所有代码及说明均上传至GitHub仓库：https://github.com/DMCXE/Numerical_Computation/tree/master/Courses_202303

1 理论介绍

1.0 插值法

我们为什么要**插值**？在一个核裂变反应堆内，为了精细化建模我们需要通过试验确定不同核素和不同能量中子发生相互反应时的反应截面。最好的情况是，有一套完整的作用理论，我们只需要测几个普遍数据作为验证即可。但我们不仅尚未构建出截面模型，甚至依托实验手段无法精准的得到不同能量得到中子束以测定不同能量下的反应截面。更致命的是，由于**多普勒（Doppler）效应**，目标核素自身热运动大小改变了中子的相对能量，使得截面产生多**普勒展宽（Doppler-Broadening）**。运用插值法，我们希望解决这样的问题：

- 通过有限的能量-截面数据推出任意能量下的截面数据。
- 确保推导产生的数值是精确的，符合客观规律的。
- 产生如多普勒展宽效益下，局部插值同样能发生改变，且保持其余区域的稳定性

- 快速计算，节省计算资源，提高分析效率

我们希望根据给定的函数表做一个既能反应截面函数 $f(x)$ 的特性，又便于计算的简单函数 $P(x)$ ，用 $P(x)$ 近似 $f(x)$ 。而这便同样是插值法的目的。

1.1 插值方法的定义

设函数 $y=f(x)$ 在区间 $[a,b]$ 上有定义，且已知 $a \leqslant x_0<x_1<\cdots<x_n \leqslant b$ 上的值 $y_0, y_1, \ldots, y_n$ ，若存在一简单函数 $P(x)$ ，使：

$P(x_i)=y_i, i=0,1, \ldots, n$

成立，就称 $P(x)$ 为 $f(x)$ 的**插值函数**，点 $x_0, x_1, \cdots, x_n$ 称为**插值节点**，包含插值节点的区间 $[a,b]$ 称为插值区间，求插值产生 $P(x)$ 的方法称为**插值法**。

依据插值函数的不同，分为：

插值多项式： $P(x)$ 是次数不超过n的代数多项式

分段插值： $P(x)$ 为分段多项式

三角插值： $P(x)$ 为三角多项式

1.2 多项式插值

定理：设在区间 $[a,b]$ 上给定的 $n+1$ 个点 $a \leqslant x_0<x_1<\cdots<x_n \leqslant b$ 上的函数值
$y_i=f(x_i)(i=0,1, \cdots, n)$ ，求次数不超过n次的多项式，使得

$$P(x_i)=y_i, i=0,1, \cdots, n$$

满足此条件的多项式是**唯一**的。

1.2.1拉格朗日（Lagrange）插值多项式

1.2.1.1 数学定义

定义：若n次多项式 $l_j(x)(j=0,1, \cdots, n)$ 在 $n+1$ 个节点 $x_0<x_1<\cdots<x_n$ 上满足条件：

$$l_j\left(x_k\right)=\begin{cases}1, & k=j, \\ 0, & k \neq j,\end{cases} \quad j, k=0,1, \cdots, n,$$

就称这 $n+1$ 个 n 次多项式 $l_0(x), l_1(x), \cdots, l_n(x)$ 为节点 $x_0<x_1<\cdots<x_n$ 上的**n次插值基函数**。可以表示为：

$$L_n(x)=\sum_{k=0}^n y_k l_k(x)$$

其中， $l_k(x)$ 表示为：

$$l_k(x)=\frac{(x-x_0) \cdots\left(x-x_{k-1}\right)\left(x-x_{k+1}\right) \cdots\left(x-x_n\right)}{\left(x_k-x_0\right) \cdots\left(x_k-x_{k-1}\right)\left(x_k-x_{k+1}\right) \cdots\left(x_k-x_n\right)}, \quad k=0,1, \cdots, n.$$

满足上式的多项式可称为**拉格朗日(Lagrange)插值多项式**。不失简便性，可写成如下形式：

$$L_n(x) = \sum_{i=1}^{n} y_i \left(\prod_{j \neq i}^{1 \leq j \leq n} \frac{(x - x_j)}{(x_i - x_j)} \right)$$

1.2.1.2 算法实现

注意到拉格朗日多项式的计算公式，包含若干个统一格式的乘积与求和。将会对应至少两个循环的嵌套。当需要插入较多数据点时，势必会大大减慢匀算速度。为此，我们希望能否通过矩阵运算以提高计算速度。

我们注意到，对于 $\prod_{j \neq i}^{1 \leq j \leq n} \frac{(x - x_j)}{(x_i - x_j)}$ ，其分子分母本质上是一致的。不同的是分母的被减数是给定的数据集，而分子的被减数是希望插值得到的数据。因此，分子分母实际对应一一个相同的函数 $def(x)$ ，它们仅仅是自变量 x 存在差别。这一发现有助于提高代码的服用率。

提升运算速度的途径之一是用向量方法代替循环。需要注意的是，这里利用了Numpy先天的矩阵计算加速功能，这一功能在不同语言不通设备上都有对应的高性能矩阵计算方法。对于数据点 $\mathbf{X} = [x_0, x_1, \cdots, x_n]$ 生成 n 阶方阵 $\mathbf{XS}$

$$\mathbf{XS} = \begin{pmatrix} x_0, x_1, \cdots, x_n \\ x_0, x_1, \cdots, x_n \\ \vdots \\ x_0, x_1, \cdots, x_n \end{pmatrix}_{n \times n}$$

为保证条件$j \neq i$，删去对角线元素，令 $n \times n$ 矩阵化为 $n \times (n-1)$

$$\mathbf{XSS} = \begin{pmatrix} x_1, \cdots, x_n \\ x_0, \cdots, x_n \\ \vdots \\ x_0, x_1, \cdots \end{pmatrix}_{n \times n-1}$$

这些构成了分子分母中共同的减数。

函数 $def(x)$ 能够接受两种不同类型的数据，**数与数组**。利用Numpy矩阵乘法的特性，无论是数还是数组，乘上n阶单位矩阵后，都能够获得全同的或行相同的矩阵。在本代码中，**数**对应预计的插值（**分子**部分），数组对应已知的数据点 $\mathbf{X}$ （**分母**部分）。

对于分母：输入数据点 $\mathbf{X}$ ，并将其转置，为保证条件 $j \neq i$ ，删去对角线元素，令 $n \times n$ 矩阵化为 $n \times (n-1)$

$$\mathbf{XP} = \begin{pmatrix} x_0, x_0, \cdots, x_0 \\ x_1, x_1, \cdots, x_1 \\ \vdots \\ x_n, x_n, \cdots, x_n \end{pmatrix}_{n \times n} \rightarrow \mathbf{XPP} = \begin{pmatrix} x_0, \cdots, x_0 \\ x_1 \cdots, x_1 \\ \vdots \\ x_n, x_n, \cdots \end{pmatrix}_{n \times n-1}$$

分母则为 $\mathbf{XPP} - \mathbf{XSS}$ ，并将按**行**将元素相乘，形成 $1 \times n$ 向量。

对于分子：输入代求位置数x，并将其转置，为保证条件 $j \neq i$ ，删去对角线元素，令 $n \times n$ 矩阵化为 $n \times (n-1)$ ：

$$\mathbf{XQ}=\begin{pmatrix}x,x,\cdots,x\\x,x,\cdots,x\\\vdots\\x,x,\cdots,x\end{pmatrix}_{n\times n}\rightarrow\mathbf{XQQ}=\begin{pmatrix}x,\cdots,x\\x,\cdots,x\\\vdots\\x,x,\cdots\end{pmatrix}_{n\times n-1}$$

分母则为 $\mathbf{XQQ}-\mathbf{XSS}$，并将按**行**将元素相乘，形成 $1\times n$ 向量。

分子分母相除，每行依次代表了 $i=1,2,\cdots,n$ 下的商，即为 S。对于数据集 $\mathbf{Y}=[y_0,y_1,\cdots,y_n]$，进行 $Y\cdot S$ 并将全部元素累加，得到对应位置数x的值。前述 $def(x)$ 过程对应代码如下：

```
    def donodo(self,x):
        one = np.ones((self.lenth, self.lenth))
        arro = (x*one).T
        arro_de = arro[~np.eye(self.lenth,dtype=bool)].reshape(self.lenth,-1)
        arro2_de = (self.arr1_x*one)[~np.eye(self.lenth,
dtype=bool)].reshape(self.lenth, -1)
        res = np.prod(arro_de - arro2_de, axis=1, keepdims=False)
        return res
```

对于应位置数x的值：

```
    def num(self,x):
        nom = self.donodo(x)
        return np.sum(self.arr1_y*nom/self.denom)
```

为了便于计算，将以上求解过程打包为类。为了加快对于任意x的计算，在类初始化的时候就对于分母进行计算：

```
class Lagrange:
    def __init__(self,arr1):
        self.arr1 = arr1
        self.arr1_x = arr1[:,0]
        self.denom = self.donodo(self.arr1_x)
```

全部代码如下：全部打代码可前往Github仓库https://github.com/DMCXE/Numerical_Computation 中文件`Lagrange.py`下查看。

```
import numpy as np
import matplotlib.pyplot as plt
class Lagrange:
    def __init__(self,arr1):
        self.arr1 = arr1
        self.arr1_x = arr1[:,0]
        self.arr1_y = arr1[:,1]
        self.lenth = len(arr1)
        self.denom = self.donodo(self.arr1_x)

    def donodo(self,x):
```

```
        one = np.ones((self.lenth, self.lenth))
        arro = (x*one).T
        arro_de = arro[~np.eye(self.lenth,dtype=bool)].reshape(self.lenth,-1)
        arro2_de = (self.arr1_x*one)[~np.eye(self.lenth,
dtype=bool)].reshape(self.lenth, -1)
        res = np.prod(arro_de - arro2_de, axis=1, keepdims=False)
        return res

    def num(self,x):
        nom = self.donodo(x)
        return np.sum(self.arr1_y*nom/self.denom)

    def visualize(self,start,end,step,text):
        x = np.linspace(start,end,step)
        y = np.zeros(1)
        for i in x:
            y = np.append(y,self.num(i))
        y = y[1:]
        plt.figure()
        plt.scatter(self.arr1_x, self.arr1_y, c='red')
        if text is True:
            for j in range(0,self.lenth):
                plt.text(self.arr1_x[j],self.arr1_y[j],(self.arr1_x[j],self.arr1_y[j]))
        plt.plot(x,y)
        plt.show()
```

1.2.2 牛顿插值多项式

1.2.2.1 数学定义

均差定义：一般地，称

$$f\left[x_0,x_1,\cdots,x_k\right]=\frac{f\left[x_0,\cdots,x_{k-2},x_k\right]-f\left[x_0,x_1,\cdots,x_{k-1}\right]}{x_k-x_{k-1}}$$

为 $f(x)$ 的**k阶均差**。一阶均差 $f[x_0,x_k]=\frac{f(x_k)-f(x_0)}{x_k-x_0}$ ，二阶均差 $f[x_0,x_1,x_k]=\frac{f\left[x_0,x_k\right]-f\left[x_0,x_1\right]}{x_k-x_1}$ 。

均差可通过直接列均差表计算：

x_k	$f(x_k)$	一阶均差	二阶均差	三阶均差	四阶均差
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_4	$f(x_4)$	$f[x_3,x_4]$	$f[x_2,x_3,x_4]$	$f[x_1,x_2,x_3,x_4]$	$f[x_0,x_1,x_2,x_3,x_4]$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

牛顿插值多项式定义：对于基函数$\{1, x-x_0, (x-x_0)\cdots(x-x_{n-1})\}$生成多项式 $P_n x$ 表示为：

$$P_n(x) = a_0 + a_1(x-x_0) + \cdots + a_n(x-x_0)\cdots(x-x_{n-1})$$

其中，$a_k = f[x_0, x_1, \cdots, x_k], \quad k = 0, 1, \cdots, n$ 。则称 $P_n(x)$ 为**牛顿插值多项式**。

注意到，系数 a_k 即为上均差表每列的第一项。因此在使用牛顿插值多项式时，对于新添加的点毋需重新计算，只需在上均差表中更新即可。

1.2.2.2 算法实现

由于多项式插值的唯一性，因此牛顿插值本质上与拉格朗日插值一致。算法实现上可以一直。但是为了突出牛顿法的k阶均差特性，依从算法本身描述进行翻译。

依据k阶均差表生成方差的步骤可以化为

```
    def f(self):
        list = [self.arr1_y] #list可以包容不同长度的向量，以区分不同阶
        fx = np.array([self.arr1_y[0]])
        for j in range(0,self.lenth-1):
            list2 = []
            long = len(list[j])
            for i in range(0,long-1):
                l2 = (list[j][i]-list[j][i+1])/(self.arr1_x[i]-self.arr1_x[j+i+1])
                list2.append(l2)
            list.append(list2)
            fx = np.append(fx,list2[0])
        return fx,list
```

调用函数的返回结果list，可以得到上节所提到的均差表。对于得到的系数 $a_k = f[x_0, x_1, \cdots, x_k]$，代入多项式表达式，

$$P_n(x) = a_0 + a_1(x-x_0) + \cdots + a_n(x-x_0)\cdots(x-x_{n-1})$$

以函数值方式给出，代码如下所示：

```
    def num(self,x):
        num = self.arr1_y[0]
        for i in range(1,self.lenth):
            prod = 1
            for j in range(0,i):
                eq = x-self.arr1_x[j]
                prod = prod*eq
            num = num + self.fr[i]*prod
        return num
```

为了便于计算，将以上求解过程打包为类，全部代码如下：全部打代码可前往Github仓库https://github.com/DMCXE/Numerical_Computation 中文件`Newton.py`下查看。

```
import numpy as np
```

```
import matplotlib.pyplot as plt
class Newton:
    def __init__(self,arr1):
        self.arr1 = arr1
        self.arr1_x = arr1[:,0]
        self.arr1_y = arr1[:,1]
        self.lenth = len(arr1)
        self.fr= self.f()[0]

    def f(self):
        list = [self.arr1_y] #list可以包容不同长度的向量，以区分不同阶
        fx = np.array([self.arr1_y[0]])
        for j in range(0,self.lenth-1):
            list2 = []
            long = len(list[j])
            for i in range(0,long-1):
                l2 = (list[j][i]-list[j][i+1])/(self.arr1_x[i]-self.arr1_x[j+i+1])
                list2.append(l2)
            list.append(list2)
            fx = np.append(fx,list2[0])
        return fx,list

    def num(self,x):
        num = self.arr1_y[0]
        for i in range(1,self.lenth):
            prod = 1
            for j in range(0,i):
                eq = x-self.arr1_x[j]
                prod = prod*eq
            num = num + self.fr[i]*prod
        return num

    def visualize(self,start,end,step,text):
        x = np.linspace(start,end,step)
        y = np.zeros(1)
        for i in x:
            y = np.append(y,self.num(i))
        y = y[1:]
        plt.figure()
        plt.scatter(self.arr1_x, self.arr1_y, c='red')
        if text is True:
            for j in range(0,self.lenth):
                plt.text(self.arr1_x[j],self.arr1_y[j],(self.arr1_x[j],self.arr1_y[j]))
        plt.plot(x,y)
        plt.show()
```

1.3 高次插值的病态现象

龙格(Runge)指出，高次多项式不一定能收敛于 $f(x)$ ，其病态性质会导致多项式在插值点间发生大幅度剧烈的变化，强烈的破坏数值的可信度。为此，可以考虑在电间使用分段低次插值。

1.4 分段低次插值

为了避免高次插值出现的病态现象，为此我们可以采用在数据点之间采用多个低次插值并令其相互光滑连接。在这里，我们主要讨论三次自由样条插值与三次Hermit差值。

1.4.1 三次自由样条插值

样条是一根富有弹性的细长的木条，将样条固定在样点上，其它地方自由弯曲并沿着木条画下细线，称为样条曲线。这样的曲线都满足二阶导数连续。因此拓展得到了数学样条概念。

1.4.1.1 数学定义

定义：若函数 $S(x) \in C^2[a,b]$ ，且在每个小区间 $[x_j, x_{j+1}]$ 上是三次多项式，其中 $a = x_0 < x_1 < \cdots < x_n = b$ 是给定节点，则称 $S(x)$ 是节点 $x_0, x_1, \cdots, x_n$ 上的**三次样条函数**。若在节点 x_j 上给定函数值 $y_i = f(x_i)(j = 0, 1, \cdots, n)$ ，并有

$$S(x_j) = y_j, j = 0, 1, \cdots, n$$

则称 $S(x)$ 为**三次样条插值函数**。

三次样条插值需要确定两个**边界条件**才可以确定 $S(x)$ ，常见的边界条件有：

- 已知两端的一阶导数值，即 $S'(x_0) = f_0', \quad S'(x_n) = f_n'$
- 已知两端的二阶导数值，即 $S''(x_0) = f_0'', \quad S''(x_n) = f_n''$
- $S(x)$ 是以 $x_n - x_0$ 为周期的周期函数

对于第二类边界条件， $S''(x_0) = f_0'', \quad S''(x_n) = f_n''$ ，当边界处二阶导数为0时，即 $S''(x_0) = S''(x_n) = 0$ ，称为**自然（由）边界条件**

这里我们主要讨论**自然边界条件**。通过分段定义可以得到三次样条插值的表达式为：

$$\begin{aligned} S(x) =& M_j \frac{(x_{j+1} - x)^3}{6h_j} + M_{j+1} \frac{(x - x_j)^3}{6h_j} + \left(y_j - \frac{M_j h_j^2}{6} \right) \frac{x_{j+1} - x}{h_j} \\ &+ \left(y_{j+1} - \frac{M_{j+1} h_j^2}{6} \right) \frac{x - x_j}{h_j}, \quad j = 0, 1, \cdots, n-1 \end{aligned}$$

由于一阶导数连续，即在分割点处左右导数相等可得：

$$\mu_j M_{j-1} + 2M_j + \lambda_j M_{j+1} = d_j, \quad j = 1, 2, \cdots, n-1$$

其中：

$$\begin{aligned} &\mu_j = \frac{h_{j-1}}{h_{j-1} + h_j}, \quad \lambda_j = \frac{h_j}{h_{j-1} + h_j} \\ &d_j = 6 \frac{f[x_j, x_{j+1}] - f[x_{j-1}, x_j]}{h_{i-1} + h_i} = 6 f[x_{j-1}, x_j, x_{j+1}] \quad j = 1, 2, \cdots, n-1, \end{aligned}$$

可以写成三对角矩阵形式：

$$\begin{pmatrix} 2 & \lambda_0 & & & \\ \mu_1 & 2 & \lambda_1 & & \\ & \ddots & \ddots & \ddots & \\ & & \mu_{n-1} & 2 & \lambda_{n-1} \\ & & & \mu_n & 2 \end{pmatrix} \begin{pmatrix} M_0 \\ M_1 \\ \vdots \\ M_{n-1} \\ M_n \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix}$$

由于 $S''(x_0) = S''(x_n) = 0$，即 $M_0 = M_n = 0, d_0 = d_n = 0$

1.4.1.2 算法实现

依据递推公式 $\mu_j M_{j-1} + 2M_j + \lambda_j M_{j+1} = d_j$ 的各项定义，由初始条件生成各系数：

```
    #hn为x之间的间隔
    def hn(self):
        hnn = np.array([])
        for i in range(0,self.lenth-1):
            hnn =np.append(hnn,self.arr1_x[i+1]-self.arr1_x[i])
        return hnn

    def mu(self):
        mu = np.zeros(1)
        hn = self.hn()
        for i in range(1,len(hn)):
            mu = np.append(mu,hn[i-1]/(hn[i-1]+hn[i]))
        return mu

    def lam(self):
        lam = np.zeros(1)
        hn = self.hn()
        for i in range(1,len(hn)):
            lam = np.append(lam,hn[i]/(hn[i-1]+hn[i]))
        return lam
    #fm为余项，定义与牛顿插值相同
    def fm(self,i):
        return (self.arr1_y[i]-self.arr1_y[i+1])/(self.arr1_x[i]-self.arr1_x[i+1])\
               -(self.arr1_y[i]-self.arr1_y[i-1])/(self.arr1_x[i]-self.arr1_x[i-1])

    def dn(self):
        dn = np.zeros(1)
        hn = self.hn()
        for i in range(1,len(hn)):
            dn = np.append(dn,6*self.fm(i)/(hn[i-1]+hn[i]))
        return dn
```

通过系数即可生成带求的线性方程组矩阵，并调用求解器求解。关于三对角线性方程组的求法，可参加后文线性方程组求解专题。

```
    def Mn(self):
        a = np.append(self.mu(),0)
        c = np.append(self.lam(),0)
        b = 2*np.ones(self.lenth)
        d = np.append(self.dn(),0)
        Mn = self.TDMA(a,b,c,d)
        return Mn
```

将各项代入三次样条插值的表达式，以函数值方式给出，代码如下所示：

```
    def num(self,x):
        j = self.zone(x)#zone函数的作用为确定输入量x处于的区间
        M = self.Mn()
        h = self.hn()
        S = M[j]*((self.arr1_x[j+1]-x)**3)/(6*h[j]) \
            + M[j+1]*((x-self.arr1_x[j])**3)/(6*h[j]) \
            + (self.arr1_y[j]-(M[j]*(h[j]**2))/6)*(self.arr1_x[j+1]-x)/h[j] \
            + (self.arr1_y[j+1]-M[j+1]*h[j]**2/6)*(x-self.arr1_x[j])/h[j]
        return S
```

为了便于计算，将以上求解过程打包为类，全部代码如下：全部打代码可前往Github仓库https://github.com/DMCXE/Numerical_Computation 中文件`CubicSplineFree.py`下查看。

```
import numpy as np
import matplotlib.pyplot as plt
class CubicSplineFree:
    def __init__(self,arr1):
        self.arr1 = arr1
        self.arr1_x = arr1[:,0]
        self.arr1_y = arr1[:,1]
        self.lenth = len(arr1)
    #hn为x之间的间隔
    def hn(self):
        hnn = np.array([])
        for i in range(0,self.lenth-1):
            hnn =np.append(hnn,self.arr1_x[i+1]-self.arr1_x[i])
        return hnn

    def mu(self):
        mu = np.zeros(1)
        hn = self.hn()
        for i in range(1,len(hn)):
            mu = np.append(mu,hn[i-1]/(hn[i-1]+hn[i]))
        return mu

    def lam(self):
        lam = np.zeros(1)
        hn = self.hn()
```

```
        for i in range(1,len(hn)):
            lam = np.append(lam,hn[i]/(hn[i-1]+hn[i]))
        return lam
    #fm为余项，定义与牛顿插值相同
    def fm(self,i):
        return (self.arr1_y[i]-self.arr1_y[i+1])/(self.arr1_x[i]-self.arr1_x[i+1])\
               -(self.arr1_y[i]-self.arr1_y[i-1])/(self.arr1_x[i]-self.arr1_x[i-1])

    def dn(self):
        dn = np.zeros(1)
        hn = self.hn()
        for i in range(1,len(hn)):
            dn = np.append(dn,6*self.fm(i)/(hn[i-1]+hn[i]))
        return dn

    def TDMA(self,a, b, c, d):
        try:
            n = len(d)  #确定长度以生成矩阵
            # 通过输入的三对角向量a,b,c以生成矩阵A
            A = np.array([[0] * n] * n, dtype='float64')
            for i in range(n):
                A[i, i] = b[i]
                if i > 0:
                    A[i, i - 1] = a[i]
                if i < n - 1:
                    A[i, i + 1] = c[i]
            # 初始化代计算矩阵
            c_1 = np.array([0] * n)
            d_1 = np.array([0] * n)
            for i in range(n):
                if not i:
                    c_1[i] = c[i] / b[i]
                    d_1[i] = d[i] / b[i]
                else:
                    c_1[i] = c[i] / (b[i] - c_1[i - 1] * a[i])
                    d_1[i] = (d[i] - d_1[i - 1] * a[i]) / (b[i] - c_1[i - 1] * a[i])
            # x: Ax=d的解
            x = np.array([0] * n)
            for i in range(n - 1, -1, -1):
                if i == n - 1:
                    x[i] = d_1[i]
                else:
                    x[i] = d_1[i] - c_1[i] * x[i + 1]
            #x = np.array([round(_, 4) for _ in x])
            return x
        except Exception as e:
            return e

    def Mn(self):
```

```
        a = np.append(self.mu(),0)
        c = np.append(self.lam(),0)
        b = 2*np.ones(self.lenth)
        d = np.append(self.dn(),0)
        Mn = self.TDMA(a,b,c,d)
        return Mn

    def zone(self,x):
        if x < np.min(self.arr1_x): zone = 0
        if x > np.max(self.arr1_x): zone = self.lenth-2
        for i in range(0,self.lenth-1):
            if x-self.arr1_x[i]>=0 and x-self.arr1_x[i+1]<=0:
                zone = i
        return zone

    def num(self,x):
        j = self.zone(x) #zone函数的作用为确定输入量x处于的区间
        M = self.Mn()
        h = self.hn()
        S = M[j]*((self.arr1_x[j+1]-x)**3)/(6*h[j]) \
            + M[j+1]*((x-self.arr1_x[j])**3)/(6*h[j]) \
            + (self.arr1_y[j]-(M[j]*(h[j]**2))/6)*(self.arr1_x[j+1]-x)/h[j] \
            + (self.arr1_y[j+1]-M[j+1]*h[j]**2/6)*(x-self.arr1_x[j])/h[j]
        return S

    def visualize(self,start,end,step,text):
        x = np.linspace(start,end,step)
        y = np.zeros(1)
        for i in x:
            y = np.append(y,self.num(i))
        y = y[1:]
        plt.figure()
        plt.scatter(self.arr1_x, self.arr1_y, c='red')
        if text is True:
            for j in range(0,self.lenth):
                plt.text(self.arr1_x[j],self.arr1_y[j],(self.arr1_x[j],self.arr1_y[j]))
        plt.plot(x,y)
        plt.show()
```

1.4.2 分段三次Hermit差值

1.4.2.1 数学原理

虽然分段线性函数$I_h(x)$的导数是间断的，但如果能够在节点处提供已知的导数，就可以构造出一阶导数连续的分段插值函数，在每一段上是三次插值函数。即构成了分段三次Hermit差值，这样，很容易可以推导得到$I_h(x)$在区间$[x_k, x_{k+1}]$上的表达式为

$$I_h(x)=\left(\frac{x-x_{k+1}}{x_k-x_{k+1}}\right)^2\left(1+2\frac{x-x_k}{x_{k+1}-x_k}\right)f_k+\left(\frac{x-x_k}{x_{k+1}-x_k}\right)^2\left(1+2\frac{x-x_{k+1}}{x_k-x_{k+1}}\right)f_{k+1}$$
$$+\left(\frac{x-x_{k+1}}{x_k-x_{k+1}}\right)^2(x-x_k)f'_k+\left(\frac{x-x_k}{x_{k+1}-x_k}\right)^2(x-x_{k+1})f'_{k+1}.$$

上述对于$k=0,1,\cdots,n-1$成立

1.4.2.2 算法实现

与1.4.1三次自由样条插值类似的，在插值类`CubicHermit`中，定义了`zone`函数以确定不同x落入的范围，并定义了`num`函数如下所示：

```
def num(self,x):
    j = self.zone(x)
    a1 = (x-self.arr1_x[j+1])/(self.arr1_x[j]-self.arr1_x[j+1])
    a2 = (x-self.arr1_x[j])/(self.arr1_x[j+1]-self.arr1_x[j])
    I = (a1**2)*(1+2*a2)*self.arr1_y[j] + \
        (a2**2)*(1+2*a1)*self.arr1_y[j+1]+ \
        (a1**2)*(x-self.arr1_x[j])*self.df[j] + \
        (a2 ** 2) * (x - self.arr1_x[j+1]) * self.df[j+1]
    return I
```

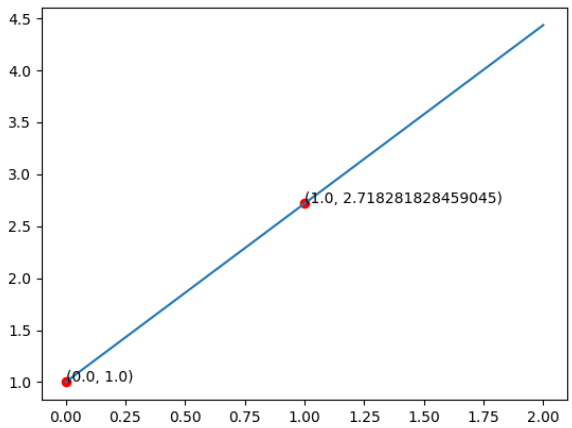
即可快速的实现插值计算，

2 回答题目

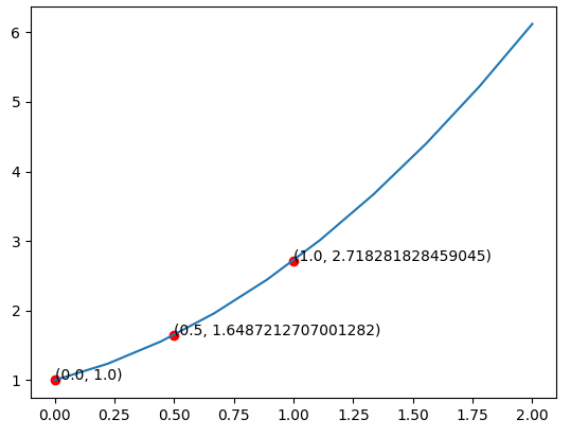
Q1：建立线性插值与二次插值多项式并绘制曲线

答案部分：

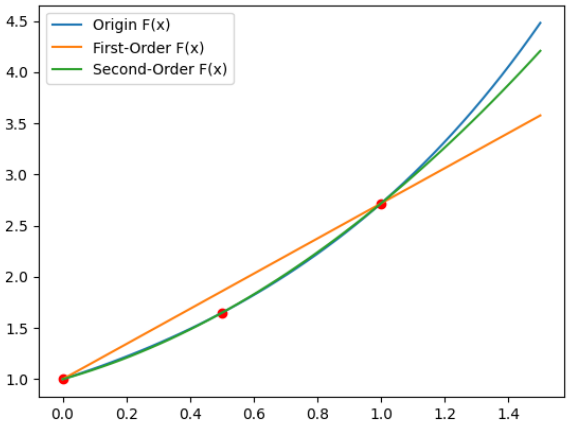
一次插值部分：对于$x=0,1,y=1,e^{-1}$，可以建立线性插值多项式并返回所需要的值。在$x=0.2:0.2:0.8$范围内，计算的值为：**[1.34365637,1.68731273,2.0309691 ,2.37462546]**。函数图像如下图所示：



二次插值部分：对于$x=0,0.5,1, y=1, e^{-0.5}, e^{-1}$，可以建立线性插值多项式并返回所需要的值。在$x=0.2:0.2:0.8$范围内，计算的值为：**[1.20898779 ,1.48530987 ,1.82896624, 2.23995689]**。函数图像如下图所示：



与原函数对比：如下图所示，在二次插值相比于线性插值更接近原函数。



代码实现：

```
import numpy as np
import matplotlib.pyplot as plt
from Lagrange import Lagrange as LA #通过上述自建库调用
"图像比较"
def visualize(start, end, step):
    x = np.linspace(start, end, step)
    y0 = np.zeros(1)
    y1 = np.zeros(1)
    y2 = np.zeros(1)
    for i in x:
        y0 = np.append(y0, OriginF(i))
        y1 = np.append(y1, LA1.num(i))
        y2 = np.append(y2, LA2.num(i))
    y0 = y0[1:]
    y1 = y1[1:]
```

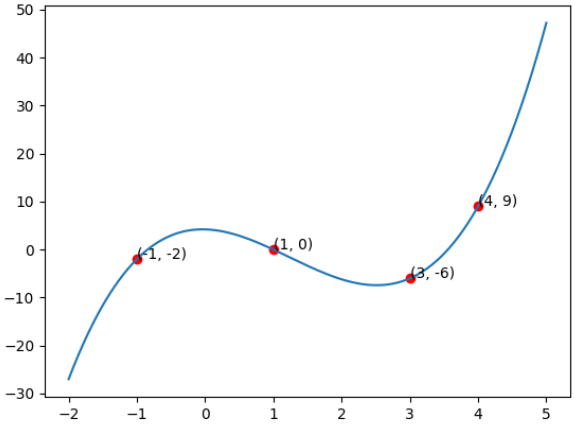
```
    y2 = y2[1:]
    plt.figure()
    plt.scatter(X2,Y2 , c='red')
    plt.plot(x, y0,label = 'Origin F(x)')
    plt.plot(x, y1,label = 'First-Order F(x)')
    plt.plot(x, y2,label = 'Second-Order F(x)')
    plt.legend()
    plt.show()

def OriginF(x):
    return np.exp(x)

"Q1-1：对节点x0 = 0, x1 = 1进行一次差值"
X1 = np.array([0,1])     #构建X1的数组
Y1 = OriginF(X1)         #获得对应实际值的数组
AXY = np.c_[X1,Y1]       #合成N x 2点阵

LA1 = LA(AXY)
Num1 = np.array([])
for x in np.linspace(0.2,0.8,4):
    Num1 = np.append(Num1,LA1.num(x))
print(Num1)
LA1.visualize(0,2,10,True)

"Q1-2：对节点x0 = 0, x1 = 1, x2 = 0.5进行二次差值"
X2 = np.array([0,0.5,1])     #构建X1的数组
Y2 = OriginF(X2)         #获得对应实际值的数组
AXY2 = np.c_[X2,Y2]       #合成N x 2点阵

LA2 = LA(AXY2)
Num2 = np.array([])
for x in np.linspace(0.2,0.8,4):
    Num2 = np.append(Num2,LA2.num(x))
print(Num2)
LA2.visualize(0,2,10,True)

"图像比较"
visualize(0,1.5,50)
```

Q2: 建立牛顿插值并给出近似值

答案部分

从插值函数形式上，牛顿插值多项式与拉格朗日是一致的，但是牛顿插值法可以通过均差表快速手工计算。因此构建均差表也是数值计算的任务之一。

对于原始数据[-1,-2],[1,0],[3,-6],[4,9]，通过牛顿插值可以在所求点$x = 0, 2, 2.5$上获得插值函数值：**[4.2, -6.2 , -7.425]**。生成图像为



均差表为：**[-2, 0, -6, 9], [1.0, -3.0, 15.0], [-1.0, 6.0], [1.4]**

代码实现

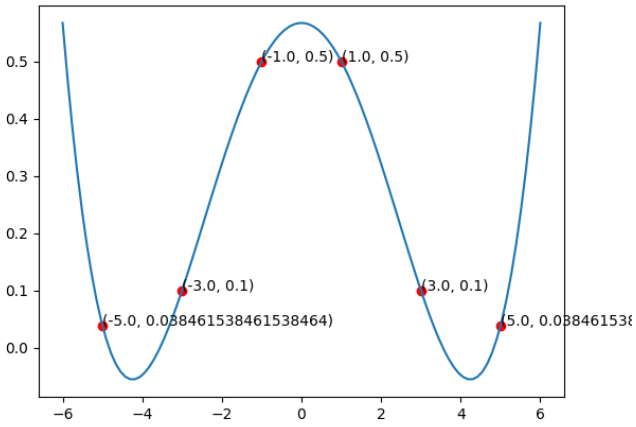
```
import numpy as np
import matplotlib.pyplot as plt
from Newton import Newton  #通过上述自建库调用

"待差值的原始数据"
AXY = np.array([[-1,-2],[1,0],[3,-6],[4,9]])
NT = Newton(AXY)
"计算x=0,2,2.5的近似值"
x0 = np.array([0,2,2.5])
y0 = np.array([])
for x in x0:
    y0 = np.append(y0,NT.num(x))
print(y0)
"生成图像"
NT.visualize(-2,5,1000,True)
"生成差值表"
Table = NT.f()[1]
print(Table)
```

Q3 实现拉格朗日、分段线性、分段Hermite以及三次样条插值并绘图

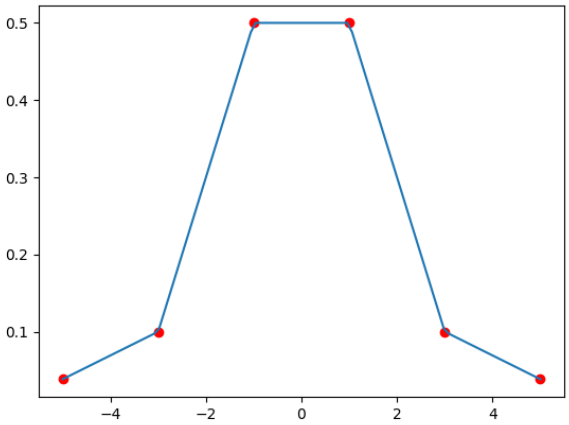
首先需要定义函数及其导函数。

答案部分

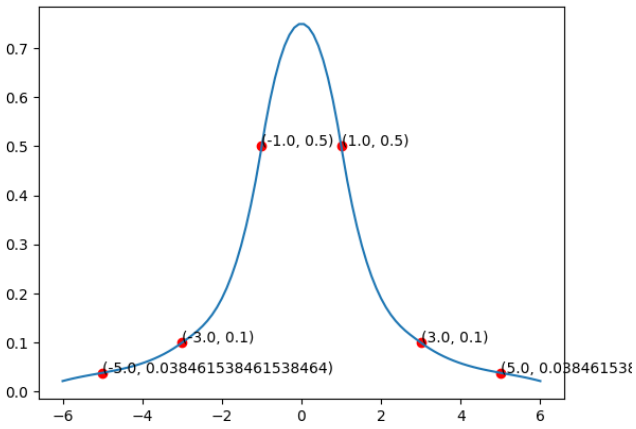
对于拉格朗日插值，对应区间插值为：**[0.03846154 ,-0.04603365, -0.04807692 , 0.0078125 , 0.1 0.20973558, 0.32115385 , 0.42127404 ,0.5 , 0.55012019, 0.56730769, 0.55012019, 0.5 , 0.42127404 , 0.32115385, 0.20973558 , 0.1 , 0.0078125, -0.04807692, -0.04603365 , 0.03846154]**。插值图像为：



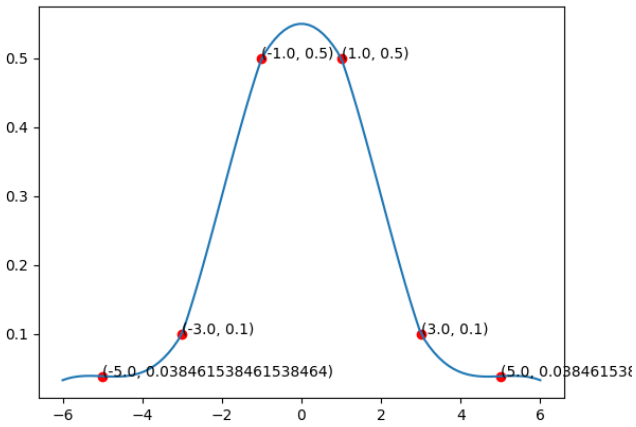
对于分段线性插值，对应区间插值为：**[0.03846154 ,0.05384615 ,0.06923077 ,0.08461538 ,0.1 ,0.2 ,0.3 , 0.4 , 0.5 ,0.5 ,0.5 ,0.5 ,0.5 ,0.4 , 0.3 ,0.2 ,0.1 ,0.08461538 ,0.06923077, 0.05384615 ,0.03846154]**



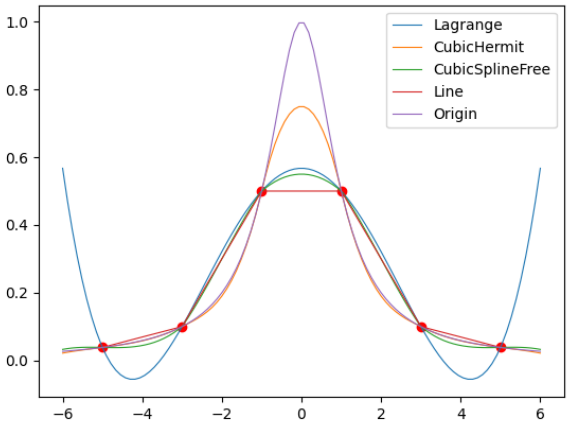
对于三次Hermite插值，对应区间插值为：**[0.03846154 ,0.04661243, 0.05792899, 0.07489645, 0.1 ,0.1325 , 0.19 ,0.3025 , 0.5 , 0.6875 , 0.75 , 0.6875 ,0.5 ,0.3025 , 0.19 , 0.1325 , 0.1 , 0.07489645 ,0.05792899 ,0.04661243, 0.03846154]**



对于自由三次样条插值，对应区间插值为：**[0.03846154, 0.03822115, 0.04423077, 0.06274038, 0.1, 0.19375, 0.3, 0.40625, 0.5, 0.5375, 0.55, 0.5375, 0.5, 0.40625, 0.3, 0.19375, 0.1, 0.06274038, 0.04423077, 0.03822115 ,0.03846154]**



原曲线与插值曲线对比图为：



代码部分

```
import numpy as np
import matplotlib.pyplot as plt
from Lagrange import Lagrange
from CubicSplineFree import CubicSplineFree
from CubicHermit import  CubicHermit
from scipy.interpolate import CubicSpline
from scipy import interpolate as spip

"原函数"
def OriginF(x):
    return 1/(1+x**2)
"原函数的导数"
def DOriginF(x):
    return -2*x/((1+x**2)**2)
X = np.linspace(-5,5,6)
```

```
Y = OriginF(X)
AXY = np.c_[X,Y]
dY = DOriginF(X)
x0 = np.linspace(-5,5,21)

"拉格朗日差值部分"
LA = Lagrange(AXY)
Num_LA = np.array([])
for x in x0:
    Num_LA = np.append(Num_LA,LA.num(x))
print("拉格朗日=",Num_LA)
LA.visualize(-6,6,100,True)

"自由三次样条部分"
CSF = CubicSplineFree(AXY)
Num_CSF = np.array([])
for x in x0:
    Num_CSF = np.append(Num_CSF,CSF.num(x))
print("自由三次样条=",Num_CSF)
CSF.visualize(-6,6,100,True)

"带导数的分段三次Hermit差值"
CH = CubicHermit(AXY,dY)
Num_CH = np.array([])
for x in x0:
    Num_CH = np.append(Num_CH,CH.num(x))
print("分段三次Hermite=",Num_CH)
CH.visualize(-6,6,100,True)

"Scipy自带函数"
"分段线性插值"
Line = spip.interp1d(X,Y,kind='linear')
print("分段线性=",Line(x0))
xs = np.linspace(-5,5,100)
ys = Line(xs)
fig = plt.figure()
plt.scatter(X,Y , c='red')
plt.plot(xs,ys)
plt.show()

"绘制全部曲线"
fig = plt.figure()
xt = np.linspace(-6,6,100)
y_LA = np.array([])
y_CH = np.array([])
y_CSF = np.array([])
for x in xt:
    y_LA = np.append(y_LA,LA.num(x))
    y_CH = np.append(y_CH, CH.num(x))
```

```
    y_CSF = np.append(y_CSF, CSF.num(x))
print(y_LA)
llw = 0.8
plt.scatter(X,Y , c='red')
plt.plot(xt,y_LA,label = "Lagrange",lw = llw)
plt.plot(xt,y_CH,label = "CubicHermit",lw = llw)
plt.plot(xt,y_CSF,label = "CubicSplineFree",lw = llw)
plt.plot(xs,Line(xs),label = "Line",lw = llw)
plt.plot(xt,OriginF(xt),label = "Origin",lw = llw)
plt.legend()
plt.show()
```