计算方法(A)

实验（一） 实现三次样条插值

姓　　名	熊恪峥
学　　号	22920202204622
日　　期	2022年3月11日
学　　院	信息学院
课程名称	计算方法(A)

实验（一） 实现三次样条插值

目录

1 项目结构

本程序(**以下称为pyCSI**)的目录结构如表1。pyCSI使用Python完成，只使用了最低限度的、必要的第三方库来进行运算的优化和符号计算的处理。该项目是高度结构化的。因为各种约束的三弯矩方程的系数矩阵有很多相似之处，并且不同约束条件的三次样条插值求解有很多的相同部分，因此pyCSI把这些部分抽象成单独的函数，并且放在不同的文件中，提高了代码的可读性，最后提供一个统一的接口$pyCSI.spline$，方便在其他需要执行三次样条插值的代码中直接调用。

表格 1: 代码文件结构

路径	功能
main.py	调用$pyCSI.spline$进行测试和可视化
pyCSI/spline.py	提供本程序实现的对外接口$pyCSI.spline$
pyCSI/impl/derivative1.py, pyCSI/impl/derivative2.py, pyCSI/impl/notaknot.py, pyCSI/impl/periodic.py	分别实现了一阶导数约束、二阶导数约束、非扭结(Not-A-Knot)约束、周期函数约束的三次样条插值
pyCSI/impl/preprocess.py	实现了对三弯矩方程矩阵构建的预处理，因为各种约束条件的三弯矩矩阵都有较为相似的部分
pyCSI/impl/thomas.py	实现了Thomas Algorithm(追赶法)解三对角矩阵的线性方程组
pyCSI/impl/thomas.py	实现了一些帮助函数，用来构建输出的系数矩阵或者SymPy分段符号函数，以及通过对角线向量构建三对角矩阵

pyCSI主要依赖两个第三方库：

- **NumPy** Python的科学计算库，提供了对向量和矩阵高效的计算操作
- **SymPy** Python的符号计算库，提供了符号计算的能力，主要用于输出符号函数和绘图

2 项目实现

pyCSI为三次样条插值提供了统一的接口，有如下参数

```
def spline(x: Iterable, y: Iterable, constraint_type: ConstraintType = ConstraintType.NOT_A_KNOT,
symbolic_result:  bool = True, **constraints):
```

- **x,y** 插值点的x，y值
- **constriant_type** 要使用的约束类型，支持一阶导数、二阶导数、自然约束(二阶导数为0)、非扭结约束、周期函数约束
- **symbolic_result** 为$true$返回一个SymPy符号函数，是插值得到的分段函数，否则返回三次样条插值函数中每一项的系数组成的矩阵
- **constraints** 额外给定的约束（如果需要）。当使用一阶导数约束时需要给定$m0$和mn，使用二阶导约束时需要给定$M0$和Mn，否则会抛出运行时异常

具体实现以一阶导数约束的实现为例，如代码2，首先预处理出三对角矩阵的三条对角线，以NumPy向量的形式，然后根据给定的参数填入一阶导约束独有的部分，最后构建三弯矩方程并使用追赶法求解，最后计算插值函数每一个括号前的系数。该结果返回给$pyCSI.spline$后它会决定直接返回还是进一步构建一

个SymPy符号函数。具体的预处理和求解都是按照公式实现的，**具体实现请参见相关代码文件**。附录：关键部分代码给出了追赶法解方程的代码实现，如代码5

代码 1: 一阶导数约束的实现

```python
def spline_impl_derive1(x: np.ndarray, y: np.ndarray, h: np.ndarray, m0: np. float, mn: np. float):
        N: Final = x.shape[0] - 1

        alpha, beta, c, dy = preprocess_args(y, h, N)

        # by constraint
        alpha[N] = 1
        beta[0] = 1
        c[0] = (6.0 / h[0]) * (dy[0] / h[0] - m0)
        c[N] = (6.0 / h[N - 1]) * (dy[N - 1] / h[N - 1] + mn)

        M: Final = thomas_solve(alpha[1:N + 1], 2 * np.ones(N + 1), beta[0:N], c)

        return calculate_coefficients(M, y, h, N)
```

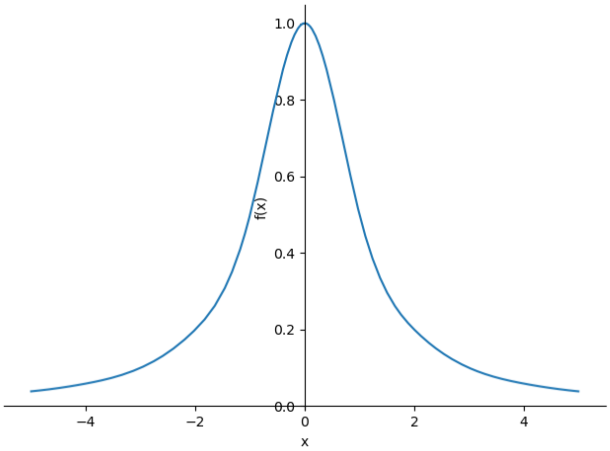
以函数$y = \frac{1}{1+x^2}$ $x \in [-5, 5]$为例，使用非扭结约束插值，输出SymPy符号函数，输出的函数如下，是已经打开括号并化简过的表达式。如果选择输出系数矩阵，输出的并不是该分段函数每一项的系数，而是按照书上给出的三次样条函数标准形式每一个括号前的系数。

代码 2: 输出的结果

```
Piecewise((0.00646792653713069*x**2 + 0.0785733297844025*x + 0.269630023955284, (x >= -5) & (x < -4)), (0.00787862656374767*x**3 + 0.101011445302103*x**2 + 0.456747404844291*x + 0.773862124035135, (x >= -4) & (x < -3)), (0.00649454351876496*x**3 + 0.0885546978972584*x**2 + 0.419377162629758*x + 0.736491881820602, (x >= -3) & (x < -2)), (0.107319669949428*x**3 + 0.693505456481235*x**2 + 1.62927867979771*x + 1.5430928932659, (x >= -2) & (x < -1)), (-0.435773223316476*x**3 - 0.935773223316476*x**2 + 1, (x >= -1) & (x < 0)), (0.435773223316476*x**3 - 0.935773223316476*x**2 + 1.11022302462516e-16*x + 1, (x >= 0) & (x < 1)), (-0.107319669949428*x**3 + 0.693505456481235*x**2 - 1.62927867979771*x + 1.5430928932659, (x >= 1) & (x < 2)), (-0.00649454351876498*x**3 + 0.0885546978972585*x**2 - 0.419377162629758*x + 0.736491881820602, (x >= 2) & (x < 3)), (-0.00787862656374766*x**3 + 0.101011445302103*x**2 - 0.45674740484429*x + 0.773862124035134, (x >= 3) & (x < 4)), (0.0064679265371307*x**2 - 0.0785733297844025*x + 0.269630023955284, (x >= 4) & (x < 5)))
```

对它进行作图，得到图1，可以看出插值效果是正确的。

图 1: 插值结果



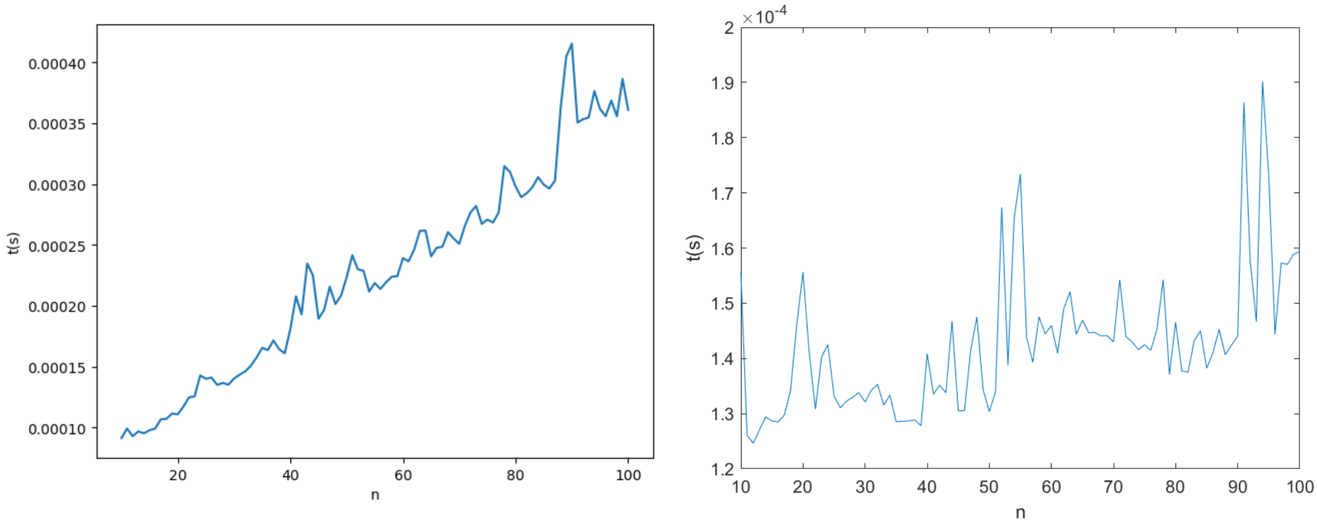
其他插值约束的实现见相关文件，插值效果都经过测试，是正确的。需要注意到的是建立符号函数时展开、化简多项式需要的时间较多，因此返回系数矩阵能大大加快运行速度。

3 基准测试

本程序(**以下称为pyCSI**)在性能方面表现较好。以Matlab R2021b为基线进行测试[1]，在点数不多的情况下，该程序性能有显著优势。运行时长随点数增加呈线性增长，增长情况比较稳定，如图2。测试代码见附录：测试代码，测试数据基本情况如下

- 测试函数： $y = \frac{1}{1+x^2} \quad x \in [-5, 5]$
- 点数范围：等距取10到100个点
- 测量方法：计算一百次插值计时间平均数，排除系统波动影响
- Matlab使用interp1函数，pyCSI使用输出各项系数的参数调用$pyCSI.spline$

图 2: 单次调用平均时间t随插值点数n变化曲线图



(a) pyCSI (b) Matlab

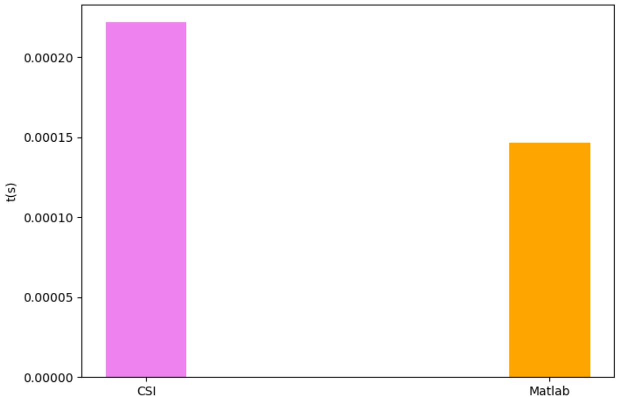
计算图2中测试运行时间的平均值，如图3。本程序平均比Matlab慢仅51.39%，考虑到在规模较低的范围内($n \leq 40$)性能稳定地具有优势，可以认为该实现已经有优秀的运行效率了。

进一步分析运行效率问题，通过cProfile对以上测试代码进行Profiling结果摘录前十位如表2，可以发现通过追赶法解三弯矩方程($thomas\_solve$)是一个性能热点，而构建三弯矩方程的矩阵($preprocess\_args$)是另一个性能热点。其他的均是Python和相关第三方库的内部函数。因此可以得出以下的结论：

- Python和第三方库的内部函数对性能有较为显著的影响
- $thomas_solve$和$preprocess_args$成为性能热点是正常的，一定程度上难以继续优化，因为计算原理决定了它们需要直接在Python代码中使用循环实现，而不能使用NumPy的运算符进行表示，这意味着更多计算发生在Python代码中，而不是NumPy的实现中。后者使用C++完成，并且能够提供大量硬件相关的并行优化的支持。

[1]测试环境： i7-10800H, RTX3060, 32GB运行内存的便携式计算机，运行Windows 11

图 3: 平均运行时间



表格 2: Profiling结果按时长排序前十位

Name	Call Count	Time(ms)	Own Time(ms)
thomas_solve	9100	977	953
<built-in method _imp.create_dynamic>	153	922	921
preprocess_args	9100	824	659
<built-in method io.open_code>	1247	269	269
cleandoc	32052	406	251
<built-in method nt.stat>	5424	178	178
<built-in method marshal.loads>	1247	156	156
<method 'read' of '_io.BufferedReader' objects>	1247	135	135
diff	27300	136	120
calculate_coefficients	9100	139	98

此外，可以发现pyCSI的插值用时随问题规模线性增长，而Matlab在一定范围内波动，增长较为缓慢。我推测Matlab采取的算法中某些插值规模产生的大量计算恰好和CPU的向量化寄存器大小有整数倍关系，能够使用CPU的向量化指令进行并行优化，因此对于特定规模的插值问题具有良好的性能表现。

对于较大规模的问题，我认为可能可以采用GPU来解最终的三弯矩线性方程组、进行对结果的系数等大量数据的运算。但对于一般规模，考虑到将数据传输到GPU的通信成本等开销，我认为实现GPU运算的优化并不划算。所以我没有进行相应的实现。

A 附录：测试代码

该部分给出了用于基准测试的测试代码，包括Python的和Matlab的

代码 3: Python性能测试用代码

```
import numpy as np
import sympy as sp

import matplotlib.pyplot as plt
import seaborn as sns

import timeit

import pyCSI.spline as sl


def test_func(x):
return 1 / (1 + x ** 2)

counts = [i for i in  range(10, 101)]
times = []

for i in counts:
x = np.linspace(start=-5, stop=5, num=i)
y = np.ones(x.shape[0]) / (np.ones(x.shape[0]) + x ** 2)

start = timeit.default_timer()
for j in  range(0, 100):
sl.spline(x, y, sl.ConstraintType.NOT_A_KNOT, False)
end = timeit.default_timer()

times.append((end - start) / 100.0)

average =  sum(times) /  len(times)
sns.lineplot(x=counts, y=times)
plt.xlabel("n")
plt.ylabel("t(s)")
plt.show()

plt.clf()

data = [average, 1.466272222222222e-04]
labels = [’pyCSI’, ’Matlab’]

plt.bar( range( len(data)), data, tick_label=labels, width=0.2, color=["violet", "orange"])
plt.ylabel("t(s)")
plt.show()

print(((average - 1.466272222222222e-04) / 1.466272222222222e-04)*100)
```

代码 4: Matlab性能测试用代码

```
clear;
counts=10:1:100;
times=[];
total_time=0;
for count=counts
x=linspace(-5,5,count);
y=1./(1+x.^2);

tic
for i=0:1:100
pp=interp1(x,y,'spline','pp') ;
end

time=toc/100;
times=[times time];
total_time=total_time + time;

clear i
clear x
clear y
clear pp
end

plot(counts,times)
xlabel("n")
ylabel("t(s)")

total_time/90
```

B 附录：关键部分代码

这里给出追赶法解三对角系数矩阵的线性方程组的代码，以及预处理系数矩阵的代码。其他代码按照项目结构中的表1参见相关的代码文件

代码 5: 追赶法解方程

```
        def thomas_solve(a, b, c, d) -> np.ndarray:
        """
        Use Thomas's algorithm to solve a tri-diagonal system
        :param a: lower diagonal
        :param b: main diagonal
        :param c: upper diagonal
        :param d: the result vector on the right of the equal sign
        :return: return the solution
        """

        n =  len(d)
        w = np.zeros(n - 1,  float)
        g = np.zeros(n,  float)
        p = np.zeros(n,  float)

        w[0] = c[0] / b[0]
        g[0] = d[0] / b[0]

        for i in  range(1, n - 1):
        w[i] = c[i] / (b[i] - a[i - 1] * w[i - 1])
        for i in  range(1, n):
        g[i] = (d[i] - a[i - 1] * g[i - 1]) / (b[i] - a[i - 1] * w[i - 1])

        p[n - 1] = g[n - 1]
        for i in  range(n - 1, 0, -1):
        p[i - 1] = g[i - 1] - w[i - 1] * p[i]
        return p
```

代码 6: 预处理系数矩阵

```
def preprocess_args(y: np.ndarray, h: np.ndarray, N: Final):
alpha: np.ndarray = np.zeros(N + 1)
        c: np.ndarray = np.zeros(N + 1)
        dy = np.diff(y)

        ddyh = np.diff(dy / h)

        # by definition
        for j in  range(1, N):
                alpha[j] = h[j - 1] / (h[j - 1] + h[j])
                c[j] = 6 * (1 / (h[j - 1] + h[j])) * (ddyh[j - 1])

        beta = np.ones(N + 1) - alpha

        return alpha, beta, c, dy
```