (y-x)*(y**2-x**2)

def ode_euler_right(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)    #0---tf step=h
    y = np.empty((ts.size, y0.size))
    y[0, :] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):    #i
        y[i + 1, :] = y[i, :] + h * f(y[i, :], t)
    return y, ts
def ode_euler_left(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)    #0---tf step=h
    y = np.empty((ts.size, y0.size))
    y[0,:] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):    #i
        y[i+1,:] = y[i,:] + h * f(y[i,:], t)
    return y, ts

def Euler():
    range_right = 0.62
    range_left = -10
    step = 0.001
    print('Solving Newton Cooling ODE...')
    y, ts = ode_euler_right(fx, 1, range_right, step)
    y2,ts2 = ode_euler_left(fx,1,range_left,-step)
    print('Done.')
    plt.figure()
    plt.plot(ts, y,"-b",ts2, y2,"-b")
    plt.xlabel('time [s]')
    plt.title('Solution to the equation')
    plt.show()

Euler()
```

Figure 5: Euler for IVP2

```python
#The second problem (improved Euler)
import numpy as np
import matplotlib.pyplot as plt
# the given function
def fx(y, x):
    return (y-x)*(y**2-x**2)

def ode_euler_right(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)    #0---tf step=h
    y = np.empty((ts.size, y0.size))
    y[0, :] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):    #i
        y[i + 1, :] = y[i, :] + h * f(y[i, :], t)
        k1 = f(y[i,:],t)
        k2 = f(y[i+1,:],t+h)
        k_use =  1/2*(k1+k2)
        y[i + 1, :] = y[i, :] + h * k_use
    return y, ts
def ode_euler_left(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)    #0---tf step=h
    y = np.empty((ts.size, y0.size))
    y[0] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):    #i
        y[i + 1, :] = y[i, :] + h * f(y[i, :], t)
        k1 = f(y[i,:],t)
        k2 = f(y[i+1,:],t+h)
        k_use =  1/2*(k1+k2)
        y[i + 1, :] = y[i, :] + h * k_use
    return y, ts

def improved_Euler():
    range_right = 0.61
    range_left = -14
    step = 0.001
    print('Solving Newton Cooling ODE...')
    y, ts = ode_euler_right(fx, 1, range_right, step)
    y2,ts2 = ode_euler_left(fx,1,range_left,-step)
    #print(y2)
    print('Done.')
    plt.figure()
    plt.plot(ts, y,"-b",ts2, y2,"-b")
    plt.xlabel('time [s]')
    plt.title('Solution to the equation')
    plt.show()

improved_Euler()
```

Figure 6: Improved Euler for IVP2

```python
#The second problem (Runge-Kutta)
import numpy as np
import matplotlib.pyplot as plt
# the given function
def fx(y, x):
    return (y-x)*(y**2-x**2)

def ode_euler_right(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)    #0---tf step=h
    y = np.empty((ts.size, y0.size))
    y[0, :] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):    #i
        k1 = f(y[i,:],t)
        k2 = f(y[i, :]+h/2*k1,t+h/2)
        k3 = f(y[i, :]+h/2*k2,t+h/2)
        k4 = f(y[i, :]+h*k3,t+h)
        y[i + 1, :] = y[i, :] + (h/6) * (k1+2*k2+2*k3+k4)
    return y, ts


def ode_euler_left(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)    #0---tf step=h
    y = np.empty((ts.size, y0.size))
    y[0] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):    #i
        k1 = f(y[i,:],t)
        k2 = f(y[i, :] + h/2 * k1,t+h/2)
        k3 = f(y[i, :]+h/2*k2,t+h/2)
        k4 = f(y[i, :]+h*k3,t+h)
        y[i + 1, :] = y[i, :] + (h/6) * (k1+2*k2+2*k3+k4)
    return y, ts

def Runge_Kutta():
    range_right = 0.61
    range_left = -15
    step = 0.001
    print('Solving Newton Cooling ODE...')
    y, ts = ode_euler_right(fx, 1, range_right, step)
    y2,ts2 = ode_euler_left(fx,1,range_left,-step)
    print('Done.')
    plt.figure()
    plt.plot(ts, y,"-b",ts2, y2,"-b")
    plt.xlabel('time [s]')
    plt.title('Solution to the equation')
    plt.show()

Runge_Kutta()
```

Figure 7: Runge-Kutta for IVP2

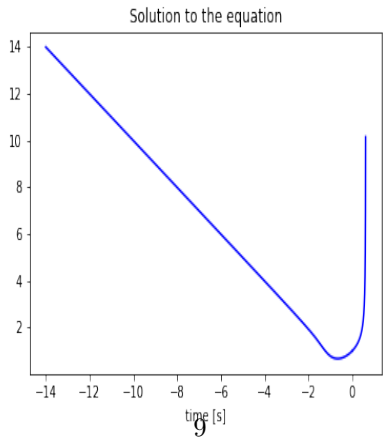


Figure 8: IVP2

```python
#the first one
#Euler plot with power series
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
an = np.zeros(1000)
an[0] = 1
an[1] = 1
an[2] = 3/2
an[3] = 5/3
def multi(i,n):
    return an[i]*an[n-i]

def sum(n):
    num = 0
    for i in range(n):
        num = num+  multi(i,n-1)
    return num/n
def multi(i,n):

    return an[i]*an[n-i]
for k in range(996):
    an[k+4] = sum(k+4)

import numpy as np
import matplotlib.pyplot as plt
def fx(y, x):
    return y**2+x**2+x
def ode_euler_right(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)    #0---tf step=h
    y = np.empty((ts.size, y0.size))
    y[0, :] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):    #i
        y[i + 1, :] = y[i, :] + h * f(y[i, :], t)
    return y, ts
def ode_euler_left(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)    #0---tf step=h
    y = np.empty((ts.size, y0.size))
    y[0,:] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):    #i
        y[i+1,:] = y[i,:] + h * f(y[i,:], t)
    return y, ts
def Euler():
    y3=0
    t3  = np.arange(0, 0.91 + 0.01, 0.01)
    for i in range(1000):
        y3 = y3 + an[i]*t3**i
    range_right = 0.91
    range_left = -2.81
    step = 0.01
    y, ts = ode_euler_right(fx, 1, range_right, step)
    y2,ts2 = ode_euler_left(fx,1,range_left,-step)
    plt.figure()
    plt.plot(ts, y,"-b",ts2, y2,"-b",t3,y3,"--r")
    plt.plot(ts,y,color = "blue",linestyle = "-",label = "Dot")
    plt.xlabel('time [s]')
    plt.title('Solution to the equation')
    plt.savefig("power series1.png")
    plt.show()

Euler()
```

```python
#the second one
#Euler plot with power series
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
an = np.zeros(400)
an[0] = 1
an[1] = 1
an[2] = 1
an[3] = 1
an[4] = 7/4
def sum(n):
    num = 0
    for i in range(n):
        for j in range(n-i):
            num = num+  multi(i,j,n-1)
    for i in range(n-1):
        num=num - multi2(i,n-1)
    num = num-an[n-3]
    return num/n

def multi(i,j,n):
    return an[i]*an[j]*an[n-i-j]

def multi2(i,n):
    return an[i]*an[n-1-i]

sum(5)
for k in range(395):
    an[k+5] = sum(k+5)
y3=0
t3  = np.arange(0, 0.91 + 0.01, 0.01)
for i in range(400):
    y3 = y3 + an[i]*t3**i
import numpy as np
import matplotlib.pyplot as plt
def fx(y, x):
    return (y-x)*(y**2-x**2)
def ode_euler_right(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)
    y = np.empty((ts.size, y0.size))
    y[0, :] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):    #i
        y[i + 1, :] = y[i, :] + h * f(y[i, :], t)
    return y, ts
def ode_euler_left(f, y0, tf, h):
    y0 = np.array(y0)
    ts = np.arange(0, tf + h, h)
    y = np.empty((ts.size, y0.size))
    y[0,:] = y0
    for t, i in zip(ts[1:], range(ts.size - 1)):    #i
        y[i+1,:] = y[i,:] + h * f(y[i,:], t)
    return y, ts
def Euler():

    y3=0
    t3  = np.arange(0, 0.62 + 0.01, 0.01)
    for i in range(200):
        y3 = y3 + an[i]*t3**i
    range_right = 0.62
    range_left = -10
    step = 0.001
    y, ts = ode_euler_right(fx, 1, range_right, step)
    y2,ts2 = ode_euler_left(fx,1,range_left,-step)

    plt.figure()
    plt.plot(ts, y,"-b",ts2, y2,"-b")
    plt.plot(t3,y3,"--r")
    plt.xlabel('time [s]')
    plt.title('Solution to the equation')
    plt.savefig("power series2new.png")
    plt.show()
Euler()
```

Figure 9: Plot Power series and Euler in one picture

```python
#第一题
import numpy as np
import matplotlib.pyplot as plt
# 定义求解函数 y_dot = y + 2*x/(y*y)
def f(y, x):
    return y**2+x**2+x
    #return -y
# 算法定义

t=0.5
h=0.01
k1 = f(2.30443312,t)
#y_temp1 = y[i, :] + h/2 * k1
k2 = f(2.30443312 + h/2 * k1,t+h/2)
k3 = f(2.30443312+h/2*k2,t+h/2)
k4 = f(2.30443312+h*k3,t+h)
u = 2.30443312 + (h/6) * (k1+2*k2+2*k3+k4)
w=2.30443312
a=w**2+t**2+t
b=2*w*a+2*t+1
c=2*a**2+2*w*b+2
d=6*a*b+2*w*c
e=6*b**2+8*a*c+2*w*d
g=12*b*c+8*b*c+10*a*d+2*w*e
r=w+a*h+b*h**2/2+c*h**3/3/2+d*h**4/4/3/2+e*h**5/5/4/3/2+g*h**6/6/5/4/3/2
print(a)
print(b/2)
print(c/2/3)
print(d/2/3/4)
print(e/2/3/4/5)
print(g/2/3/4/5/6)
print(r-u)
print("yyy")
```

```python
#第二题
import numpy as np
import matplotlib.pyplot as plt
# 定义求解函数 y_dot = y + 2*x/(y*y)
def f(y, x):
    return (y-x)*(y**2-x**2)
    #return -y
# 算法定义

t=0.5
h=0.01
k1 = f(1,t)
k2 = f(1 + h/2 * k1,t+h/2)
k3 = f(1+h/2*k2,t+h/2)
k4 = f(1+h*k3,t+h)
u = 1 + (h/6) * (k1+2*k2+2*k3+k4)
w=1
a=(w-t)*(w**2-t**2)
b=3*w**2*a-2*w*t-a*t**2-w**2-2*w*a*t+3*t**2
c=6*w*a**2+3*w**2*b-2*a*t-2*w-b*t**2-2*a*t-4*w*a-2*(w*b+a**2)*t+6*t
d=12*w*a*b+6*w*a+3*w**2*c-2*b*t-6*a-c*t**2-4*b*t-4*a**2-4*w*b-2*w*c*t-6*a*b*t-2*w*b-2*a**2+6
e=12*a*b+12*w*(b**2+a*c)+6*a**2+6*w*b+6*w*a*c+3*w**2*d-2*b-8*c*t-6*b-8*w*c-2*t*(w*d+a*c)-6*t*(b**2+a*c)-24*a*b
g=24*a*b**2+12*a**2*c+12*a*(b**2+a*c)+12*w*(2*b*c+a*d+b*c)+12*a*b+6*a*b+6*w*c+6*a**2*c+6*w*(a*d+b*c)+6*w*a*d+3*w**2*e
-10*c-8*d*t-6*c-2*t*d-e*t**2-4*c-8*w*d-2*(w*d+a*c)-2*t*(a*d+w*e+b*c+a*d)-6*(b**2+a*c)-6*t*(2*b*c+b*c+a*d)-24*a*c-24*b**2
r=w+a*h+b*h**2/2+c*h**3/3/2+d*h**4/4/3/2+e*h**5/5/4/3/2
print(a)
print(b/2)
print(c/2/3)
print(d/2/3/4)
print(e/2/3/4/5)
print(g/2/3/4/5/6)
print(r-u)
print("yyy")
```

Figure 10: difference for Runge-Kutta