

目录

1	问题	2
2	求解方程：解析解	2
3	透过三种不同方法求解微分方程	3
3.1	隐式欧拉法	3
3.2	显式欧拉法	4
3.3	4th-order Runge-Kutta	6
4	处理三种方法的误差-代码实现	9
4.1	隐式欧拉法	9
4.2	显式欧拉法	11
4.3	4th-order Runge-Kutta	13
5	处理三种方法的误差-比较	15
5.1	误差分析	16
5.2	收敛性分析	16

使用隐式欧拉法求解常微分方程问题

202011010101物理2001孙陶庵

2023 年 3 月 28 日

1 问题

对于下列微分方程初值问题:

$$\begin{cases} \frac{dy}{dx} = xy^2 + 2y \\ y(0) = -5 \end{cases}$$

使用隐式欧拉法求解其在 $0 < x < 5$ 时的数值解。

2 求解方程：解析解

透过python的sympy库求解

```
1  from sympy import Function, dsolve, Eq, Derivative, sin, cos,
    symbols
2  from sympy.abc import x
3  f = Function('f')
4  dsolve(Derivative(f(x), x) - x * f(x)**2 - 2 * f(x), f(x), ics={f(0):
    -5})
```

Listing 1: 0.1s

可以得到方程解为:

$$f(x) = \frac{4e^{2x}}{-2xe^{2x} + e^{2x} - \frac{9}{5}}$$

3 透过三种不同方法求解微分方程

3.1 隐式欧拉法

```
1  x0 = 0;
2  y0 = -5;
3  a = 0;
4  b = 5;
5  h = 0.2;
6  y = Euler_Implicit(@f, y0, a, b, h);
7
8  x = linspace(a, b, length(y));
9  hold on
10 plot(x, y, '-o');
11 plot(x, RealFunc(x), "-b");
12 xlabel('x');
13 ylabel('y');
14 title('Numerical Solution and error');
15
16 function y = Euler_Implicit(f, y0, a, b, h)
17     n = round((b-a)/h);
18     y = zeros(1, n+1);
19     y(1) = y0;
20
21     for i = 2:n+1
22         xi = a + (i-1) * h;
23         yi = y(i-1);
```

```

24         for j = 1:100
25             yi = y(i-1) + h * f(xi, yi);
26         end
27         y(i) = yi;
28     end
29 end
30
31 function f = f(x, y)
32     f= x.*y.^2 + 2.*y;
33 end
34
35 function ye = RealFunc(x)
36     ye = (20.*exp(2.*x))./(exp(2.*x)-10.*x.*exp(2.*x)-9);
37 end

```

Listing 2: 隐式欧拉法

其中 $function ye = RealFunc(x)$ 部分是文章第一部分所解出的解析解； $function f = f(x, y)$ 部分是原方程，将 $\frac{dy}{dx}$ 替换为 f ； $function y = Euler_Implicit(f, y0, a, b, h)$ 是函数主体，解释如下：

1. 根据步长 h 计算需要多少个点 n ，并初始化结果数组；
2. 将初始值赋给第一个元素；
3. 循环从 $i=2$ 到 $i=n+1$ ，每次计算 xi 和 yi ；
4. 在内部循环中进行100次迭代，利用欧拉隐式法更新 yi 的数值；
5. 将最终得到的 yi 赋给结果数组。

3.2 显式欧拉法

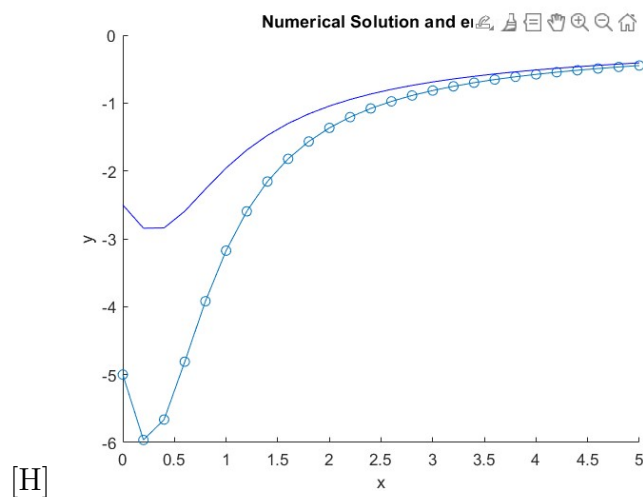


图 1: 隐式欧拉法

```

1  x0 = 0;
2  y0 = -5;
3  h = 0.1;
4  y = Euler_Explicit(@f, y0, h);
5
6  x = linspace(0, 5, length(y));
7  hold on
8  plot(x, y, '-o');
9  plot(x, RealFunc(x), "-b");
10
11 xlabel('x');
12 ylabel('y');
13 title('Numerical Solution and Error');
14 legend('Numerical Solution');
15
16
17 function y = Euler_Explicit(f, y0, h)
18     t0 = 0;
19     tp = 5;

```

```

20     t = t0:h:tp;
21     y = zeros(size(t));
22     y(1) = y0;
23
24     for i = 1:length(t)-1
25         y(i+1) = y(i) + h * f((i+1) * h, y(i));
26     end
27 end
28
29 function fullderi = f(x, y)
30     fullderi = x.*y.^2 + 2*y;
31 end
32
33 function ye = RealFunc(x)
34     ye = (20.*exp(2.*x))./(exp(2.*x)-10.*x.*exp(2.*x)-9);
35 end

```

Listing 3: 显式欧拉法

$function y = Euler_Explicit(f, y_0, h)$ 是函数主体，解释如下： 1. 初始化变量 x_0 、 y_0 和 h

2. 调用 $Euler_Explicit$ 函数计算数值解 y

3. 生成等间隔的自变量 x

4. 绘制数值解曲线和真实解曲线，并添加标签、标题及图例。

其中， $Euler_Explicit$ 函数使用欧拉显式法对微分方程进行离散化处理并求出数值近似解； $f(x, y)$ 为给定的微分方程右侧； $RealFunc(x)$ 为该微分方程的真实解。

3.3 4th-order Runge-Kutta

```

1     y0 = -5;
2     t0 = 0;

```

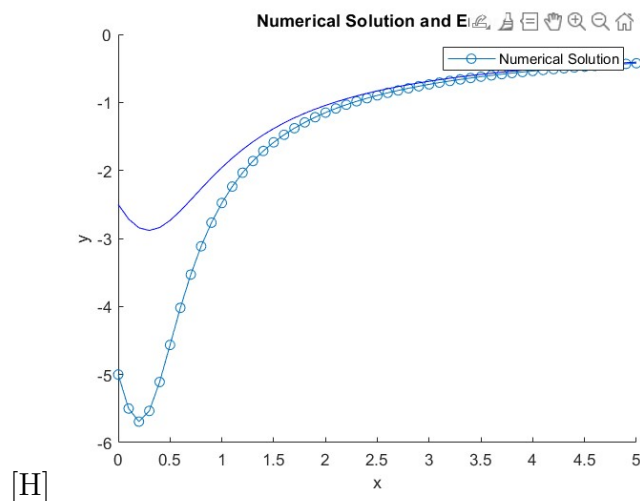


图 2: 显式欧拉法

```

3     tn = 5;
4     h = 0.2;
5
6     result = Runge_Kutta41(@f, t0, y0, tn, h);
7     x = linspace(0, 5, length(result));
8     hold on
9     plot(x, result, '-o');
10    plot(x, RealFunc(x), "-b");
11
12    xlabel('x');
13    legend();
14
15    function res = Runge_Kutta41(f, t0, y0, tn, h)
16        res = [y0];
17        t = t0;
18        for i = 1:floor((tn-t0)/h)
19            k1 = f(t, res(end));
20            k2 = f(t + h / 2, res(end) + h * k1 / 2);
21            k3 = f(t + h / 2, res(end) + h * k2 / 2);

```

```

22     k4 = f(t + h, res(end) + h * k3);
23     y_next = res(end) + h*(k1 + 2 * k2 + 2 * k3 + k4)/6;
24     res = [res, y_next];
25     t = t + h;
26
27 end
28
29 function y = f(x, y)
30     y = x*y^2 + 2*y;
31 end
32
33 function ye = RealFunc(x)
34     ye = (20.*exp(2.*x))./(exp(2.*x)-10.*x.*exp(2.*x)-9);
35 end

```

Listing 4: 4th-order Runge-Kutta

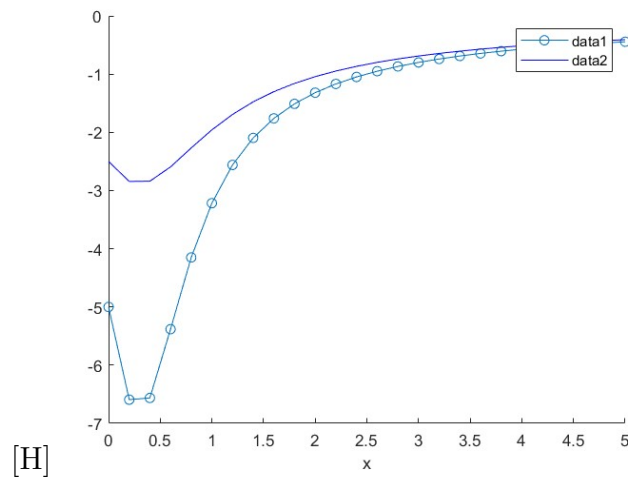


图 3: 4th-order Runge-Kutta

这部分比较简单，主要是透过以下公式进行循环而得：

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

$$k_2 = f\left(t_n + \frac{h}{2}, y_n + h \frac{k_1}{2}\right),$$

$$k_3 = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right),$$

$$k_4 = f(t_n + h, y_n + hk_3).$$

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

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

4 处理三种方法的误差-代码实现

主要是分析误差。

4.1 隐式欧拉法

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4
5  def Euler_Implicit(f, y0, a, b, h):
6      n = round((b-a)/h)
7      y = np.zeros(n+1)
8      y[0] = y0
9
10     for i in range(1, n+1):
11         xi = a + i * h
12         yi = y[i - 1]
13         for j in range(10):
14             yi = y[i - 1] + h * f(xi, yi)
15         y[i] = yi
16
17     return y
18
```

```
19
20 def error(f, y0, a, b, h):
21
22     def y_exact(x):
23         return 20*np.exp(2*x)/(np.exp(2*x) - 10*x*np.exp(2*x) - 9)
24
25     y_num = Euler_Implicit(f, y0, a, b, h)
26     x = np.arange(a, b + h, h)[:len(y_num)]
27     e = y_num - y_exact(x)
28
29     return x, e
30
31
32 def f(x, y):
33     return x*y**2+2*y
34
35
36 x0 = 0
37 y0 = -5
38 a = 0
39 b = 5
40 h = 0.1
41 y = Euler_Implicit(f, y0, a, b, h)
42
43 error_list = []
44 ha = np.arange(0.1, 0.7, 0.01)
45 for h in np.arange(0.1, 0.7, 0.01):
46
47     y = Euler_Implicit(f, y0, a, b, h)
```

```
48     x, e = error(f, y0, a, b, h)
49     x = np.linspace(0, 5, len(y))
50     error_list.append(e[-1])
51     print(h,e[-1])
52
53
54     plt.plot(ha, error_list)
55     plt.xlabel('h')
56     plt.ylabel('e[-1]')
57     plt.title('Error vs. Step Size')
58     plt.show()
```

Listing 5: 隐式欧拉法

4.2 显式欧拉法

```
1     import numpy as np
2     import matplotlib.pyplot as plt
3
4     def Euler_Explicit(f, y0, h):
5         t0 = 0
6         tp = 5
7         t = np.arange(t0, tp+h, h)
8         y = np.zeros(len(t))
9         y[0] = y0
10
11         for i in range(0, len(t)-1):
12             y[i+1] = y[i] + h * f((i+1) * h, y[i])
13         return y
14
```

```
15     def error(f, y0, h):
16         t0 = 0
17         tp1 = 5
18
19         def y_exact(x):
20             return 20*np.exp(2*x)/(np.exp(2*x) - 10*x*np.exp(2*x) - 9)
21
22         y_num = Euler_Explicit(f, y0, h)
23         x = np.arange(t0, tp1 + h, h)
24         e = y_num - y_exact(x)
25
26         return x, e
27
28     def f(x, y):
29         return x*y**2+2*y
30
31     x0 = 0
32     y0 = -5
33
34     error_list = []
35     ha = np.arange(0.26, 1.5, 0.01)
36
37     for h in np.arange(0.26, 1.5, 0.01):
38
39         y = Euler_Explicit(f, y0, h)
40         x, e = error(f, y0, h)
41         x = np.linspace(0, 5, len(y))
42         error_list.append(e[-1])
43         print(h,e[-1])
44
45     xdata = ha
```

```
44 ydata = np.array(error_list)
45 coef = np.polyfit(xdata, ydata, 2)
46 f_fit = np.poly1d(coef)
47 xfit = np.linspace(xdata[0], xdata[-1], 100)
48 yfit = f_fit(xfit)
49
50 plt.plot(ha, error_list, 'bo', label='data')
51 plt.plot(xfit, yfit, 'r-', label='fit')
52 plt.xlabel('h')
53 plt.ylabel('e[-1]')
54 plt.title('Error vs. Step Size')
55 plt.legend()
56 plt.show()
57 plt.plot(ha, error_list)
58 plt.xlabel('h')
59 plt.ylabel('e[-1]')
60 plt.title('Error vs. Step Size')
61 plt.show()
```

Listing 6: 显式欧拉法

4.3 4th-order Runge-Kutta

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def Runge_Kutta4(f, t0, y0, tn, h):
5     res = [y0]
6     t = t0
7     for i in range(int((tn-t0)/h)):
```

```
8         k1 = f(t, res[-1])
9         k2 = f(t + h / 2, res[-1] + h * k1 / 2)
10        k3 = f(t + h / 2, res[-1] + h * k2 / 2)
11        k4 = f(t + h, res[-1] + h * k3)
12        y_next = res[-1] + h * (k1 + 2 * k2 + 2 * k3 + k4)/6
13        res.append(y_next)
14        t += h
15    return res
16
17    def f(x, y):
18        return x*y**2+2*y
19
20    def error(f, y0, t0, tn, h):
21
22        def y_exact(t):
23            return 20*np.exp(2*x)/(np.exp(2*x) - 10*x*np.exp(2*x) - 9)
24
25        y_num = Runge_Kutta4(f, t0, y0, tn, h)
26        t = np.arange(t0, tn + h, h)
27        e = y_num - y_exact(t)
28
29        return t, e
30
31    y0 = -5
32    t0 = 0
33    tn = 5
34    h = 0.1
35
36    result = Runge_Kutta4(f, t0, y0, tn, h)
```

```

37 x = np.linspace(0, 5, len(result))
38 plt.plot(x, result, '-o',label='Numerical Solution')
39
40 t, e = error(f, y0, t0, tn, h)
41
42 plt.plot(t, e, '-o', label='Error')
43
44 plt.xlabel('t')
45 plt.title('Numerical Solution and Error')
46 plt.legend()
47 plt.show()

```

Listing 7: 4th-order Runge-Kutta

5 处理三种方法的误差-比较

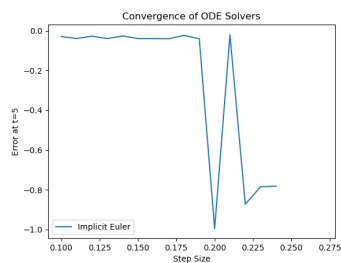


图 4: 隐式欧拉法的误差

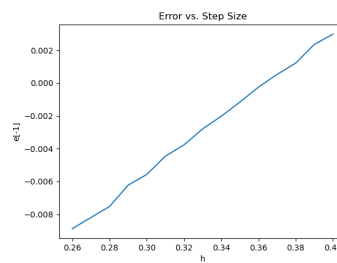


图 5: 显式欧拉法的误差

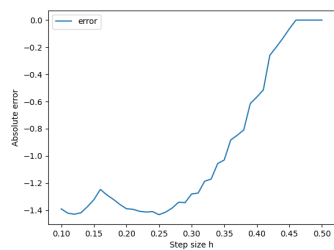


图 6: 4th-order Runge-Kutta的误差

5.1 误差分析

我将数据处理的档案加到附件里输出结果为：

EE.txt: 最接近 0 的值是 $[0.26, -0.008873641307191593]$ ，其距离为

0.2688736413071916。

EIM.txt: 最接近 0 的值是 $[0.1, -0.03759539961851743]$ ，其距离为

0.13759539961851744。

runge.txt: 最接近 0 的值是 $[0.44999999999999984, -0.06431906956574363]$ ，其距离为

0.5143190695657435。

比较上面三张图片及处理完的数据可以看出隐式欧拉法(EIM)在步长的选取上可以选择的精度最小，说明在这三种方法中最为精确。同时，我们也可以通过这个图形来选择一个合适的步长，使得数值解的绝对误差达到一个满意的精度水平。

5.2 收敛性分析

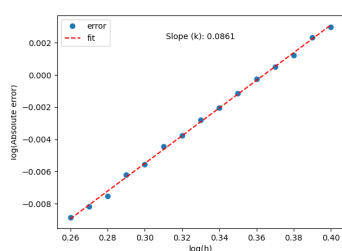


图 7: 显式欧拉法

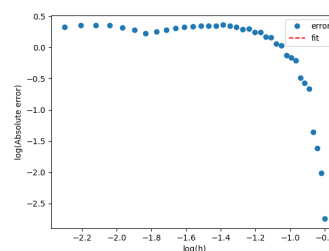


图 8: Runge

收敛应该指的是随着步长 h 的减小，数值解 y_n 逐渐接近精确解 $y(x_n)$ ，即 $\lim_{h \rightarrow 0} y(x_n) = y(x_n)$ 。

为了判断一个数值方法是否收敛，可以考虑不同步长 h 下的数值解和精确解之间的误差，并观察误差随着步长 h 的变化情况。

如果误差随着 h 的减小而减小，那么这个数值方法就是收敛的。

图形反映了步长(h)和数值解的误差之间的关系。横轴表示步长(h)，纵轴表示数值解的绝对误差。通过这个图形，我们可以了解到，当步长减小时，数值解的绝对误差会逐渐减小。

拟合直线的斜率可以表示误差随步长 h 的变化率，因此可以通过拟合直线来估计误差的收敛阶。其绝对值越小，误差随步长 h 的变化越慢，收敛阶就越高。拟合直线截距的意义则是误差的常数项。如果误差本身存在常数项，那么通过拟合直线的截距就可以得到误差的常数项大小。也就是误差与步长的关系，从而评估数值方法的收敛速度和精度。通过拟合出的直线，我们可以得到收敛阶的估计，也就是该数值方法在每次步长减小一半时误差的减小速率。一般来说，收敛阶越高，数值方法的收敛速度越快，精度越高。