数值分析第3次作业

211240021 田铭扬

（一）最佳平方逼近多项式

§1 问题

求 $f(x)=e^x$ 在区间[0,1]上的n次最佳平方逼近多项式 $\varphi(x)=\sum_{k=0}^{n} a_k \varphi_k(x)$ ，
(1) n=5, $\varphi_k(x)=x^k$; (2) n=5, $\varphi_k(x)$为k次Legendre多项式;
(3) n=10, $\varphi_k(x)=x^k$; (4) n=10, $\varphi_k(x)$为k次Legendre多项式.

§2 算法分析

·2.1 问题(1)与(3)

考察教材P325的线性方程组(3.5)，记$Ar=b$。其中 $A=((\varphi_i,\varphi_j))_{0\le i,j\le n}$ ，而注意到 $(\varphi_i,\varphi_j)=\int_0^1 x^{i+j}dx=\frac{1}{i+j+1}$ ，即A为n+1阶Hilbert矩阵H_{n+1}，故此式可以改写为$r=H_{n+1}^{-1}b$。

Matlab中有内置的Hilbert逆矩阵，而 $b=(\int_0^1 e^x dx,\ldots,\int_0^1 x^n e^x dx)$ 可由数值积分得出。进而可以计算得到所求最佳平方逼近多项式的系数向量r。

·2.2 问题(2)与(4)

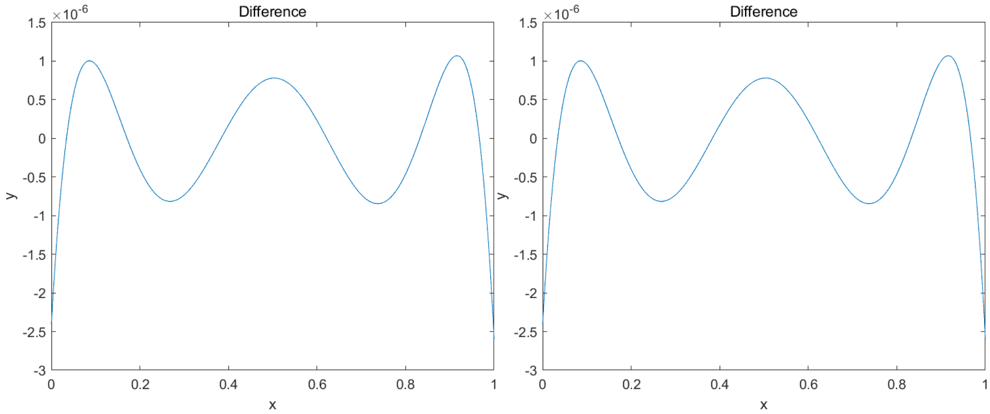
由于希望利用到Legendre多项式的直交性，我们将问题变换到[-1,1]上。此时需要逼近的函数为$f_1(x)=e^{(x+1)/2}$. Matlab中有内置的Legendre多项方式方法，故直接按照教材P327页的(3.7)式计算即可。

§3 运行结果与分析

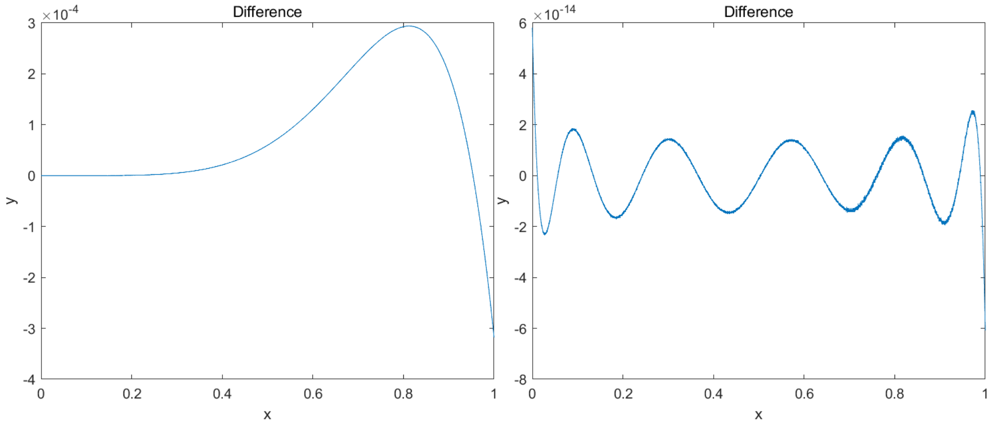
由于两种逼近方式都很好，下一页给出的是误差函数的图像。

问题(1)与(3)的运行结果中出现了意想不到的现象：根据图像，当最佳平方逼近多项式的次数取n=5时，绝对误差界$\varepsilon\approx2.5*10^{-6}$；而当n=10时，$\varepsilon\approx3*10^{-4}$，误差变大。推测这是由于Hilbert矩阵的“病态”性质，在n较大时算法不稳定导致的。注意到(1)和(2)图像的“形状”十分相似，说明在n=5时用幂函数逼近的方法较为准确。这是支持以上的推测的证据。

问题(2)和(4)的运行结果符合预期。根据图像，次数n=5时，绝对误差界$\varepsilon\approx2.5*10^{-6}$；n=10时，$\varepsilon\approx6*10^{-14}$，误差随逼近多项式次数的增加而减小。还有一个有意思的现象：n=10时的误差函数图像，相比n=5时有更多的“波动”。这一现象是可以理解的，回忆最佳一致逼近多项式的“交错点组”，或者是多项式插值的情况，甚至是离散拟合时的过拟合现象，都与此一致。



左上:问题(1) 右上:问题(2) 左下:问题(3) 右下:问题(4)



§4 完整代码

问题(1)与(3)的Matlab代码如下：

```
n=input("n=");
r=zeros(n+1,1);
for i=1:(n+1)
    r(i)=integral(@(x) exp(x).*(x.^(i-1)),0,1);
end
r=flip(invhilb(n+1)*r);
//将所求多项式以系数形式保存在向量r中，以下是画图
x=0:0.0001:1;
y=polyval(r,x);
figure(1)
plot(x,y,x,exp(x));
title('Result');
legend("Poly","Original","Location","north")
figure(2)
plot(x,y-exp(x));
title('Difference');
```

问题(2)与(4)的Matlab代码如下：

```
//函数文件pro2_f.m
function y=pro2_f(n,x)
    z=legendre(n,x);
    y=z(1,:).*exp((x+1)./2);
    //Matlab中的legendre方法得到的是“连带Legendre函数”，输出为一个矩阵，
矩阵第1行即所需Legendre多项式的系数
```

```
//主程序
n=input("n=");
r=zeros(n+1,1);
for i=1:(n+1)
    r(i)=(i-0.5)*integral(@(x) pro2_f(i-1,x),-1,1);
end
x=0:0.0001:1;
y=0.*x;
for i=1:(n+1)
    z=legendre(i-1,2.*x-1);
    y=y+z(1,:)*r(i);
end
figure(1)
plot(x,y,x,exp(x));
title('Result');
legend("Poly","Original","Location","north")
figure(2)
plot(x,y-exp(x));
title('Difference');
```

（二）快速傅里叶变换(FFT)

§1 特殊情况——数据点数为 2 的幂

课堂上讲解给出了这一算法的详细推导，按照 ppt 上的伪代码进行实现只是“体力活”，因此无需累述。由于下文解决一般情况（数据点数不为 2 的幂）时也需要用到，代码请见第 4 部分程序的 FFT()函数。

另外注意，课堂 ppt 给出的算法的第 4 步中，$A_1(N)\to A_0(N)$可以通过数组指针的交换实现，这能够节省约 NlogN 次寻址与赋值操作。

§2 Bluestein 算法解决一般情况

在许多实际应用（例如声音信号的频谱分析）中，可以在采样时采集数量为 2 的幂的数据；即使数据数量不是 2 的幂，也可以在末尾加入若干个 0 来补足。但是，相比于对原来的数据直接进行 DFT，“补 0”得到的结果实际上是本来结果的一种“插值”，这并不尽如人意。我们还是需要一种在一般情况下快速进行 DFT 的算法。通过查阅资料，作者选择使用 Bluestein 算法。

对于 N 个数据的情况，我们的目标是计算 $y_k=\sum_{j=0}^{n-1} x_j \omega^{kj}\ (0\le k<N)$ ，其中 $\omega=e^{-i\frac{2\pi}{N}}$ 。注意到有恒等式 $jk=\binom{j+k}{2}-\binom{j}{2}-\binom{k}{2}$ ，故原式可变形为

$$y_k=\omega^{-\binom{k}{2}}\sum_{j=0}^{n-1} x_j \omega^{-\binom{j}{2}} \omega^{\binom{j+k}{2}}\ (0\le k<N)$$

即令 $A_j=x_j\omega^{-\binom{j}{2}}\ (0\le j<N)$ $B_j=\omega^{\binom{j}{2}}\ (0\le j<2N-1)$ ，有 $y_k=A_k\sum_{j=0}^{N-1} A_j B_{j+k}$ 。求和符号后面的部分其实是数列 A_j 与 B_{2N-2-j}（以下简记为 A，B）的卷积，而数列的卷积可以通过三次 FFT 变换得出，并且“补 0”不影响计算结果。

具体而言，将数列（数组）A与B补0至2^n $\left(n=max\{n|2^n\geq 3N-2\}\right)$ 项，然后做FFT变换得到A′和B′。再令数列C=A′B′，做IFFT变换——即在FFT的基础上，ω取原来的共轭，且结果除以2^n——得到C′。由于A,B分别有N-1和2N-1项，C′共有3N-2项，这也是n取法的由来。取C′“中部”的N项即为所求：$y_k=A_kC'_{2N-1-K}$ (0≤k<n)。

接下来计算Bluestein算法的复杂度。三次FFT变换，每次的时间复杂度是$O(2^n\log(2^n))=O(n2^n)$，n的定义与上一段中相同，此外还有若干次时间复杂度为O(N)或$O(2^n)$的操作。综上，可以认为时间复杂度为$O(n2^n)$，且有常数3。又注意到$2^n\geq 3N-2$，因而相比于特殊情况的O(NlogN)，一般情况下的耗时会更长。此外，两种情况的空间复杂度（O(N)或$O(2^n)$）相近。

§3 运算结果

手动尝试几组数据，运算结果与使用Matlab的fft方法得到的结果一直。

§4 FFT的应用

FFT有很多应用场景。

在比较“抽象”的领域，例如在Bluestein算法中已经用到的，FFT可以用于计算数列的卷积。事实上，它的本质是在计算多项式乘法——上一部分中的数列A, B和C′均可视为多项式的系数，而FFT与IFFT则为多项式系数表示法与点值（即各2^n次单位根处的值）表示法之间转换。而由多项式乘法衍生，FFT算法还能够用于计算p进制下的大数乘法：把p进制数$(a_na_{n-1}\dots a_1a_0)_p$视为多项式 $f(x)=\sum_{k=0}^{n} a_k x^k$ 在x=p处的取值即可。

而在更为具体的领域，由于傅里叶变换自身的一些性质（主要是函数族 $\{1, cosx, sinx, \dots, \cos nx, \sin nx, \dots\}$ 的直交性）和三角函数的物理意义，FFT可被用于频谱的分析。例如，假如连续函数f(t)是一段波（可以是声波、光波乃至概率波（此时自变量是位置x而非时间t）），对它进行傅里叶变换，得到的$f_1(\omega)$就代表这段波对于频率ω（ω为负表示反向的波）的能量分布。而在实际场景中，我们采样得到的只是f(t)在某些点处的值，因而需要进行的就是离散傅里叶变换DFT，可以使用FFT算法加快计算速度。

§5 完整代码

下一页是是Bluestein算法的完整代码，使用C++实现。其中的FFT()子函数可以在略加修改（将数组w的计算挪到函数内）后直接用于数据点数为2的幂的特殊情况。

```
#include <cstdio>
#include <cmath>
#include <cstdlib>
using namespace std;
struct cplx{
    double real,imag;
    cplx(double x=0,double y=0){real=x,imag=y;}
};
cplx operator + (cplx a,cplx b){
    return cplx(a.real+b.real,a.imag+b.imag);
}
cplx operator - (cplx a,cplx b){
    return cplx(a.real-b.real,a.imag-b.imag);
}
cplx operator * (cplx a,cplx b){
    return cplx(a.real*b.real-a.imag*b.imag,a.real*b.imag+a.imag*b.real);
}
cplx bar(cplx x){
    x.imag=(-x.imag); return x;
}//复数的实现
cplx* FFT(int n,cplx *data,cplx *w){
    //n:共2^n个点 data:数据 w:FFT与IFFT相差一个共轭
    int N=(1<<n),N1=(N>>1),jMax,kMax,tmp; cplx *swapTmp;
    cplx *rslt=(cplx*)malloc(N*sizeof(cplx));
    for(int q=1;q<=n;q++){
        jMax=1<<(n-q); kMax=1<<(q-1);
        for(int j=0;j<jMax;j++) for(int k=0;k<kMax;k++){
            tmp=(j<<(q-1))+k;//这个值使用了4次，故提前计算以节省时间
            rslt[(j<<q)+k]=data[tmp]+data[tmp+N1];
            rslt[(j<<q)+k+kMax]=(data[tmp]-data[tmp+N1])*w[j<<(q-1)];
        }
        swapTmp=data; data=rslt; rslt=swapTmp;
    }
    return data;
}//2^n个点的FFT
cplx* bluestein(int N,cplx *data); //子函数较长，后置
int main(){
    int N;
    printf("N points, input N >>"); scanf("%d",&N);
    cplx *data=(cplx*)malloc(N*sizeof(cplx));
    double dataInput;
    printf("Input %d real numbers >>",N);
    for(int i=0;i<N;i++){//数据输入
        scanf("%lf",&dataInput);
        data[i]=cplx(dataInput,0);
    }
    data=bluestein(N,data);//计算
    printf("Ressult:\n");
    for(int i=0;i<N;i++)//数据输出
        printf("%d (%.4f)+(%.4f)i\n",i,data[i].real,data[i].imag);
    getchar();getchar();
    return 0;
}
```

```
cplx* bluestein(int N,cplx *data){
    int n=1; while((1<<n)<(3*N-2)) n++;
    //后续计算需要3N-2项，故找不小于3N-2的最小的2的幂，进行“补零”
    int N1=2*N-1,N2=(1<<n),N3=(N2>>1); //需要的三个常数
    cplx *w=(cplx*)malloc(N*sizeof(cplx));
    cplx *Nw=(cplx*)malloc(N3*sizeof(cplx));
    cplx *Niw=(cplx*)malloc(N3*sizeof(cplx));
    //w用于构造需要卷积的两个数列，Nw用于FFT，Niw用于IFFT
    double tmpw=2*acos(-1)/N,tmpNw=2*acos(-1)/N2;
    for(int i=0;i<N;i++) w[i]=cplx(cos(i*tmpw),sin(i*tmpw));
    for(int i=0;i<N3;i++){
        Nw[i]=cplx(cos(i*tmpNw),-sin(i*tmpNw));
        Niw[i]=bar(Nw[i]);
   }//以上是预处理部分,以下是计算部分

    cplx *A=(cplx*)malloc(N2*sizeof(cplx));
    cplx *B=(cplx*)malloc(N2*sizeof(cplx));
    cplx *C=(cplx*)malloc(N2*sizeof(cplx));
    for(int i=N;i<N2;i++) A[i]=B[i]=0; //先把后面的项置为0，注意N1>N
    for(int i=0;i<N;i++) A[i]=data[i]*w[(i*(i-1)/2)%N];
    for(int i=0;i<N1;i++) B[i]=bar(w[((N1-i-1)*(N1-i-2)/2)%N]);
    A=FFT(n,A,Nw); B=FFT(n,B,Nw);
    for(int i=0;i<N2;i++) C[i]=A[i]*B[i];
    C=FFT(n,C,Niw);//为了FFT与IFFT使用同一子程序，没有除N
    cplx tmpDiv=cplx(1.0/N2,0);
    for(int i=0;i<N2;i++) C[i]=C[i]*tmpDiv; //卷积完成

    cplx *rslt=(cplx*)malloc(N*sizeof(cplx));
    for(int i=0;i<N;i++) rslt[i]=w[(i*(i-1)/2)%N]*C[N1-i-1];
    return rslt;
}
```