数值分析上机报告

常微分方程数值解

1 问题描述

本次实验考虑的是一个二维线性自治系统：

$$\begin{cases} \frac{\mathrm{d}u}{\mathrm{d}t} = -2000u(t) + 999.75v(t) + 1000.25 \\ \frac{\mathrm{d}v}{\mathrm{d}t} = u(t) - v(t). \end{cases} \tag{1}$$

初始值为 $u(0) = 0, v(0) = -2$.

为了简单起见，其也可以写成如下的线性形式：

$$\begin{pmatrix} u'(t) \\ v'(t) \end{pmatrix} = \mathbf{A} \begin{pmatrix} u(t) \\ v(t) \end{pmatrix} + \mathbf{b}, \tag{2}$$

这里的

$$\mathbf{A} = \begin{pmatrix} -2000 & 999.75 \\ 1 & -1 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 1000.25 \\ 0 \end{pmatrix}. \tag{3}$$

初始值为：

$$\begin{pmatrix} u(0) \\ v(0) \end{pmatrix} = \begin{pmatrix} 0 \\ -2 \end{pmatrix}. \tag{4}$$

根据平面自治系统的相关理论，我们很容易得到该 ODE 方程组的解析解：

$$\begin{cases} u(t) &= -1.499875e^{-0.5t} + 0.499875e^{-2000.5t} + 1 \\ v(t) &= -2.99975e^{-0.5t} - 0.00025e^{-2000.5t} + 1. \end{cases} \tag{5}$$

在本次实验中，我们将分别使用四阶古典 Runge-Kutta 格式和隐式二级四阶 Runge-Kutta 格式对该线性系统进行数值求解，并与精确解相比较.

2 Runge-Kutta 方法概述

Runge-Kutta 法是数值求解常微分方程的一种高阶单步格式，我们可以从数值积分的角度对其进行推导. 事实上，线性系统总是可以改写成如下的积分形式：

$$\mathbf{x}(t) = \mathbf{x}(0) + \int_0^t \mathbf{f}(\mathbf{x}(s))\mathrm{d}s, \tag{6}$$

这里 $\mathbf{x}(t)=(u(t),v(t))^{\mathrm{T}}$，$\mathbf{f}(\mathbf{x})=\mathbf{A}\mathbf{x}+\mathbf{b}$（注意这是一个自治系统，$\mathbf{f}$ 不显含时间 t）.

这样，我们就有了单步递推公式：

$$\mathbf{x}(t_{n+1})=\mathbf{x}(t_n)+\int_{t_n}^{t_{n+1}}\mathbf{f}(\mathbf{x}(s))\mathrm{d}s. \tag{7}$$

一般而言，为了高精度地近似计算上面的积分，可以在 $[t_n,t_{n+1}]$ 上引入若干新的节点：$t_n\leq s_1\leq\cdots\leq s_m\leq t_{n+1}$，记 $s_j=t_n+a_jh$. 如果我们知道了 $\mathbf{f}$ 在 $(\mathbf{x})$ 的值 K_j，则可以用公式：

$$\mathbf{x}_{n+1}=\mathbf{x}_n+h\sum_{j=1}^{m}c_jK_j. \tag{8}$$

来计算 $\mathbf{x}_{n+1}$，其中 K_j 为数值积分公式

$$\int_{t_n}^{t_{n+1}}\mathbf{f}(\mathbf{x}(s))\mathrm{d}s\approx h\sum_{j=1}^{m}c_jK_j=h\sum_{j=1}^{m}c_j\mathbf{f}(\mathbf{x}(s_j)) \tag{9}$$

中的求积系数. 而计算 $\mathbf{x}(s_j)=\mathbf{x}(t_n)+\int_{t_n}^{t_n+a_jh}\mathbf{f}(\mathbf{x}(s))\mathrm{d}s$ 又有隐式和显式两种选择. 如果选用显式格式，则是采用 $\mathbf{f}$ 在 $s_1,\cdots,s_{j-1}$ 上的值为插值来计算积分，也就是：

$$\mathbf{x}(s_j)\approx\mathbf{x}_n+h\sum_{\ell=1}^{j-1}b_{j\ell}\mathbf{f}(\mathbf{x}(s_\ell)). \tag{10}$$

将其代入 K_j 的表达式即可得到：

$$K_j=\mathbf{f}\left(\mathbf{x}_n+h\sum_{\ell=1}^{j-1}b_{j\ell}K_\ell\right),\quad j=1,2,\cdots,m. \tag{11}$$

由此得到完整的显式 Runge-Kutta 格式：

$$\begin{cases}\mathbf{x}_{n+1}=\mathbf{x}_n+h\sum_{j=1}^{m}c_jK_j,\\ K_j=\mathbf{f}\left(\mathbf{x}_n+h\sum_{l=1}^{j-1}b_{jl}K_l\right),\quad j=1,\cdots,m.\end{cases} \tag{12}$$

有了 Runge-Kutta 格式的一般格式，我们对差分格式进行 Taylor 展开，再通过比较系数即可得到我们所需要的阶数的高阶格式. 例如，可以查得 4 阶显式 Runge-Kutta 方法的系数如下：

$$\begin{gathered}\mathbf{a}=\left(0,\frac{1}{2},\frac{1}{2},1\right),\quad \mathbf{c}=\left(\frac{1}{6},\frac{1}{3},\frac{1}{3},\frac{1}{6}\right),\\ b_{21}=\frac{1}{2},b_{31}=0,b_{32}=\frac{1}{2},b_{41}=0=b_{42},b_{43}=1.\end{gathered} \tag{13}$$

对于隐式 Runge-Kutta 方法，就是用 $\mathbf{f}$ 在所有 s_ℓ 上的值为插值节点来计算数值积分，也就是：

$$\mathbf{x}(s_j)\approx\mathbf{x}_n+h\sum_{\ell=1}^{m}b_{j\ell}\mathbf{f}(\mathbf{x}(s_\ell)). \tag{14}$$

代入 K_j 表达式可得隐式 Runge-Kutta 格式：

$$\begin{cases} \mathbf{x}_{n+1} = \mathbf{x}_n + h\sum_{j=1}^{m} c_j K_j, \\ K_j = \mathbf{f}\left(\mathbf{x}_n + h\sum_{l=1}^{m} b_{jl} K_l\right), \quad j = 1, \cdots, m. \end{cases} \tag{15}$$

对于二级四阶 Runge-Kutta 方法，其具体系数为：

$$\begin{cases} \mathbf{x}_{n+1} &= \mathbf{x}_n + \frac{h}{2}\left(K_1 + K_2\right) \\ K_1 &= \mathbf{f}\left(\mathbf{x}_n + \frac{1}{4}hK_1 + \left(\frac{1}{4} - \frac{\sqrt{3}}{6}\right)hK_2\right) \\ K_2 &= \mathbf{f}\left(\mathbf{x}_n + \left(\frac{1}{4} + \frac{\sqrt{3}}{6}\right)hK_1 + \frac{1}{4}hK_2\right). \end{cases} \tag{16}$$

一般来说，在隐式 Runge-Kutta 方法中，我们无法显式计算 K_j，只能通过求解方程组来得到. 好在，在我们的问题中，这总是一个线性方程组，可以直接求解如下：

$$K_2 = \left(\mathrm{Id} - \left(\frac{1}{4} + \frac{\sqrt{3}}{6}\right)h\mathbf{A}\left(\mathrm{Id} + \frac{\sqrt{3}}{6}h\mathbf{A}\right)^{-1}\left(\mathrm{Id} - \frac{\sqrt{3}}{6}h\mathbf{A}\right) - \frac{1}{4}h\mathbf{A}\right)^{-1}\left(\mathbf{A}\mathbf{y}_n + \mathbf{b}\right), \tag{17}$$

$$K_1 = \left(\mathrm{Id} + \frac{\sqrt{3}}{6}h\mathbf{A}\right)^{-1}\left(\mathrm{Id} - \frac{\sqrt{3}}{6}h\mathbf{A}\right)K_2. \tag{18}$$

在下一个部分中，我们将示出具体的代码实现.

3 代码实现

Python 代码实现如下：

```python
import numpy as np

A = np.array([[-2000, 999.75],[1, -1]])
b = np.array([1000.25,0])

def f(x):
    return A@x + b

x0 = np.array([0,-2])

def s(t):
    return np.array([-1.499875*np.exp(-0.5*t)+0.499875*np.exp(-2000.5*t)+1,-2.99975*
    np.exp(-0.5*t)-0.00025*np.exp(-2000.5*t)+1])

# classical RK4
def cl_rk4(x0, t0, tn, n):
```

```
    h = tn / n
    print('\n'+'='*15+'classical_rk4'+'='*15)
    print('-'*43)
    print('step(n) u_n v_n u(t_n) v(t_n)')
    print('-'*43)
    print('\n{:.1f} {:.6f} {:.6f} {:.6f} {:.6f}'.format(0, x0[0], x0[1], s(0)[0], s(0)[1]))
    for i in range(n):
        k1 = f(x0)
        k2 = f(x0+h*k1/2)
        k3 = f(x0+h*k2/2)
        k4 = f(x0+h*k3)
        k = k1/6 + k2/3 + k3/3 + k4/6
        xn = x0 + h*k
        x0 = xn
        t0 += h
        if i%(n/20) == 0 and i != 0:
            print('\n{:.1f} {:.6f} {:.6f} {:.6f} {:.6f}'.format(t0, xn[0], xn[1], s(t0)[0], s(t0)[1]))

# implicit RK4
def im_rk4(x0, t0, tn, n):
    h = tn / n
    print('\n'+'='*15+'implicit_rk4'+'='*15)
    print('-'*42)
    print('step(n) u_n v_n u(t_n) v(t_n)')
    print('-'*42)
    print('\n{:.1f} {:.6f} {:.6f} {:.6f} {:.6f}'.format(0, x0[0], x0[1], s(0)[0], s(0)[1]))
    xn = x0
    for i in range(n):
        B = np.eye(2)-(1/4 + np.sqrt(3)/6)*h*A@np.linalg.pinv(np.eye(2)+(np.sqrt(3)/6)*h*A, rcond = 1e-5)@(np.eye(2)-(np.sqrt(3)/6*h*A))
        K2 = np.linalg.pinv(B, rcond = 1e-5)@(A@xn + b)
        K1 = np.linalg.pinv(np.eye(2)+(np.sqrt(3)/6)*h*A, rcond = 1e-5)@(np.eye(2)-(np.sqrt(3)/6)*h*A)@K2
        xn = x0 + (h/2)*(K1+K2)
        x0 = xn
        t0 += h
        if i%(n/20) == 0 and i != 0:
            print('\n{:.1f} {:.6f} {:.6f} {:.6f} {:.6f}'.format(t0, xn[0], xn[1], s(t0)[0], s(t0)[1]))
```

4 结果

我们首先取 $h = 0.01$，运行结果如下：

```
===============classical_rk4===============
-------------------------------------------
step(n) u_n v_n u(t_n) v(t_n)
-------------------------------------------

0.0 0.000000 -2.000000 0.000000 -2.000000

1.0 nan nan 0.094817 -0.810366

2.0 nan nan 0.450979 -0.098042

3.0 nan nan 0.667002 0.334004

4.0 nan nan 0.798026 0.596053

5.0 nan nan 0.877497 0.754994

6.0 nan nan 0.925698 0.851396

7.0 nan nan 0.954934 0.909867

8.0 nan nan 0.972666 0.945332

9.0 nan nan 0.983421 0.966842

10.0 nan nan 0.989944 0.979889

11.0 nan nan 0.993901 0.987802

12.0 nan nan 0.996301 0.992601

13.0 nan nan 0.997756 0.995513

14.0 nan nan 0.998639 0.997278

15.0 nan nan 0.999175 0.998349

16.0 nan nan 0.999499 0.998999

```

```
17.0 nan nan 0.999696 0.999393

18.0 nan nan 0.999816 0.999632

19.0 nan nan 0.999888 0.999777

===============implicit_rk4===============
------------------------------------------
step(n) u_n v_n u(t_n) v(t_n)
------------------------------------------

0.0 0.000000 -2.000000 0.000000 -2.000000

1.0 0.095388 -0.809224 0.094817 -0.810366

2.0 0.451668 -0.096664 0.450979 -0.098042

3.0 0.667628 0.335255 0.667002 0.334004

4.0 0.798532 0.597064 0.798026 0.596053

5.0 0.877880 0.755759 0.877497 0.754994

6.0 0.925977 0.851953 0.925698 0.851396

7.0 0.955131 0.910261 0.954934 0.909867

8.0 0.972802 0.945605 0.972666 0.945332

9.0 0.983514 0.967028 0.983421 0.966842

10.0 0.990007 0.980014 0.989944 0.979889

11.0 0.993943 0.987885 0.993901 0.987802

12.0 0.996328 0.992657 0.996301 0.992601

13.0 0.997774 0.995549 0.997756 0.995513

14.0 0.998651 0.997302 0.998639 0.997278

15.0 0.999182 0.998365 0.999175 0.998349
```

```
82
83 16.0 0.999504 0.999009 0.999499 0.998999
84
85 17.0 0.999700 0.999399 0.999696 0.999393
86
87 18.0 0.999818 0.999636 0.999816 0.999632
88
89 19.0 0.999890 0.999779 0.999888 0.999777
```

可见，显式格式出现了严重的数值发散现象，而隐式格式则已经有一定位数的有效数字. 为了解决数值发散问题，我们进一步减小步长为 $h = 0.001$，此时结果为：

```
1 ===============classical_rk4===============
2 ---------------------------------------------
3 step(n) u_n v_n u(t_n) v(t_n)
4 ---------------------------------------------
5
6 0.0 0.000000 -2.000000 0.000000 -2.000000
7
8 1.0 0.090735 -0.818531 0.090735 -0.818531
9
10 2.0 0.448503 -0.102995 0.448503 -0.102995
11
12 3.0 0.665500 0.331000 0.665500 0.331000
13
14 4.0 0.797115 0.594231 0.797115 0.594231
15
16 5.0 0.876944 0.753889 0.876944 0.753889
17
18 6.0 0.925363 0.850726 0.925363 0.850726
19
20 7.0 0.954730 0.909461 0.954730 0.909461
21
22 8.0 0.972543 0.945085 0.972543 0.945085
23
24 9.0 0.983346 0.966692 0.983346 0.966692
25
26 10.0 0.989899 0.979798 0.989899 0.979798
27
28 11.0 0.993873 0.987747 0.993873 0.987747
29
30 12.0 0.996284 0.992568 0.996284 0.992568
31
```

```
13.0 0.997746 0.995492 0.997746 0.995492

14.0 0.998633 0.997266 0.998633 0.997266

15.0 0.999171 0.998342 0.999171 0.998342

16.0 0.999497 0.998994 0.999497 0.998994

17.0 0.999695 0.999390 0.999695 0.999390

18.0 0.999815 0.999630 0.999815 0.999630

19.0 0.999888 0.999776 0.999888 0.999776

===============implicit_rk4===============
-----------------------------------------
step(n) u_n v_n u(t_n) v(t_n)
-----------------------------------------

0.0 0.000000 -2.000000 0.000000 -2.000000

1.0 0.090791 -0.818417 0.090735 -0.818531

2.0 0.448572 -0.102857 0.448503 -0.102995

3.0 0.665563 0.331125 0.665500 0.331000

4.0 0.797166 0.594332 0.797115 0.594231

5.0 0.876983 0.753966 0.876944 0.753889

6.0 0.925391 0.850782 0.925363 0.850726

7.0 0.954750 0.909500 0.954730 0.909461

8.0 0.972556 0.945113 0.972543 0.945085

9.0 0.983356 0.966711 0.983346 0.966692

10.0 0.989905 0.979811 0.989899 0.979798

11.0 0.993878 0.987755 0.993873 0.987747
```

```
12.0 0.996287 0.992574 0.996284 0.992568

13.0 0.997748 0.995496 0.997746 0.995492

14.0 0.998634 0.997268 0.998633 0.997266

15.0 0.999172 0.998343 0.999171 0.998342

16.0 0.999498 0.998995 0.999497 0.998994

17.0 0.999695 0.999391 0.999695 0.999390

18.0 0.999815 0.999630 0.999815 0.999630

19.0 0.999888 0.999776 0.999888 0.999776
```

可见此时数值发散现象已经得到解决，其中经典方法已经有 6 位有效数字，隐式方法也有了 4 位以上的有效数字.

我们将结果绘制成图，如图1和2所示：

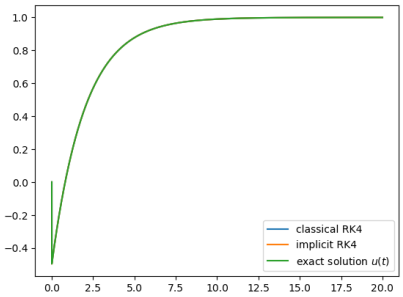

图 1: $u(t)$ 在 $[0, 20]$ 上的解

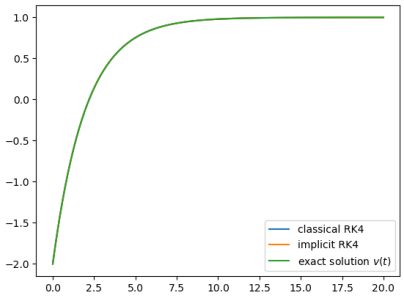

图 2: $v(t)$ 在 $[0, 20]$ 上的解

可见，在 $u(t)$ 中，我们观察到解将以一个非常快的的速度下降到 -0.47 左右，随后再上升，最后在 1 附近达到稳定. 这样的系统我们称为 “刚性系统”. 在这样的系统中，如果选择的步长不够短，则会出现严重的数值发散现象，导致无法得到结果或结果不可信.

5 小结

我们使用显式和隐式两种 RK4 格式对一个刚性系统进行了数值求解，当选取步长 $h = 0.001$ 时，两种格式均得到了精度较高的结果（分别有 6 位和 4 位有效数字），与精确解符合很好.