三次样条函数插值实验报告

统计1801 孔畅 2020年3月23日

问题

根据不同的边界条件，编写不同的三次样条函数，并评估最终结果。

输入

坐标$x = [x_0, \dots, x_n], y = [y_0, \dots, y_n]$

输出

$I(x) = \{S_i(x), \quad i = 1, \dots, n\}$的系数

数学理论和算法

定义

三次样条函数插值，即找到满足下列条件的分段多项式：

1、在每个分段区间都是三次多项式

2、$S(x_i) = y_i$

3、在整个区间上原函数、一阶导数、二阶导数都是连续的

所以我们有$4n$个未知量

求解

$S_i(x_i) = y_i$

$S_i(x_{i-1}) = y_{i-1}$

$S_i'(x_i) = S_{i-1}'(x_i)$

$S_i''(x_i) = S_{i-1}''(x_i)$

通过以上方程，可以得到$4n-2$个方程组成的方程组

为了求解，我们还需要两个额外条件

在本次实验中，它们分别为：

1、$S_1''(x_0) = 0, S_n''(x_n) = 0$

2、$S_1''(x_0) = a, S_n''(x_n) = b$

3、$S_1'(x_0)=a, S_n'(x_n)=b$

4、$S_1''(x_0)=S_1''(x_1), S_n''(x_n)=S_n''(x_{n-1})$

5、$S_1'''(x_1)=S_2'''(x_1), S_{n-1}'''(x_{n-1})=S_n'''(x_{n-1})$

然后通过解方程求得最终结果

程序

```
import math
import numpy as np
import matplotlib.pyplot  as plt
from sympy import *
from pylab import mpl

def func(y):
    y = np.float64(y)
    return 1/(1 + y * y)

def draw_pic1(x, y):
    fig=plt.figure()
    plt.plot(x, y, label='插值函数')
    plt.plot(x, func(x), label='原函数')
    plt.legend()
    plt.show()

def draw_pic2(x, y):
    fig=plt.figure()
    plt.plot(x, np.fabs(y-func(x)), label='误差')
    plt.legend()
    plt.show()

def spline3_Parameters(x_vec):
        parameter = []
        size_of_Interval = len(x_vec) - 1;
        i = 1

        while i < len(x_vec) - 1:
            data = np.zeros(size_of_Interval * 4)
            data[(i - 1) * 4] = x_vec[i] * x_vec[i] * x_vec[i]
            data[(i - 1) * 4 + 1] = x_vec[i] * x_vec[i]
            data[(i - 1) * 4 + 2] = x_vec[i]
            data[(i - 1) * 4 + 3] = 1
            data1 = np.zeros(size_of_Interval * 4)
            data1[i * 4] = x_vec[i] * x_vec[i] * x_vec[i]
            data1[i * 4 + 1] = x_vec[i] * x_vec[i]
            data1[i * 4 + 2] = x_vec[i]
            data1[i * 4 + 3] = 1

            parameter.append(data)
            parameter.append(data1)
            i += 1

        data = np.zeros(size_of_Interval * 4)
        data[0] = x_vec[0] * x_vec[0] * x_vec[0]
```

```
        data[1] = x_vec[0] * x_vec[0]
        data[2] = x_vec[0]
        data[3] = 1
        parameter.append(data)

        data = np.zeros(size_of_Interval * 4)
        data[(size_of_Interval - 1) * 4] = x_vec[-1] * x_vec[-1] * x_vec[-1]
        data[(size_of_Interval - 1) * 4 + 1] = x_vec[-1] * x_vec[-1]
        data[(size_of_Interval - 1) * 4 + 2] = x_vec[-1]
        data[(size_of_Interval - 1) * 4 + 3] = 1
        parameter.append(data)

        i = 1
        while i < size_of_Interval:
            data = np.zeros(size_of_Interval * 4)
            data[(i - 1) * 4] = 3 * x_vec[i] * x_vec[i]
            data[(i - 1) * 4 + 1] = 2 * x_vec[i]
            data[(i - 1) * 4 + 2] = 1
            data[i * 4] = -3 * x_vec[i] * x_vec[i]
            data[i * 4 + 1] = -2 * x_vec[i]
            data[i * 4 + 2] = -1
            parameter.append(data)
            i += 1

        i = 1
        while i < len(x_vec) - 1:
            data = np.zeros(size_of_Interval * 4)
            data[(i - 1) * 4] = 6 * x_vec[i]
            data[(i - 1) * 4 + 1] = 2
            data[i * 4] = -6 * x_vec[i]
            data[i * 4 + 1] = -2
            parameter.append(data)
            i += 1
        #the other two equations
        data = np.zeros(size_of_Interval * 4)
        data[0] = 6 * x_vec[0]
        data[1] = 2
        parameter.append(data)
        data = np.zeros(size_of_Interval * 4)
        data[-4] = 6 * x_vec[-1]
        data[-3] = 2
        parameter.append(data)
        return parameter

def solution_of_equation(parametes, x):
        size_of_Interval = len(x) - 1;
        result = np.zeros(size_of_Interval * 4)
        i = 1
        while i < size_of_Interval:
            result[(i - 1) * 2] = func(x[i])
            result[(i - 1) * 2 + 1] = func(x[i])
            i += 1
        result[(size_of_Interval - 1) * 2] = func(x[0])
        result[(size_of_Interval - 1) * 2 + 1] = func(x[-1])
        result[-2] = 0
        result[-1] = 0
        a = np.array(spline3_Parameters(x))
        b = np.array(result)
```

```
        #print(b)
        return np.linalg.solve(a, b)

def calculate(paremeters, x):
        result = []
        for data_x in x:
            result.append(
                paremeters[0] * data_x * data_x * data_x + paremeters[1] *
data_x * data_x + paremeters[2] * data_x +
                paremeters[3])
        return result

x_init4 = np.arange(0, 1.01, 0.05)
result = solution_of_equation(spline3_Parameters(x_init4), x_init4)
#print(spline3_Parameters(x_init4))
#print(result)
x_axis4 = []
y_axis4 = []
for i in range(20):
    temp = np.arange(i/20, 0.05 + i/20, 0.01)
    x_axis4 = np.append(x_axis4, temp)
    y_axis4 = np.append(y_axis4, calculate(
        [result[4 * i], result[1 + 4 * i], result[2 + 4 * i], result[3 + 4 *
i]], temp))
draw_pic1(x_axis4, y_axis4)
draw_pic2(x_axis4, y_axis4)
print(np.fabs([result[4 * 0] * 0.03**3 + result[1 + 4 * 0] * 0.03**2 +
               result[2 + 4 * 0] * 0.03 + result[3 + 4 * 0]] - func(0.03)))
print(np.fabs([result[4 * 19] * 0.97**3 + result[1 + 4 * 19] * 0.97**2 +
               result[2 + 4 * 19] * 0.97 + result[3 + 4 * 19]] - func(0.97)))
```

以上是情况一的代码，通过改变最后两个方程和result的值，可以计算各种不同的情况。

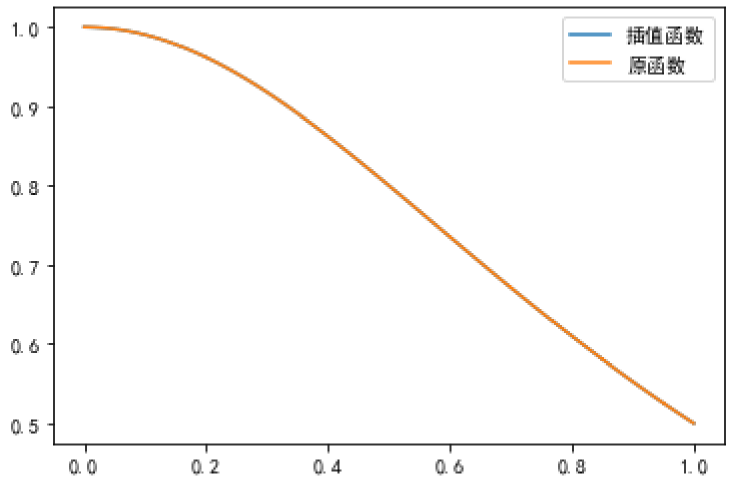
通过被注释掉的$print(result)$可以输出函数的每段系数，最后输出了$x = 0.03, x = 0.97$的误差。

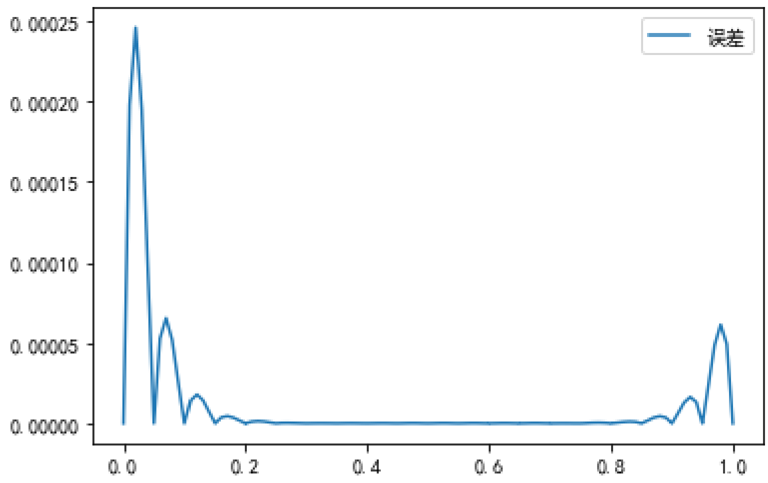
结果

情况一

直接运行上述代码，可得

```
'''
Out:
[0.00019511]
[4.86739892e-05]
'''
```

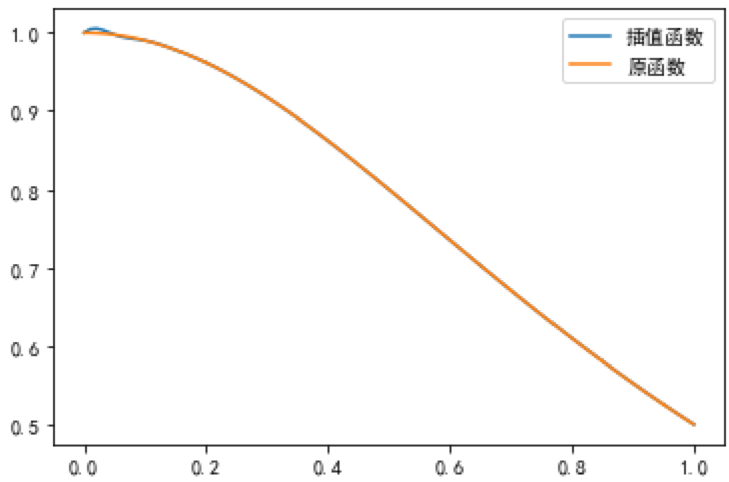


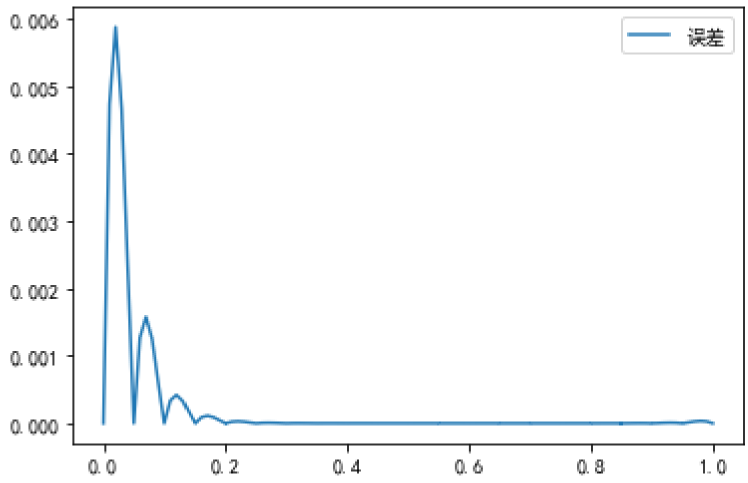


情况二

将result值，即二阶导数边界值设定为原函数的二阶导数，可得

```
result[-2] = (-50)
result[-1] = (-50*26+5000)/26**3
'''
Out:
[0.0046613]
[2.82271253e-05]
'''
```



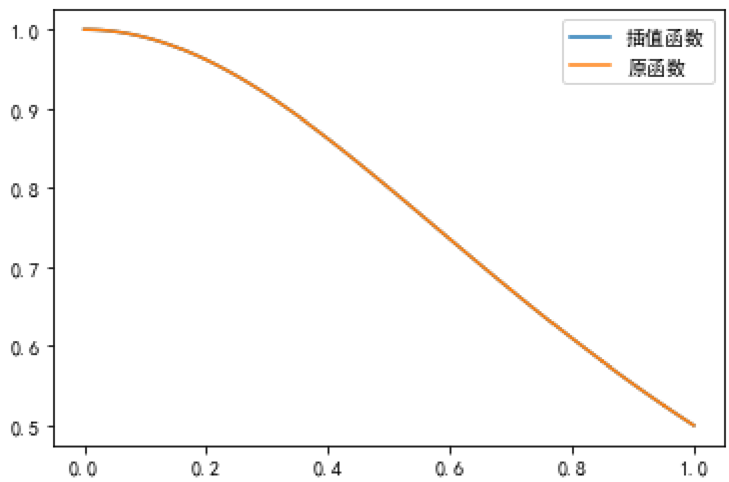


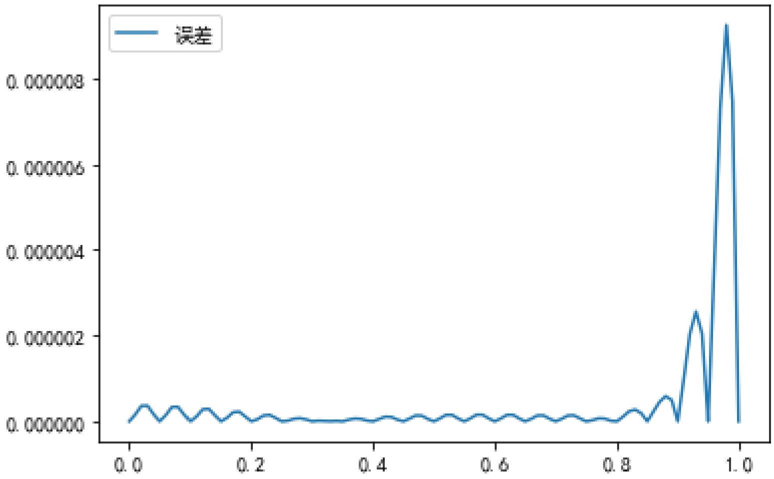
情况三

修改最后两个方程为一阶导数，以及相应的result值

```
#the other two equations
data = np.zeros(size_of_Interval * 4)
data[0] = 3 * x_vec[0] * x_vec[0]
data[1] = 2 * x_vec[0]
data[2] = 1
parameter.append(data)
data = np.zeros(size_of_Interval * 4)
data[-4] = 3 * x_vec[0] * x_vec[0]
data[-3] = 2 * x_vec[0]
data[-2] = 1
parameter.append(data)

result[-2] = 0
result[-1] = (-50)/26**2
'''
Out:
[3.64854933e-07]
[7.3229683e-06]
'''
```

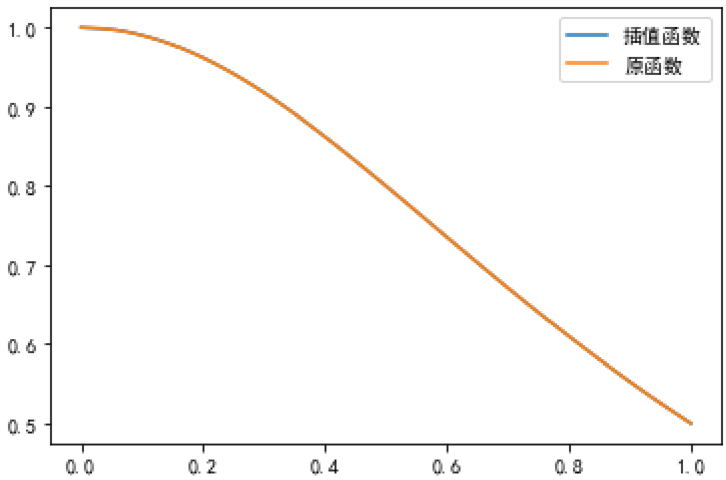


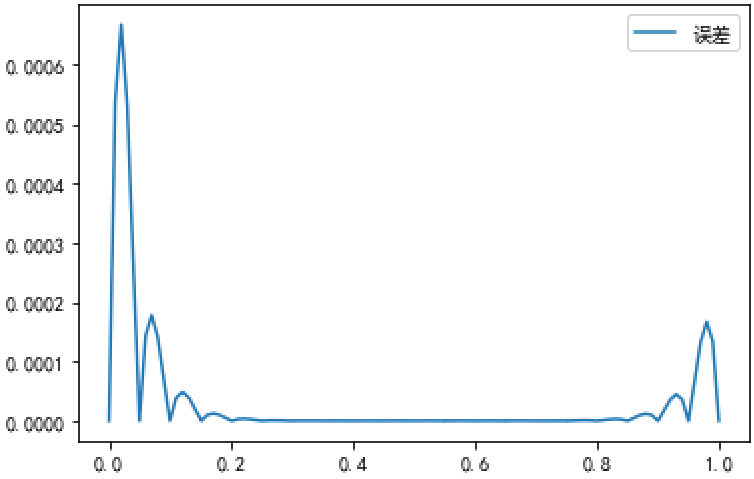


情况四

修改最后两个方程，并将result设为0

```
#the other two equations
data = np.zeros(size_of_Interval * 4)
data[0] = 6 * x_vec[0]
data[1] = 2
data[4] = 6 * x_vec[1]
data[5] = 2
parameter.append(data)
data = np.zeros(size_of_Interval * 4)
data[-8] = 6 * x_vec[-2]
data[-7] = 2
data[-4] = 6 * x_vec[-1]
data[-3] = 2
parameter.append(data)
'''
Out:
[0.00052843]
[0.00013238]
'''
```

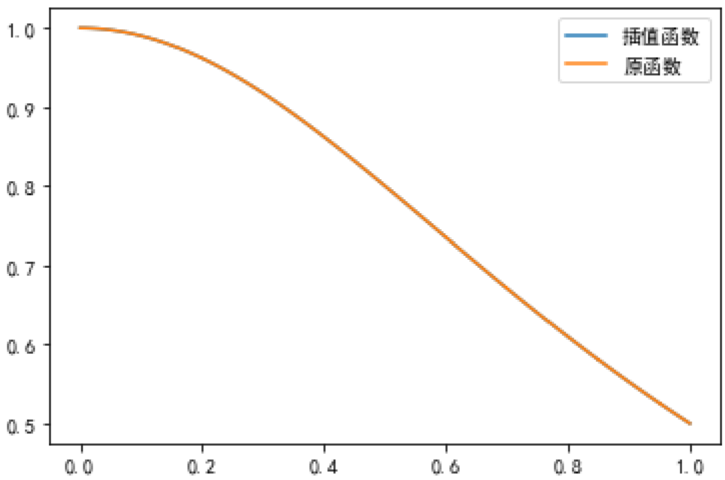


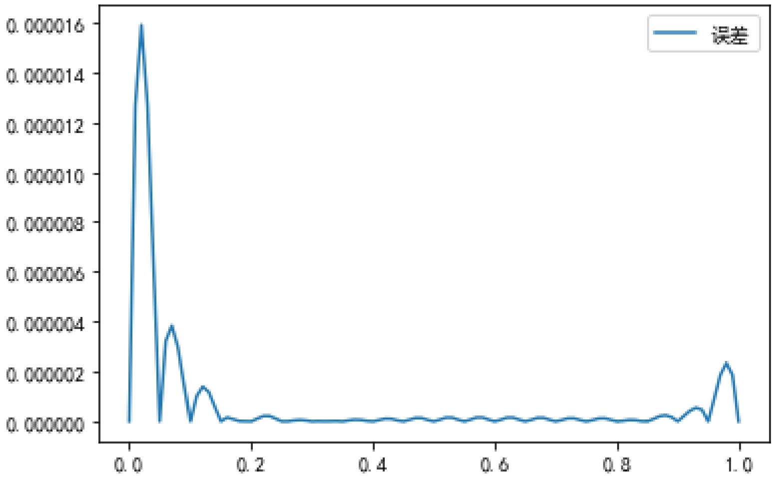


情况五

修改最后两个方程，并将result设为0

```
#the other two equations
data = np.zeros(size_of_Interval * 4)
data[0] = 6
data[4] = 6
parameter.append(data)
data = np.zeros(size_of_Interval * 4)
data[-8] = 6
data[-4] = 6
parameter.append(data)
'''
Out:
[1.26888786e-05]
[1.86244019e-06]
'''
```





结论

从测试的两个值$x = 0.03, x = 0.97$表现的误差来看，表现最好的是第三种插值函数，最差的是第二种插值函数。

由于原函数的二阶导数比较大，因而情况二中强行将函数的边界导数设置成和原函数相同，导致了较大的误差，使得表现十分糟糕。同样的还有强行令边界两个二阶导数值相同的情况四，误差也比较大。

因而在进行三次样条函数插值时，需要考虑到原函数、一阶导数、二阶导数的具体情况，最好不要和三次多项式差距过大，选取较为合理的边界条件。