# 《数值分析》课程实验报告

实验名称_____实验三 插值与拟合

| 班级 | | 姓名 | | 学号 | | 序号 | |
|---|---|---|---|---|---|---|---|
| 教师 | | 地点 | 数学实验中心 | | | 评分 | |

一. 实验目的

二、实验过程和结果

三、思考题分析解答

四、重点难点分析

# 一.　实验目的

① 掌握多项式插值法的基本思路和步骤；
② 了解整体插值的局限性及分段插值的基本思想。
③ 掌握最小二乘法拟合的基本原理和方法；
④ 培养运用计算机模拟解决问题的能力。

# 二、实验过程和结果

## 1. 拉格朗日插值[1]

Lagrange 插值的算法是：已知有给定的 k + 1个取值点 $(x_0, y_0), \ldots, (x_k, y_k)$，则其 Lagrange 插值多项式为：

$$L(x) = \sum_{j=0}^{k} y_j \ell_j(x)$$

其中的 $\ell_j(x)$ 为插值基函数：

$$\ell_j(x) = \prod_{i=0, i\neq j}^{k} \frac{x - x_i}{x_j - x_i} = \frac{(x - x_0)}{(x_j - x_0)} \cdots \frac{(x - x_{j-1})}{(x_j - x_{j-1})} \frac{(x - x_{j+1})}{(x_j - x_{j+1})} \cdots \frac{(x - x_k)}{(x_j - x_k)}$$

由于插值的目的是得到一个多项式函数，为了方便表达，这里使用 Python 语言并引入 SymPy 库来编程实现。

```
from sympy import *
```

```
lagrange_interpolate(points: list, simplify_result=True,
verbose=False)
```

函数输入：

- points: list, [(x1, y1), (x2, y2), ..., (xn, yn)]
- simplify_result: bool, 化简最终结果, default True
- verbose: bool, 输出每一步的结果, default False

[1] 拉格朗日插值的完整实现见 https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex3/src/lagrange_interpolate.py

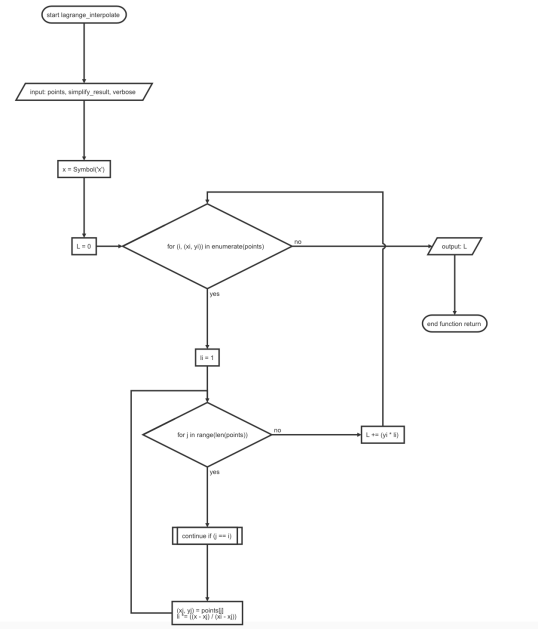
输出：

- L: sympy object of Symbol('x'), 插值多项式 $L(x)$。

该算法的关键代码实现如下：

```
L = 0  # 插值多项式
for i, (xi, yi) in points:
    li = 1
    for j in range(len(points)):
        if j == i:
            continue
        xj, yj = points[j]
        li *= (x - xj) / (xi - xj)
    L += yi * li
```

其中的 x 是来自 SymPy 的符号 x（x=Symbol('x')），这样得到的 L 就是一个符号表示的多项式函数。




【图1】Lagrange 插值程序流程图

例如：

可以看到，这里直接得到了一个完整的多项式函数，十分方便。但使用 SymPy

```
[6]: points = [(11, 0.190809), (12, 0.207912), (13, 0.224951)]

     L = lagrange_interpolate(points)
     print('Hypothesis:  ', L.subs(Symbol('x'), 11.5))
     print('Actual vaule:', sin(rad(11.5)).evalf())
     print('插值多项式:'); L

     Hypothesis:   0.199368500000004
     Actual vaule: 0.199367934417197
     插值多项式:
[6]: −3.20000000000042 · 10^−5 x^2 + 0.0178390000000004x − 0.00154799999999966
```

的一个弊端是计算速度比较慢，如果需要对大型问题进行计算，并且只需得到部分点的预测值，将代码中的 x 的定义改为一个数或者类似 NumPy 数组的对象即可快速得到结果。

## 2. 牛顿插值[2]

另一种常用的多项式插值方式是 Newton 多项式插值：对于给定的 k+1 个数据点 $(x_0, y_0), \ldots, (x_k, y_k)$，则插值多项式为 $N(x) = \sum_{j=0}^{k} a_j n_j(x)$.

[2] 牛顿插值的实现见：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex3/src/newton_interpolate.py

其中的 $n_j(x)$ 为牛顿插值基函数：$n_j(x) = \prod_{i=0}^{j-1}(x - x_i)$，参数 $a_j$ 为差商 $a_j = [y_0, \ldots, y_j]$.

所以想要实现 Newton 插值，首先需要实现差商的计算：

**差商表（高阶差商是两个低一阶差商的差商）**

| | 0阶差商 | 1阶差商 | 2阶差商 | 3阶差商 | ... | $k-1$阶差商 |
|---|---|---|---|---|---|---|
| $x_0$ | $f[x_0]$ | | | | | |
| $x_1$ | $f[x_1]$ | $f[x_0, x_1]$ | | | | |
| $x_2$ | $f[x_2]$ | $f[x_1, x_2]$ | $f[x_0, x_1, x_2]$ | | | |
| $x_3$ | $f[x_3]$ | $f[x_2, x_3]$ | $f[x_1, x_2, x_3]$ | $f[x_0, x_1, x_2, x_3]$ | | |
| ... | ... | ... | ... | ... | ... | |
| $x_k$ | $f[x_k]$ | $f[x_{k-1}, x_k]$ | $f[x_{k-2}, x_{k-1}, x_k]$ | $f[x_{k-3}, x_{k-2}, x_{k-1}, x_k]$ | ... | $f[x_0, \ldots, x_k]$ |

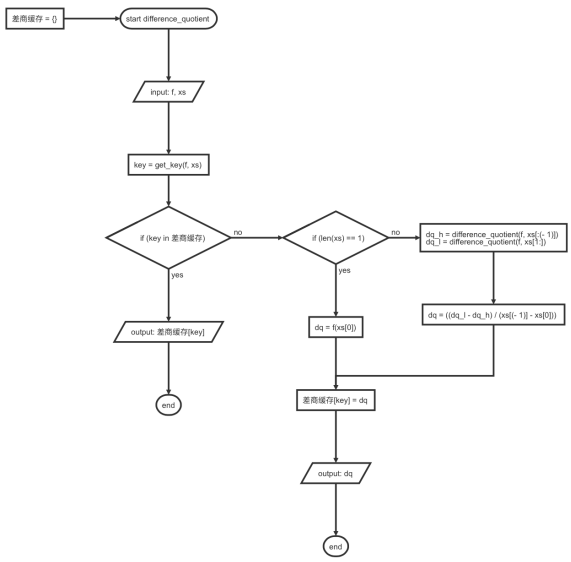
【图2】差商表 (from wikipedia)

这个计算容易用递归算法实现。设计如下递归函数：

```
difference_quotient[3](f, xs: list, verbose=False)
```

输入参数：一元函数 `f`；要计算的差商的 $f[x0,x1,...]$ 的参数 `xs=[x0, x1, ...]`。返回输出差商值 `dq`。

具体实现中需要使用的一个重要技巧是差商缓存。考虑 $f[x_1, x_2, x_3]$ 和 $f[x_2, x_3, x_4]$ 的计算，二者都需要使用到 $f[x_2, x_3]$ 的值，这就出现了重复计算。当需要计算的差商阶数增加时，这种重复计算量会急剧增长，严重影响算法效率。




【图3】差商算法程序流程图

所以有必要设计一个缓存机制，通过一个 key-value 结构把之前求过的差商值记录下来，下次再需要这个值时，直接从缓存中取用，不必去重复计算。

---

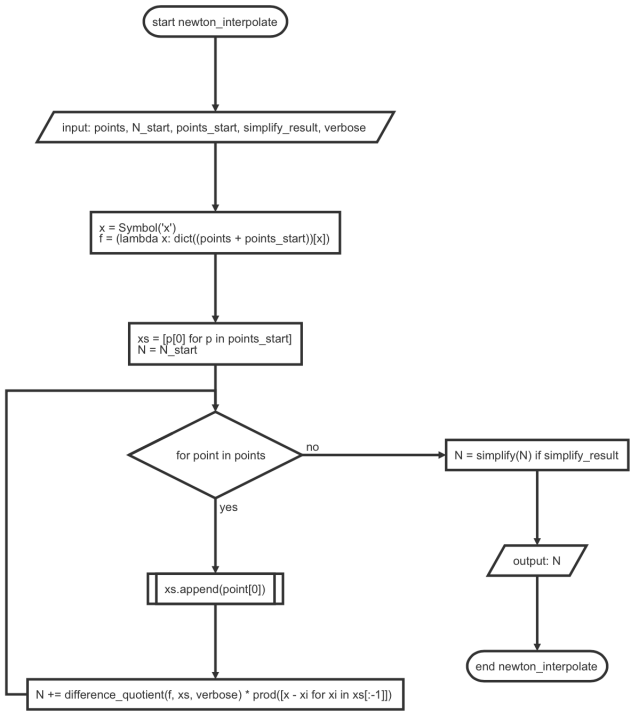
[3] 差商的计算代码：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex3/src/difference_quotient.py

有了差商的实现，牛顿插值的实现就比较容易了：

```
newton_interpolate(points, N_start=0, points_start=[],
simplify_result=True, verbose=False)
```

输入：

- `points: [(x1, y1), (x2, y2), ..., (xn, yn)]`：插值点
- `N_start: a sympy object of Symbol('x')`, 起始插值多项式。default N_start=0 (从头开始构建)。
- `points_start: [(x1, y1), (x2, y2), ..., (xn, yn)]`: 计算 N_start 用的插值点。default points_start=[] (从头开始构建)。
- `simplify_result: bool`：化简最终结果, default True
- `verbose`: 打印出每一步计算差商的值, default False




【图4】牛顿插值程序流程图

输出：

- `N`: 一个关于 Symbol('x') 的插值多项式 $N(x)$.

该函数被设计成可以利用牛顿插值的承袭性的。给定插值点 points 调用该函数做的插值是承袭自 points_start 和 N_start 的。也就是是说，如果之前已经做过一次差值（这里 N_start 和 points_start 取默认值 0 与 [] 即不承袭，从头开始做插值）：

```
N0 = newton_interpolate(points0)
```

就可以在其基础上，新增一些插值点 points1 去改进插值结果：

```
N1 = newton_interpolate(points1, N_start= N0,
points_start=points0)
```

这时，通过之前介绍的差商缓存机制，只需计算新增点补充的差商，而避免了大量重复之前的工作。

下面这个调用实例中演示了承袭性的利用：

```
points = [(11, 0.190809), (12, 0.207912), (13, 0.224951)]

N = newton_interpolate(points)
print('插值多项式:', N)
print('Hypothesis:  ', N.subs(Symbol('x'), 11.5))
print('Actual vaule:', sin(rad(11.5)).evalf())

# 承袭之前的 N:
print('承袭:')
new_points = [(11.2, sin(rad(11.2)).evalf()), (11.7, sin(rad(11.7)).evalf())]
N1 = newton_interpolate(new_points, N_start=N, points_start=points, verbose=True)
print('插值多项式:', N1)
print('Hypothesis:  ', N1.subs(Symbol('x'), 11.5).evalf())
print('Actual vaule:', sin(rad(11.5)).evalf())
```

```
插值多项式：-3.20000000000042e-5*x**2 + 0.0178390000000001*x - 0.00154800000000063
Hypothesis:   0.199368500000000
Actual vaule: 0.199367934417197
承袭：
cached: f[11, 12, 13]: -0.0000320000000000042
cached: f[12, 13]: 0.0170390000000000
cached: f[13]: 0.224951000000000
f[11.2]: 0.194234351219972
f[13, 11.2]: 0.0170648048777933
f[12, 13, 11.2]: -0.0000322560972416778
f[11, 12, 13, 11.2]: -0.00000128048620836791
cached: f[11, 12, 13, 11.2]: -0.00000128048620836791
cached: f[12, 13, 11.2]: -0.0000322560972416778
cached: f[13, 11.2]: 0.0170648048777933
cached: f[11.2]: 0.194234351219972
f[11.7]: 0.202787295356512
f[11.2, 11.7]: 0.0171058882730810
f[13, 11.2, 11.7]: -0.0000316026117597499
f[12, 13, 11.2, 11.7]: -0.00000217828493975971
f[11, 12, 13, 11.2, 11.7]: -0.00000128256961627400
插值多项式：-1.282569616274e-6*x**4 + 5.9256799679765e-5*x**3 - 0.00105582207039453
*x**2 + 0.0256792199573976*x - 0.0240006476355346
Hypothesis:   0.199367875528590
Actual vaule: 0.199367934417197
```

【图5】牛顿插值程序的调用实例

从输出可以看到，承袭时使用上一次计算记录下的差商缓存避免了许多重复的计算，实现了插值多项式的快速改进。

## 3. 三次样条插值——三弯矩法[4]

使用多项式插值的一个问题是，如果我们希望提高精度就需要提高插值多项式的次数，而高次多项式插值会存在一些问题，例如 Runge 现象。所以在实际使用中会考虑使用分段的低次插值。

这里选择使用三弯矩法实现一个分段三次样条插值。

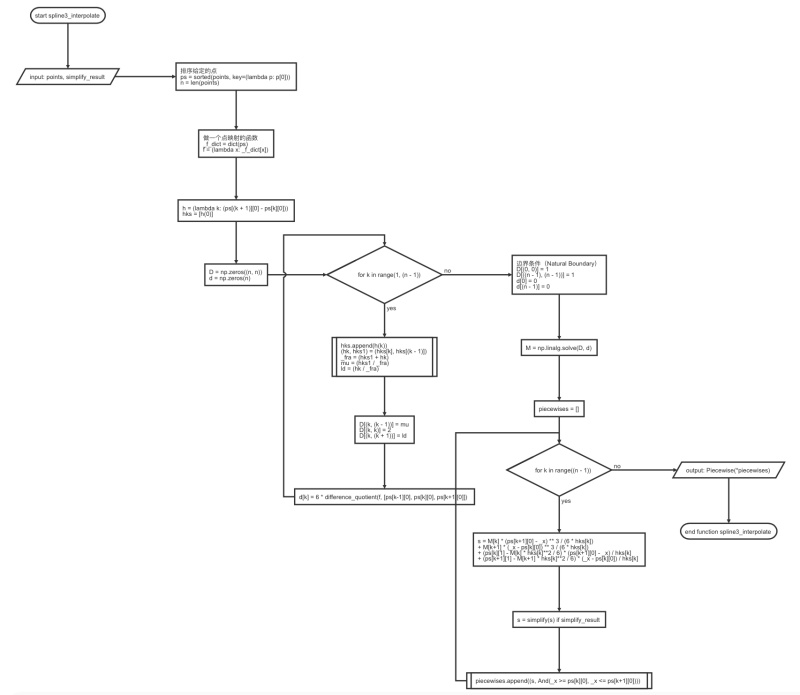
[4]三弯矩法实现代码见：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex3/src/spline3_interpolate.py

三弯矩法实现函数：

```
spline3_interpolate(points: list, simplify_result=True)
```

输入参数：

- `points: list, [(x1, y1), (x2, y2), ..., (xn, yn)]`
- `simplify_result: bool, 化简最终结果, default True`




【图6】三弯矩法实现程序流程图

返回输出：

- S: 一个关于 Symbol('x') 的 SymPy 分段函数（Piecewise）对象：插值函数 $S(x)$.

为了通用性考虑，也为了和之前的 Lagrange 插值、Newton 插值实现保持一致，该实现中没有传入被插值的原函数，只使用一些离散点进行插值，所以无法利用原函数的导数实现更优的边界。这里参考 Wikipedia: Spline (mathematics)，采用了避免使用依赖原函数的自然边界（natural cubic spline）。

调用测试：

[13]:
```
points = [(11, 0.190809), (12, 0.207912), (13, 0.224951), (11.2, sin(rad(11.2)).evalf()), (11.7, sin(rad(11.7)).evalf())]

S = spline3_interpolate(points)
print('Hypothesis:  ', L.subs(Symbol('x'), 11.5))
print('Actual vaule:', sin(rad(11.5)).evalf())
print('插值函数:'); S
```

```
Hypothesis:   0.199368500000004
Actual vaule: 0.199367934417197
插值函数:
```

[13]:

$$\begin{cases} 0.0171291662851547x - 6.02546323704911 \cdot 10^{-5}(x-11)^3 + 0.00238817086329846 & \text{for } x \geq 11 \wedge x \leq 11.2 \\ 8.11573408739488 \cdot 10^{-6}x^3 - 0.000308841444758762x^2 + 0.0209858710400982x - 0.0134683556585243 & \text{for } x \geq 11.2 \wedge x \leq 11.7 \\ -2.58576144575224 \cdot 10^{-5}x^3 + 0.00088623089167835x^2 + 0.00703403599315696x + 0.0409438010245449 & \text{for } x \geq 11.7 \wedge x \leq 12 \\ 0.0170232496562323x + 1.57503437676575 \cdot 10^{-5}(x-13)^3 + 0.00364875446897983 & \text{for } x \geq 12 \wedge x \leq 13 \end{cases}$$

## 4. 插值实验题目[5]

1. 给定 $\sin 11^o = 0.190809, \sin 12^o = 0.207912, \sin 13^o = 0.224951$, 构造插值多项式计算 $\sin 11^o 30'$。

(1) 编程实现拉格朗日插值，并计算结果。

(2) 将计算结果和查表结果进行比较。

2. 区间 $[-5,5]$ 作等距划分：$x_k = -1 + kh \quad (k = 0,1,\cdots,n), \quad h = \frac{10}{n}$，以 $x_k$（$k = 0,1,\cdots,n$）为节点对函数 $f(x) = \frac{1}{1+x^2}$ 进行插值逼近。（分别取 $n = 5,10,20$）

(1) 用多项式插值对 $f(x)$ 进行逼近，并在同一坐标系下作出函数的图形，进行比较。写出插值函数对 $f(x)$ 的逼近程度与节点个数的关系，并分析原因。

(2) 试用分段插值（任意选取）对 $f(x)$ 进行逼近，在同一坐标下画出图形，观察分段插值函数对 $f(x)$ 的逼近程度与节点个数的关系。

解：

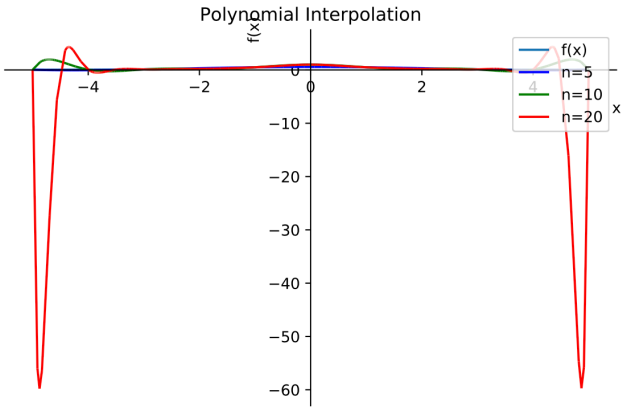
首先实现目标函数 $f$ 以及一个获取插值点的辅助函数 `get_points`：

```python
def f(x):
    return 1 / (1 + x ** 2)

def get_points(s, n, f):
    """在闭区间 s 作 n 等距划分，输出 n+1 个插值点 [(x0, f(x0)), (x1, f(x1)), ..., (xn, f(xn))]
    e.g.
        get_points([-5, 5], 2, lambda x: 1 / (1 + x ** 2))
    output: [(-5.0, 0.038461538461538464), (0.0, 1.0), (5.0, 0.038461538461538464)]
    """
    h = (s[1] - s[0]) / n
    xs = [s[0] + k * h for k in range(n+1)]
    points = [(x, f(x)) for x in xs]
    return points
```

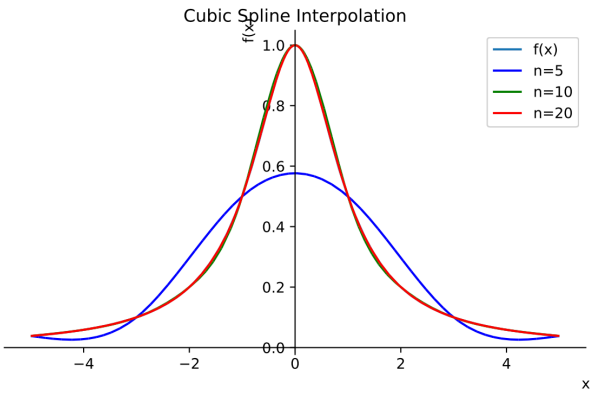
然后调用之前的插值实现函数获取 5、10、20 个插值点的插值结果。这里选用牛顿法做多项式插值，三弯矩法做三次样条插值。得到的插值函数太多，并且过于复杂，不方便展示，请参考本页脚注5到实现的源文件中查看。

[5] 推荐到下面的连接中查看题目解答过程的 Jupyter Notebook： https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex3/src/ex3.ipynb

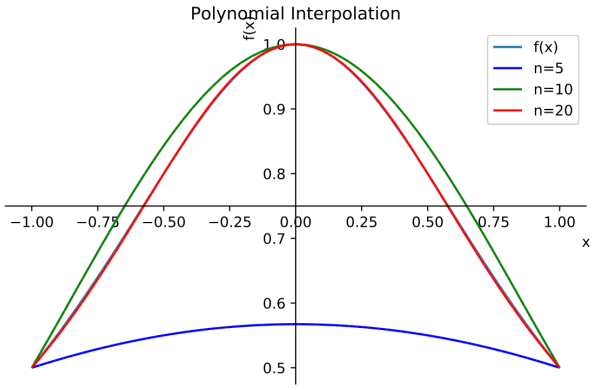
下面是一些插值结果的图像。图7、8、9为多项式插值；图10、11、12是三次样条插值。从中可以看到随着插值点的增多，插值效果的改进，以及高次多项式插值的 Runge 现象。



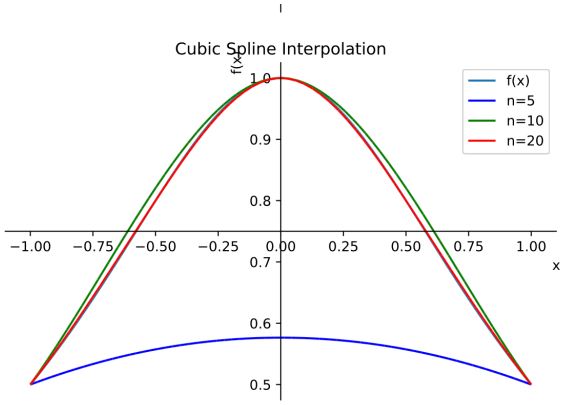

【图7】



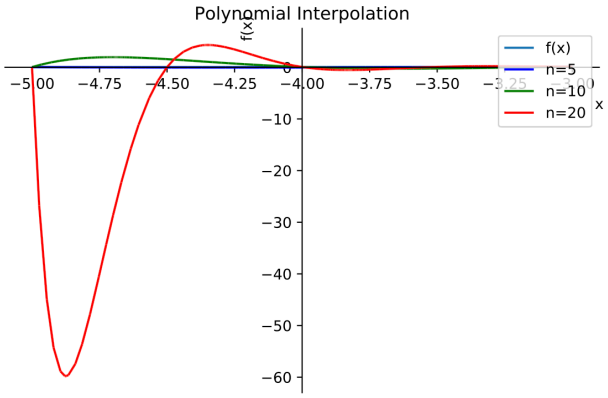

【图10】



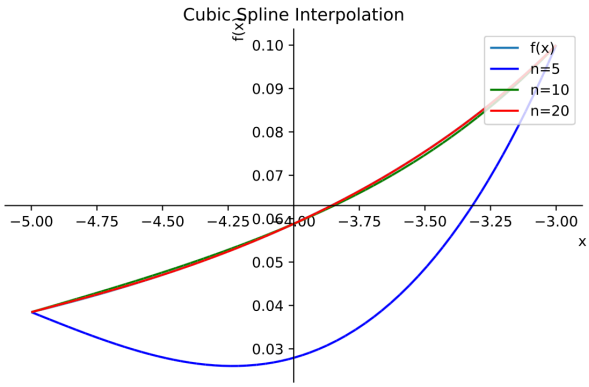

【图8】




【图11】




【图9】




【图12】

## 5. 数据的最小二乘拟合

通过最小二乘法，可以导出向量化表示的正规方程：

$$\theta = (X^T X)^{-1} X^T y$$

其中：

$$X = \begin{bmatrix} -(x^{(1)})^T - \\ -(x^{(2)})^T - \\ \vdots \\ -(x^{(m)})^T - \end{bmatrix} \qquad y = [y^{(1)}, y^{(2)}, \ldots, y^{(m)}]^T$$

$x^{(i)}$ 为一组数据，比如 $(1, x_1, x_1^2)$，求得的 $\theta$ 即拟合曲线 $h = \theta^T x$ 的参数。

关于此方法更多详细的说明可以参考本人的这篇文章：Machine Learning: Normal Equation。

将上述计算公式封装成函数就实现了最小二乘法拟合：

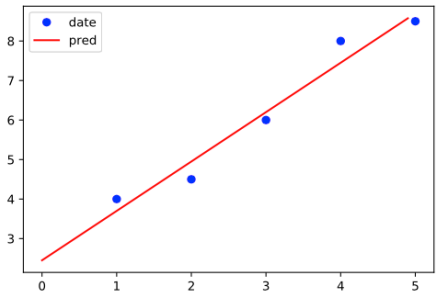
`normalEquationFit`[6]`(X, y)`

使用这个方法的关键是如何构造拟合数据矩阵 $X$。对于线性拟合（单变量线性回归），$X$ 就是一列全 1 的向量和数据向量 $x$ 的拼接。例如下面是实验内容 3.4.2 的题目1：

```
x = np.array([1, 2, 3, 4, 5])
y = np.array([4, 4.5, 6, 8, 8.5])

X = np.c_[np.ones((len(x), 1)), x]

theta = normalEquationFit(X, y)
print(theta)

pred_f = lambda t: theta[0] + theta[1] * t
```

```
[2.45 1.25]
```

拟合结果：




【图13】线性拟合结果

[6] 此方法的实现源码：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex3/src/normal_eqn.py

题目 2 解答[7]：

```
x = np.array([2, 3, 4, 7, 8, 10, 11, 14, 16, 18, 19])
y = np.array([106.42, 108.2, 109.5, 110, 109.93, 110.49, 110.59, 110.6, 110.76, 111, 111.2])
```

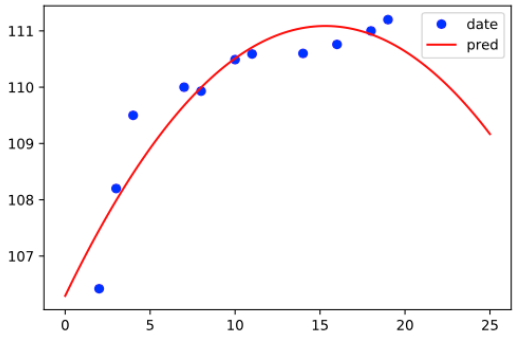
- 使用多项式拟合：$y(x) = c_0 + c_1 x + c_2 x^2$：

按照【图14】构建数据矩阵 $X$，然后代入拟合，得到结果如【图15】所示。

```
# X: [1, x, x**2]
X = np.c_[np.ones((len(x), 1)), x, x ** 2]
print(X)
```

```
[[  1.   2.   4.]
 [  1.   3.   9.]
 [  1.   4.  16.]
 [  1.   7.  49.]
 [  1.   8.  64.]
 [  1.  10. 100.]
 [  1.  11. 121.]
 [  1.  14. 196.]
 [  1.  16. 256.]
 [  1.  18. 324.]
 [  1.  19. 361.]]
```

【图14】多项式拟合的X构建




【图15】多项式拟合结果

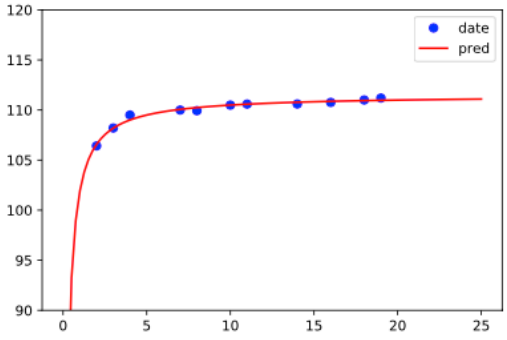
- 使用非线性的 $y(x) = a e^{\frac{b}{x}}$ 拟合：

两边取对数：$\ln(y) = \ln(a) + \frac{b}{x}$，就可以利用最小二乘拟合得到参数 ln(a) 和 b。

```
# X: [1, 1/x]
X = np.c_[np.ones((len(x), 1)), 1 / x]
print(X)
```

```
[[1.         0.5       ]
 [1.         0.33333333]
 [1.         0.25      ]
 [1.         0.14285714]
 [1.         0.125     ]
 [1.         0.1       ]
 [1.         0.09090909]
 [1.         0.07142857]
 [1.         0.0625    ]
 [1.         0.05555556]
 [1.         0.05263158]]
```

【图16】指数型数据X的构造




【图17】指数型拟合结果

[7] 实验的内容在 Jupyter Notebook 中查看会比较方便：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex3/src/ex3.ipynb

## 三、思考题分析解答

1、整体插值有何局限性？如何避免？

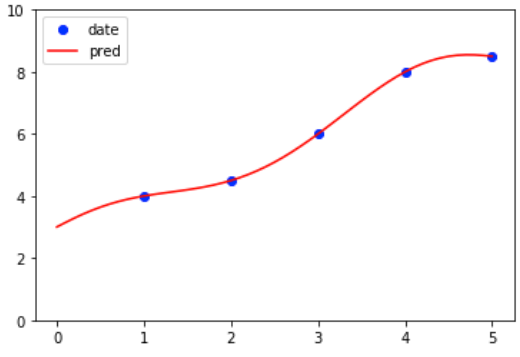
如上文的描述中所提及，使用整体的多项式插值时，如果我们希望提高精度就需要提高插值多项式的次数，而高次多项式插值在局部上会存在一些问题，例如【图7】、【图9】所示的 Runge 现象。

为了避免这种局限性，在实际使用中会考虑使用如上文中实现的三次样条插值的分段低次插值法。

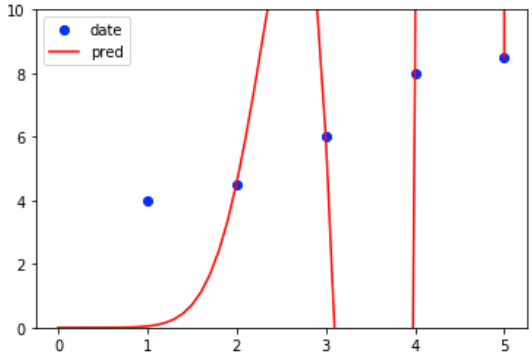
2、基函数的选择对拟合的结果有何影响?

从实验内容中可以看到，不同的基函数对于不同的问题表现会有很大不同，需要根据数据的分布情况选择合适的函数。题目2 使用的两种函数的比较就验证了这一点。

同时，使用多项式拟合时，次数的选择也是一个关键的因素。如 题目1 的数据明显成比较线性的关系，所以用一个 1次的线性基函数的拟合就可以得到比较好的结果，而如果我们对其使用相当高次的多项式回归就会得到严重过拟合的结果，这显然不是我们希望的:




【图18】对题目1使用7次多项式拟合




【图19】对题目1使用14次多项式拟合

3、简述数据拟合与插值的异同。

插值必过数据点，而拟合不然。

拟合达到的效果得到一个总体上最好地表述了离散点分布的曲线，也就是用一个曲线从整体上去大限度地逼近这些点。

而插值更多是已知一个原函数的情况下，用一个比如多项式的插值函数去尽最大可能去穿过原函数曲线上的一些点。

4、试着编程实现三次样条插值

见上文实验内容中的 三次样条插值——三弯矩法。

## 四、重点难点分析

重点：

1. 多项式插值法的基本思路和步骤；
2. 整体插值的局限性及分段插值的基本思想。
3. 最小二乘法拟合的基本原理和方法；

难点：

1. 差商的缓存机制设计
2. 三次样条插值的实现