这次，我们将通过插值、拟合与线性方程组的数学描述导出数值方法，并用其解决研究生课程《数值计算》大作业中的部分问题。本文涉及的全部代码及其用法可前往Github仓库https://github.com/DMCXE/Numerical_Computation 查看。

1 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：

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

其中，$l_k(x)$表示为：

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

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	$f(x_4)$	$f[x_3,x_4]$	$f[x_2,x_3,x_4]$	$f[x_1,x_2,x_3,x_4]$	$f[x_0,x_1,x_2,x_3,x_4]$
…	…	…	…	…	…

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
```

```python
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，我们讨论三次自由样条插值。

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：

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

由于一阶导数连续，即在分割点处左右导数相等可得：

$$\mu_jM_{j-1}+2M_j+\lambda_jM_{j+1}=d_j,\quad j=1,2,\cdots,n-1$$

其中：

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

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()
```

2 曲线拟合

对于函数类A中给定的函数$f(x)$，记作$f(x) \in A$，要求在另一类简单的便于计算的函数类B中求函数$p(x) \in B$，使$p(x)$与$f(x)$的误差在某种度量意义下最小。

2.1 曲线拟合的最小二乘法

2.1.1 数学定义

对于在数据$\{(x_i,y_i),i=0,1,\cdots,m\}$上的拟合函数$S^*(x)$，设$\varphi=\operatorname{span}\{\varphi_0(x),\varphi_1(x),\cdots,\varphi_n(x)\}$为$C[a,b]$上线性无关函数族，找一函数$S^*(x)$，使误差平方和

$$\|\boldsymbol{\delta}\|_2^2=\sum_{i=0}^m\delta_i^2=\sum_{i=0}^m\left[S^*\left(x_i\right)-y_i\right]^2=\min_{S(x)\in\varphi}\sum_{i=0}^m\left[S\left(x_i\right)-y_i\right]^2$$

这里

$$S(x)=a_0\varphi_0(x)+a_1\varphi_1(x)+\cdots+a_n\varphi_n(x)\quad(n<m)$$

当$\varphi=\operatorname{span}\{\varphi_0(x),\varphi_1(x),\cdots,\varphi_n(x)\}$取$\varphi=\operatorname{span}\{1,x,\cdots,x^n\}$时，满足**Haar条件**，即一定能找到唯一一组$\{a_0,a_1,\cdots,a_n\}$使得$S(x)$存在最小值。

要使得获得最小误差平方和，可以等价为取得全部元素加权平方和的最小值，也就是将系数

$\{a_0,a_1,\cdots,a_n\}$看作一组未知量。对其求偏导可以得到如下极小值存在的条件和定义：若记

$$(\varphi_j,\varphi_k)=\sum_{i=0}^m\omega\left(x_i\right)\varphi_j\left(x_i\right)\varphi_k\left(x_i\right)$$

$$(f,\varphi_k)=\sum_{i=0}^m\omega\left(x_i\right)f\left(x_i\right)\varphi_k\left(x_i\right)\equiv d_k,\quad k=0,1,\cdots,n$$

满足：

$$\sum_{j=0}^n(\varphi_k,\varphi_j)a_j=d_k,\quad k=0,1,\cdots,n$$

即

$$\begin{pmatrix}(\varphi_0,\varphi_0)&(\varphi_0,\varphi_1)&\cdots&(\varphi_0,\varphi_n)\\(\varphi_1,\varphi_0)&(\varphi_1,\varphi_1)&\cdots&(\varphi_1,\varphi_n)\\\vdots&\vdots&&\vdots\\(\varphi_n,\varphi_0)&(\varphi_n,\varphi_1)&\cdots&(\varphi_n,\varphi_n)\end{pmatrix}\begin{pmatrix}a_0\\a_1\\\vdots\\a_n\end{pmatrix}=\begin{pmatrix}d_0\\d_1\\\vdots\\d_n\end{pmatrix}$$

$$Ga=d$$

2.1.2 算法实现

最小二乘法的算法实现即通过循环嵌套翻译数学表达式

$$(\varphi_j,\varphi_k)=\sum_{i=0}^m\omega\left(x_i\right)\varphi_j\left(x_i\right)\varphi_k\left(x_i\right)$$

$$(f,\varphi_k)=\sum_{i=0}^m\omega\left(x_i\right)f\left(x_i\right)\varphi_k\left(x_i\right)\equiv d_k,\quad k=0,1,\cdots,n$$

并求解线性方程组。求解线性方程组的方法将在最后一章给出。下面给出上述描述的核心代码

```
    def phiprod(self):
        #确定总长度
        n = self.n
        #初始化G,d向量
        G = np.array([])
        d = np.array([])
        #计算并生成G,d向量
        for i in range(0,n):
            d = np.append(d,np.sum((self.arr1_y)*(self.arr1_x**i)))
            for j in range(0,n):
                #这里的G向量是有n个元素的行向量
                G = np.append(G,np.sum((self.arr1_x**i)*(self.arr1_x**j)))
        #通过.reshape方法将G向量转为n阶方阵
        G = G.reshape(n,n)
        #通过np求逆求解，待更新轮子解法
        #an = np.dot(np.linalg.inv(G), d)
        #通过自制LU求解器
        an = self.MartrixSolver(G,d)
        return an,G,d
```

对于满足最小化的an，代入

$$S(x) = a_0\varphi_0(x) + a_1\varphi_1(x) + \cdots + a_n\varphi_n(x) \quad (n < m)$$

即可确定最小二乘拟合函数。以函数值方式给出，代码如下所示：

```
    def num(self,x):
        num = 0
        for i in range(0,self.n):
            num = num+(self.an[i])*(x**i)
        return num
```

对应的平方误差可通过定义给出。

```
    def delta(self):
        de = np.zeros(self.lenth)
        for i in range(0,self.lenth):
            de[i] = (self.num(self.arr1_x[i])-self.arr1_y[i])**2
        return np.min(de)
```

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`FitSquares.py`下查看。

```
import numpy as np
import matplotlib.pyplot as plt
import time
class FitSquares_polynomial:
    def __init__(self,arr1,n):
```

```
        self.arr1 = arr1
        self.arr1_x = arr1[:,0]
        self.arr1_y = arr1[:,1]
        self.lenth = len(arr1)
        self.n = n
        self.an = self.phiprod()[0]

    def phiprod(self):
        #确定总长度
        n = self.n
        #初始化G,d向量
        G = np.array([])
        d = np.array([])
        #计算并生成G,d向量
        for i in range(0,n):
            d = np.append(d,np.sum((self.arr1_y)*(self.arr1_x**i)))
            for j in range(0,n):
                #这里的G向量是有n个元素的行向量
                G = np.append(G,np.sum((self.arr1_x**i)*(self.arr1_x**j)))
        #通过.reshape方法将G向量转为n阶方阵
        G = G.reshape(n,n)
        #通过np求逆求解，待更新轮子解法
        #an = np.dot(np.linalg.inv(G), d)
        #通过自制LU求解器
        an = self.MartrixSolver(G,d)
        return an,G,d

    def num(self,x):
        num = 0
        for i in range(0,self.n):
            num = num+(self.an[i])*(x**i)
        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()

    def delta(self):
        de = np.zeros(self.lenth)
```

```
        for i in range(0,self.lenth):
            de[i] = (self.num(self.arr1_x[i])-self.arr1_y[i])**2
        return np.min(de)

    #LU分解
    def MartrixSolver(self,A, d):
        n = len(A)
        U = np.zeros((n, n))
        L = np.zeros((n, n))

        for i in range(0, n):
            U[0, i] = A[0, i]
            L[i, i] = 1
            if i > 0:
                L[i, 0] = A[i, 0] / U[0, 0]
        # LU分解
        for r in range(1, n):
            for i in range(r, n):
                sum1 = 0
                sum2 = 0
                ii = i + 1
                for k in range(0, r):
                    sum1 = sum1 + L[r, k] * U[k, i]
                    if ii < n and r != n - 1:
                        sum2 = sum2 + L[ii, k] * U[k, r]
                U[r, i] = A[r, i] - sum1
                if ii < n and r != n - 1:
                    L[ii, r] = (A[ii, r] - sum2) / U[r, r]
        # 求解y
        y = np.zeros(n)
        y[0] = d[0]

        for i in range(1, n):
            sumy = 0
            for k in range(0, i):
                sumy = sumy + L[i, k] * y[k]
            y[i] = d[i] - sumy
        # 求解x
        x = np.zeros(n)
        x[n - 1] = y[n - 1] / U[n - 1, n - 1]
        for i in range(n - 2, -1, -1):
            sumx = 0
            for k in range(i + 1, n):
                sumx = sumx + U[i, k] * x[k]
            x[i] = (y[i] - sumx) / U[i, i]

        return x
```

3 线性方程组求解

3.1 直接LU分解求解线性方程组

3.1.1 数学原理

定理（矩阵的LU分解）：设A为n阶举证，如果A的顺序主子式$D_i \neq 0 (i = 1, 2, \cdots, n-1)$，则A可分解为一个单位下三角矩阵L和一个上三角矩阵U的乘积，且这种分解是唯一的。

采用LU分解，对于方程$\mathbf{Ax} = \mathbf{b}$，由于$\mathbf{A} = \mathbf{LU}$，则等价为求解两个三角形方程组$\mathbf{Ly} = \mathbf{b}, \mathbf{Ux} = \mathbf{y}$

杜利特尔（Doolittle）分解：若A非奇异且LU分解存在，即A能够被分解为：

$$\boldsymbol{A} = \begin{pmatrix} 1 & & & \\ l_{21} & 1 & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \cdots & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & \cdots & u_{2n} \\ & & \ddots & \vdots \\ & & & u_{nn} \end{pmatrix}$$

Doolittle分解步骤为:

1. $u_{1i} = a_{1i} (i = 1, 2, \cdots, n), l_{i1} = a_{i1} / u_{11}, i = 2, 3, \cdots, n$

   计算**U**的第r行，**L**的第r列$(r = 2, 3, \cdots, n)$
2. $u_{ri} = a_{ri} - \sum_{k=1}^{r-1} l_{rk} u_{ki}, i = r, r+1, \cdots, n$
3. $l_{ir} = \left(a_{ir} - \sum_{i=1}^{r-1} l_{ik} u_{kr} \right) / u_{rr}, i = r+1, \cdots, n$, 且 $r \neq n$

   求解$\mathbf{Ly} = \mathbf{b}, \mathbf{Ux} = \mathbf{y}$
4. $\begin{cases} y_1 = b_1, \\ y_i = b_i - \sum_{k=1}^{i-1} l_{ik} y_k, i = 2, 3, \cdots, n \end{cases}$
5. $\begin{cases} x_n = y_n / u_{nn} \\ x_i = \left(y_i - \sum_{k=i+1}^{n} u_{ik} x_k \right) / u_{ii}, i = n-1, n-2, \cdots, 1 \end{cases}$

3.1.2 算法实现

直接三角分解的算法实现较为简单，只需要将分解的数学步骤翻译为编程语言即可。但由于设计两个矩阵的分解，需要多次循环嵌套。在python中受限于编译器速度，当矩阵规模变大时，需要计算很长的时间。若使用`numpy.linalg.solve`进行计算，将获得千倍的速度提升。（以500×500的满秩随机矩阵为例，自编`MartrixSolver`需要耗时8.2961781s，`numpy.linalg.solve`仅耗时0.126146s）

```
def MartrixSolver(A,d):
    n = len(A)
    U = np.zeros((n,n))
    L = np.zeros((n,n))
    #LU分解的初始值，对应第一步
    for i in range(0,n):
        U[0,i] = A[0,i]
        L[i,i] = 1
```

```
        if i>0:
            L[i,0] = A[i,0]/U[0,0]
    #LU分解，对应2，3步
    for r in range(1,n):
        for i in range(r,n):
            sum1 = 0
            sum2 = 0
            ii = i + 1
            for k in range(0,r):
                sum1 = sum1 + L[r,k]*U[k,i]
                if ii < n and r != n-1:
                    sum2 = sum2 + L[ii,k]*U[k,r]
            U[r,i] = A[r,i]-sum1
            if ii < n   and r != n-1:
                L[ii,r] = (A[ii,r]-sum2)/U[r,r]
    #求解y
    y = np.zeros(n)
    y[0] = d[0]
    for i in range(1,n):
        sumy = 0
        for k in range(0,i):
            sumy = sumy + L[i,k]*y[k]
        y[i] = d[i] - sumy
    #求解x
    x = np.zeros(n)
    x[n-1] = y[n-1]/U[n-1,n-1]
    for i in range(n-2,-1,-1):
        sumx = 0
        for k in range(i+1,n):
            sumx = sumx + U[i,k]*x[k]
        x[i] = (y[i]-sumx)/U[i,i]

    return x
```

3.2 追赶法

追赶法是LU分解的一种一种变式，解决的是系数矩阵为对角占优的三对角线性方程组。

3.2.1 数学原理

在传热学数值计算那篇文章中，已经提过了这一概念。三对角线性方程组形如下所示：

$$\begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-1} \\ f_n \end{pmatrix}$$

且满足：$|b_1|>|c_1|>0;\ |b_i|\geqslant|a_i|+|c_i|,a_i,c_i\neq0,i=2,3,\cdots,n-1;|b_n|>|a_n|>0$。。三对角线性方程组系数行列式可被LU分解为：

$$\begin{pmatrix}\alpha_1 & & & \\ r_2 & \alpha_2 & & \\ & \ddots & \ddots & \\ & & r_n & \alpha_n\end{pmatrix}\begin{pmatrix}1 & \beta_1 & & \\ & 1 & \ddots & \\ & & \ddots & \beta_{n-1} \\ & & & 1\end{pmatrix}$$

将LU分解的方法带入到其中，可以得到**追赶法公式**：

1. 计算$\{\beta_i\}$的递推公式：

   $\beta_1=c_1/b_1$

   $\beta_i=c_i/(b_i-a_i\beta_{i-1}),i=2,3,\cdots,n-1$

2. 求解$\mathbf{Ly}=\mathbf{f}$

   $y_1=f_1/b_1$

   $y_i=(f_i-a_iy_{i-1})/(b_i-a_i\beta_{i-1}),i=2,3,\cdots,n$

3. 求解$\mathbf{Ux}=\mathbf{y}$

   $x_n=y_n$

   $x_i=y_i-\beta_ix_{i+1},\ i=n-1,n-2,\cdots,2,1$

3.1.2 算法实现

仅需要翻译上述算法即可。

```
    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
```

案例

以下案例来源于研究生课程——数值计算中的大作业内容。

Q1:关于插值

题目

美国的人口普查每10年举行一次，下表列出了从1960年到2020年的人口(按千人计)。

年	1960年	1970年	1980年	1990年	2000年	2010年	2020年
人口(千)	180 671	205 052	227 225	249 623	282 162	309 327	329 484

（1）用适当Lagrange插值法分别求在1950年、2005年和2030年人口的近似值。

（2）1950年的人口大约是151326（千人），你认为你得到的2005年（据查295,516千人）和2030年（预测）的人口数字精确度如何?

（3）用适 Newton插值法重做（1）和（2）。

（4）使用适当自由三次样条插值法重做（1）和（2）。

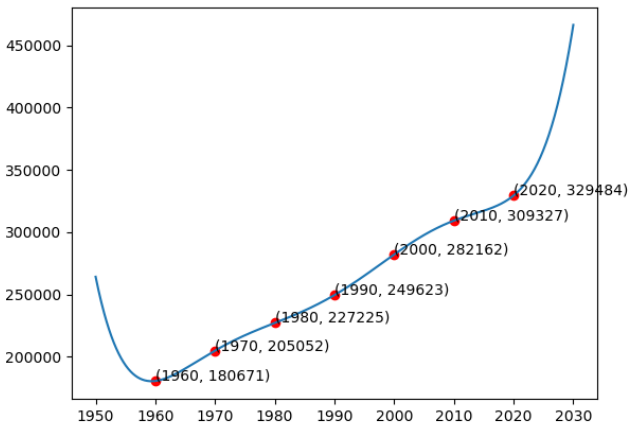
解答及代码

代码如下：

```
from Lagrange import Lagrange as la
from Newton import Newton as Ne
from CubicSplineFree import CubicSplineFree as CS
import numpy as np
import matplotlib.pyplot as plt

arr = np.array([[1960,180671],[1970,205052],[1980,227225],[1990,249623],[2000,282162],
[2010,309327],[2020,329484]])
```

```
#Lagrange插值法实例化
Lag = la(arr)
#牛顿插值法实例化
New = Ne(arr)
#三次样条插值实例化
Csf = CS(arr)

#可视化
Lag.visualize(1950,2030,1000,text=True)
New.visualize(1950,2030,1000,text=True)
Csf.visualize(1950,2030,1000,text=True)

#Lagrange部分
po0_L = Lag.num(1950)
eff0_L = abs(Lag.num(1950)-151326)/151326
po1_L = Lag.num(2005)
eff1_L = abs(Lag.num(2005)-295516)/295516
po2_L = Lag.num(2030)
print(po0_L,po1_L,po2_L,eff0_L,eff1_L)
#牛顿部分
po0_N = New.num(1950)
eff0_N = abs(New.num(1950)-151326)/151326
po1_N= New.num(2005)
eff1_N = abs(New.num(2005)-295516)/295516
po2_N = New.num(2030)
print(po0_N,po1_N,po2_N,eff0_N,eff1_N)
#三次样条部分
po0_C = Csf.num(1950)
eff0_C = abs(Csf.num(1950)-151326)/151326
po1_C = Csf.num(2005)
eff1_C = abs(Csf.num(2005)-295516)/295516
po2_C = Csf.num(2030)
print(po0_C,po1_C,po2_C,eff0_C,eff1_C)
```

第一/二问：Lagrange拟合曲线为：

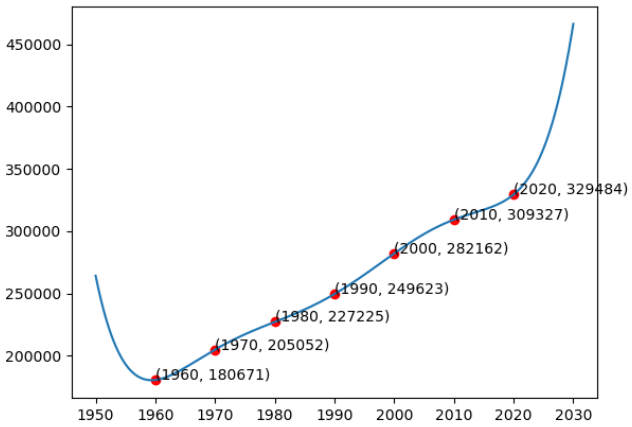


预测1950年人口为：264272.0，偏差：74.64%

预测2005年人口为：297798.13，偏差：0.7723%

预测2030年人口为：466418.0

第三问：牛顿插值拟合曲线为：



预测1950年人口为：264272.0，偏差：74.64%

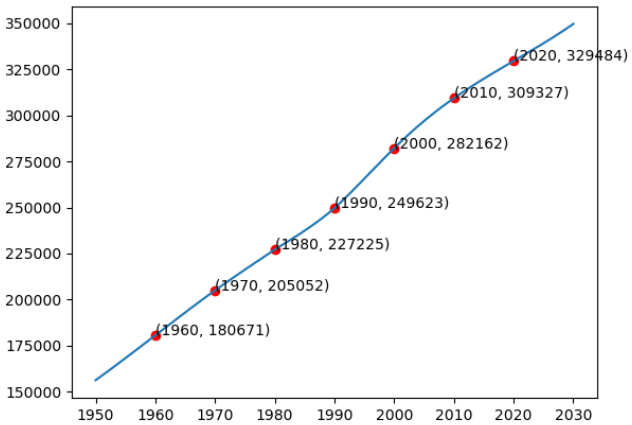
预测2005年人口为：297798.13，偏差：0.7723%

预测2030年人口为：466418.0

均差表为：[[180671, 205052, 227225, 249623, 282162, 309327, 329484], [2438.1, 2217.3, 2239.8, 3253.9, 2716.5, 2015.7], [-11.039999999999987, 1.125, 50.705, -26.870000000000005, -35.04], [0.4054999999999996, 1.6526666666666665, -2.5858333333333334, -0.27233333333333315], [0.031179166666666674, -0.1059625, 0.05783750000000001], [-0.0027428333333333332, 0.003276], [0.00010031388888888887]]

与Lagrange插值法的结果相同，验证了定理的正确性。

第四问：三次自由样条插值为：



预测1950年人口为：156290.0，偏差：3.28%

预测2005年人口为：296944.13，偏差：0.4834%

预测2030年人口为：349641.0

三次样条插值优于前两者。

Q2：关于拟合

题干

生物学家 在研究天蛾幼虫的生长时采用了下面的数据确定 (活幼虫的重量，以克计算)和 (幼虫消耗的氧气，以毫升/小时计算)之间的关系 。

w	R	w	R	w	R	w	R	w	R
0.017	0.154	0.174	0.363	1.29	0.87	3.04	3.59	4.83	4.66
0.020	0.181	0.210	0.428	1.32	1.15	3.34	2.83	5.30	3.88
0.025	0.234	0.211	0.366	1.35	2.48	4.09	3.58	5.45	3.52
0.085	0.260	0.233	0.537	1.69	1.44	4.28	3.28	5.48	4.15
0.087	0.296	0.783	1.47	1.74	2.23	4.29	3.40	5.53	6.94
0.119	0.299	0.999	0.771	2.75	1.84	4.58	2.96	5.96	2.40
0.171	0.334	1.11	0.531	3.02	2.01	4.68	5.10		

（1）利用对数最小二乘方程$lnR = lnb + alnw$拟合，确定参数a,b。

（2）计算（1）中的平方误差。

（3）修改（1）中的对数最小二乘方程$lnR = lnb + alnw + c(lnw)^2$，确定参数a,b,c 。

（4）计算（3）中的平方误差。

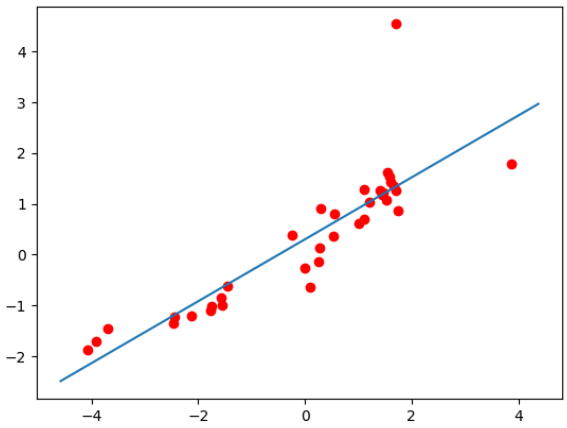
解答及代码

代码如下所示：

```
from FitSquares import FitSquares_polynomial as FSs
import numpy as np
w = np.array([0.017,0.02,0.025,0.085,0.087,0.119,0.171,
              0.174,0.210,0.211,0.233,0.783,0.999,1.11,
              1.29,1.32,1.35,1.69,1.74,2.75,3.02,
              3.04,3.34,4.09,4.28,4.29,4.58,4.68,
              4.83,5.30,5.45,5,48,5.53,5.69])
R = np.array([0.154,0.181,0.234,0.260,0.296,0.299,0.334,
              0.363,0.428,0.366,0.537,1.47,0.771,0.531,
              0.87,1.15,2.48,1.44,2.23,1.84,2.01,
              3.59,2.83,3.58,3.28,3.40,2.96,5.10,
              4.66,3.88,3.52,4.15,6,94,2.40])
R_ln = np.log(R)
w_ln = np.log(w)
arr = np.c_[w_ln,R_ln]
S2 = FSs(arr,2)
```

```
S3 = FSs(arr,3)
mi = np.min(w_ln)
ma = np.max(w_ln)
S2.visualize(mi-0.5,ma+0.5,1000,False)
S3.visualize(mi-0.5,ma+0.5,1000,False)
an2 = S2.an
an3 = S3.an
b2 = np.e**(an2[0])
b3 = np.e**(an3[0])
print(b2,an2[1:],S2.delta())
print(b3,an3[1:],S3.delta())
```

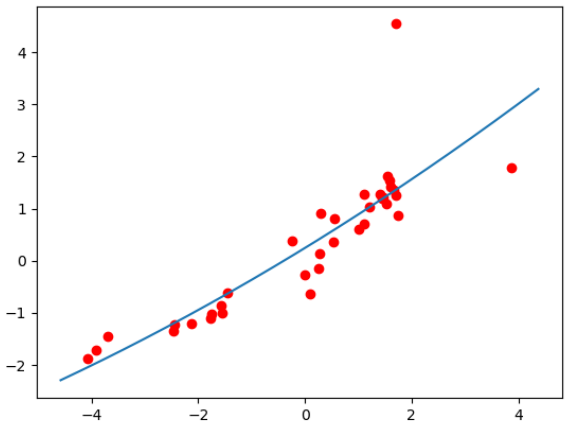
第一问：$lnR = lnb + alnw$对应的拟合曲线为：



$b = 1.3534, a = 0.6103$

第二问：$lnR = lnb + alnw$对应的误差为：$2.6305502895585346e-06$

第三问：$lnR = lnb + alnw + c(lnw)^2$对应拟合曲线为：



$b = 1.2786, a = 0.6275, c = 0.0160$

第四问：$lnR = lnb + alnw + c(lnw)^2$对应的误差为：$1.7143596831005267e-05$