含时薛定谔方程的数值解法

Author[a,1]

[a] 武汉大学，物理科学与技术学院

Keywords—薛定谔方程，切比雪夫多项式，紧束缚模型

Contents

1. 薛定谔方程

薛定谔方程在量子力学中占有十分重要的位置，根据量子力学的六个假定，正是薛定谔方程决定了物理体系随时间演变的规律[1]。

薛定谔方程

$$i\hbar\frac{\mathrm{d}}{\mathrm{d}t}|\varphi(t)\rangle = \hat{H}|\varphi(t)\rangle \tag{1}$$

由此可知，只要给出了初态 $|\varphi(0)\rangle$ 就足以决定此后任意时刻的态 $|\varphi(t)\rangle$。因此在物理体系演变过程中发没有任何不确定性。

1.1. 原子单位制

在数值计算中，一些常数，比如 $\hbar, m_e, \varepsilon$ 等，不仅使得公式非常繁琐，并且在实际的计算当中增加了复杂度，同时由于这些常数往往很大或者很小，大大降低了计算精度，容易出现数值溢出。为此，人们往往使用原子单位制重写方程：[2]

原子单位制	
质量	$m_e = 9.1094\times10^{-31}kg$
电荷	$e = 1.6022\times10^{-19}C$
角动量	$\hbar = 1.0546\times10^{-43}J\cdot s$
介电常数	$4\pi\varepsilon_0 = 1.1127\times10^{-10}F\cdot m^{-1}$
长度	$a_0 = \frac{4\pi\varepsilon_0\hbar^2}{m_e e^2} = 5.2918\times10^{-11}m$
能量	$Hartree = \frac{m_e e^2}{(4\pi\varepsilon_0)^2\hbar} = 4.3597\times10^{-18}J$

在合适的单位制下（事实上此处也并非原子单位制，而是令 $\frac{\hbar^2}{2m}=0$），二维的薛定谔方程改写为：

$$\begin{aligned} i\frac{\mathrm{d}}{\mathrm{d}t}|\varphi(t)\rangle &= \hat{H}|\varphi(t)\rangle \\ \hat{H} &= -\frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial y^2} + V(\mathbf{r}) \end{aligned} \tag{2}$$

2. 数值解法

2.1. 离散化

显然，数值求解薛定谔方程之前需要对方程2进行离散化，设 δ 为网格的间距，用 $\psi_{l,k}(t)$ 近似地表示 $\varphi(l\delta, k\delta, t)$。用最简单的差分近似替代关于 x,y 的导数：

$$\frac{\partial^2\psi(x,y,t)}{\partial x^2} \approx \frac{\psi(x+\delta,y,t) - 2\psi(x,y,t) + \psi(x-\delta,y,t)}{\delta^2} \tag{3}$$

$$= \delta^{-2}[\psi_{l+1,k} - 2\psi_{l,k} + \psi_{l+1,k}] \tag{4}$$

一维的离散薛定谔方程可以写成

$$\frac{\partial\psi(t)}{\partial t} = -i\{-\delta^{-2}[\psi_{l+1} + \psi_{l-1}] + \upsilon_l\}$$

哈密顿量可以写为矩阵的形式

$$\hat{H} = \begin{bmatrix} \upsilon_1 & -\delta^2 & \dots & 0 \\ -\delta^2 & \upsilon_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \upsilon_l \end{bmatrix}$$

同理, 二维的薛定谔方程可以写为

$$\frac{\partial\psi(t)}{\partial t} = -i\{-\delta^{-2}[\psi_{l+1,k} + \psi_{l-1,k} + \psi_{l,k+1} + \psi_{l,k-1}] + v_{l,k}\} \quad (5)$$

其中 $v_{l,k} = V(x,y) - 4$。如果我们将 ψ_{lk} 按照先排列 x 方向后 y 方向的方式堆积成一个列向量，即：

$$\psi_{lk} = \psi_n, n = (k-1)L + l \quad (6)$$

, 那么 $\hat{H}$ 仍然可以写成一个矩阵：

$$\hat{H} = \begin{bmatrix} v_{11} & -\delta^2 & \cdots & -\delta^2 & \cdots & 0 \\ -\delta^2 & v_{12} & \cdots & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ -\delta^2 & 0 & \cdots & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \cdots & -\delta^2 & v_{LK} \end{bmatrix}$$

方程5也可以理解为一个粒子在 $N = L \times K$ 的二维网格中传播，这种理解对于处理更加复杂的哈密顿量是十分重要的。

$$\begin{aligned} \hat{H} &= -t\sum_n^N (c_{n+1}^\dagger c_n + c_n^\dagger c_{n+1} + c_{n+L}^\dagger c_n + c_n^\dagger c_{n+L} - W\sum_n^N \epsilon_n c_n^\dagger c_n) \\ |\varphi(t)\rangle &= \sum_{n=1}^{N+1} \psi_n(t) c_n^\dagger |0\rangle \end{aligned} \quad (7)$$

$c^\dagger, c$ 是产生湮灭算符。对比方程5和7，不难发现 $t = \delta^{-2}, W\epsilon_n = v_n$。

2.2. 矩阵指数

与普通的微分方程类似，矩阵微分方程有解析解 [3]：

$$\begin{aligned} i\frac{\mathrm{d}\hat{U}(t)}{\mathrm{d}t} &= \hat{H}(t)\hat{U} \\ \hat{U}(t) &= e^{-i\hat{H}t}\hat{U}(0) \end{aligned} \quad (8)$$

问题的关键在于求出 $e^{-i\hat{H}t}$。矩阵指数由 Taylor 级数定义：

$$e^{xH} = \sum_{n=0}^{\infty} \frac{x^n}{n!} H^n \quad (9)$$

尽管这样的展开在数学上是非常重要的，但是在实际计算中没有意义，一方面存储矩阵需要占用大量的内存，对计算资源造成巨大的浪费；另一方面，Taylor 展开无法满足波函数模方守恒的要求：

$$\begin{aligned} &\langle\varphi(0)|\varphi(0)\rangle = \langle\varphi(t)|\varphi(t)\rangle \\ &\neq \langle\varphi(0)|(I + itH - \frac{t^2}{2}H + \cdots)(I - itH - \frac{t^2}{2}H + \cdots)|\varphi(0)\rangle \end{aligned} \quad (10)$$

为此我们引入切比雪夫多项式方法计算 $e^{-i\hat{H}t}$

2.3. 切比雪夫多项式

为了演化波函数，需要对时间演化算子进行数值展开，由于哈密顿量是稀疏的，所以可以方便地使用 Chebyshev 多项式进行展开，这种方法对求解含时薛定谔方程是无条件稳定的。假设 $x \in [-1,1]$，

$$e^{-izx} = J_0(z) + 2\sum_{m=1}^{\infty}(-i)^m J_m(z) T_m(x) \quad (11)$$

其中 J_m 是 m 阶贝塞尔函数，T_m 是第一类切比雪夫多项式。$T_m(x)$ 可以由递归关系求解：

$$T_{m+1}(x) = 2xT_m - T_{m-1} \quad (12)$$

为了利用切比雪夫多项式方法，我们需要将 $\hat{H}$ 缩放为 $\tilde{H} = \hat{H}/\|H\|$，使 $\tilde{H}$ 的特征值分布在 $[-1,1]$。

$$|\varphi(t)\rangle = \{J_0(\tau) + 2\sum_{m=1}^{\infty}(-i)^m J_m(\tau)\hat{T}_m(\tilde{H})\}|\varphi(0)\rangle \quad (13)$$

$\tau = t \cdot \|H\|$。在实际计算中不需要存储 $\hat{T}_m$，而是由递归关系直接计算波函数：

$$\hat{T}_{m+1}(\tilde{H})\varphi(0) = 2\tilde{H}\hat{T}_m(\tilde{H})\varphi(0) - \hat{T}_{m-1}(\tilde{H})\varphi(0) \quad (14)$$

除了时间演化算子 $e^{-it\hat{H}}$ 外，其他算子也可以使用类似方法，展开为切比雪夫多项式的级数。

$$f(x) = \frac{1}{2}c_0 T_0(x) + \sum_m c_m T_m(x) \quad (15)$$

系数定义为

$$c_m = \frac{2}{\pi}\int_{-1}^{1} \frac{\mathrm{d}x}{\sqrt{1-x^2}} f(x) T_m(x) \quad (16)$$

令 $x = \cos\theta$，并带入上式，可以得到：

$$\begin{aligned} c_m &= \frac{2}{\pi}\int_0^{\pi} f(\cos\theta)\cos(m\theta)\,\mathrm{d}\theta \\ &= Re[\frac{2}{\pi}\sum_{n=0}^{N-1} f(\cos\frac{2\pi n}{N}) e^{i\frac{2\pi n}{N}m}] \end{aligned} \quad (17)$$

因此可以使用快速傅里叶变换计算出 c_k。以 Fermi-Dirac 算符为例：

$$\begin{aligned} f(\hat{H}) &= \frac{ze^{-\beta\hat{H}}}{1 + ze^{-\beta\hat{H}}}, \\ \text{where} \quad \beta &= \frac{1}{k_B T}, z = e^{\beta\mu} \end{aligned} \quad (18)$$

我们定义 $\tilde{\beta} = \beta \cdot \|\hat{H}\|$。根据前面的讨论：

$$f(\tilde{H}) = \sum_{m=0}^{\infty} c_m T_m(\tilde{H}) \quad (19)$$

3. 高斯波包

3.1. Theory

本节我们将用上面提到的方法计算一个高斯波包通过波导时波函数随时间的演化。初始的波函数设置为:

$$\Psi(x,y,t_0) = Ae^{-\frac{x^2}{4\sigma_x^2} - \frac{y^2}{4\sigma_y^2} + ik_0 x} \quad (20)$$

为了合理地设置网格数量和演化时间等参数，我们有必要讨论一个波包自由传播时的解析理论。为简单起见，我们首先考虑一维的情形：

$$\Psi(x,0) = Ae^{-\frac{x^2}{4\sigma^2} + ik_0 x} \quad (21)$$

由归一化求得 $A = (\frac{1}{2\pi\sigma^2})^{\frac{1}{4}}$ 叠加原理告诉我们，自由粒子的波函数时各种波矢的平面波的叠加：

$$\Psi(x,t) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} g(k) e^{i(kx - \omega t)} dk \quad (22)$$

我们已知 t=0 时刻的波函数，于是可以由傅里叶变换求出 $g(k)$:

$$\begin{aligned} g(k) &= \frac{1}{\sqrt{2\pi}} \int \Psi(x,0) e^{-ikx} \, \mathrm{d}dx \\ &\frac{A}{\sqrt{2\pi}} \int e^{-\frac{x^2}{4\sigma^2}} e^{-i(k-k_0)x} \mathrm{d}x \\ &\frac{A}{\sqrt{2\pi}} \int e^{-\frac{1}{(2\sigma)^2}(x+2i(k-k_0)\sigma^2)^2-(k-k_0)^2\sigma^2} \, \mathrm{d}x \end{aligned} \tag{23}$$

利用留数的性质可以证明, 若满足 $-\frac{\pi}{4} < Arg(\alpha) < \frac{\pi}{4}$:

$$\begin{aligned} I(\alpha,\beta) &= \int_{-\infty}^{\infty} e^{-\alpha^2(\xi+\beta)^2} \, \mathrm{d}\xi \\ &= I(\alpha,0) = \frac{\sqrt{\pi}}{\alpha} \end{aligned} \tag{24}$$

于是:

$$\Psi(x,t) = \frac{\sqrt{2\sigma}}{(2\pi)^{3/4}} \int_{-\infty}^{\infty} e^{-(k-k_0)^2\sigma^2} e^{ikx} \, \mathrm{d}k \tag{25}$$

不难证明 t 时刻波包的形状仍然为高斯波包。再来计算 t 时刻波函数的模方:

$$|\Psi(x,t)|^2 = \sqrt{\frac{2}{\pi a^2}} \frac{1}{\sqrt{1+\frac{4\hbar^2 t^2}{m^2 a^4}}} \exp\{-\frac{2a^2(x-\frac{\hbar k_0 t}{m})^2}{a^4+\frac{4\hbar^2 t^2}{m^2}}\} \tag{26}$$

$a = 2\sigma$。此时波包的最大值位于 x_M:

$$x_M = \frac{\hbar k_0}{m} t \tag{27}$$

求出波包的群速度:

$$V_G = \frac{\mathrm{d}x_M}{\mathrm{d}t} = \frac{\hbar k_0}{m} \tag{28}$$

二维波包穿过狭缝时衍射波的角宽度近似满足:

$$2\theta \approx \frac{2\lambda}{\Delta y} \tag{29}$$

3.2. Result

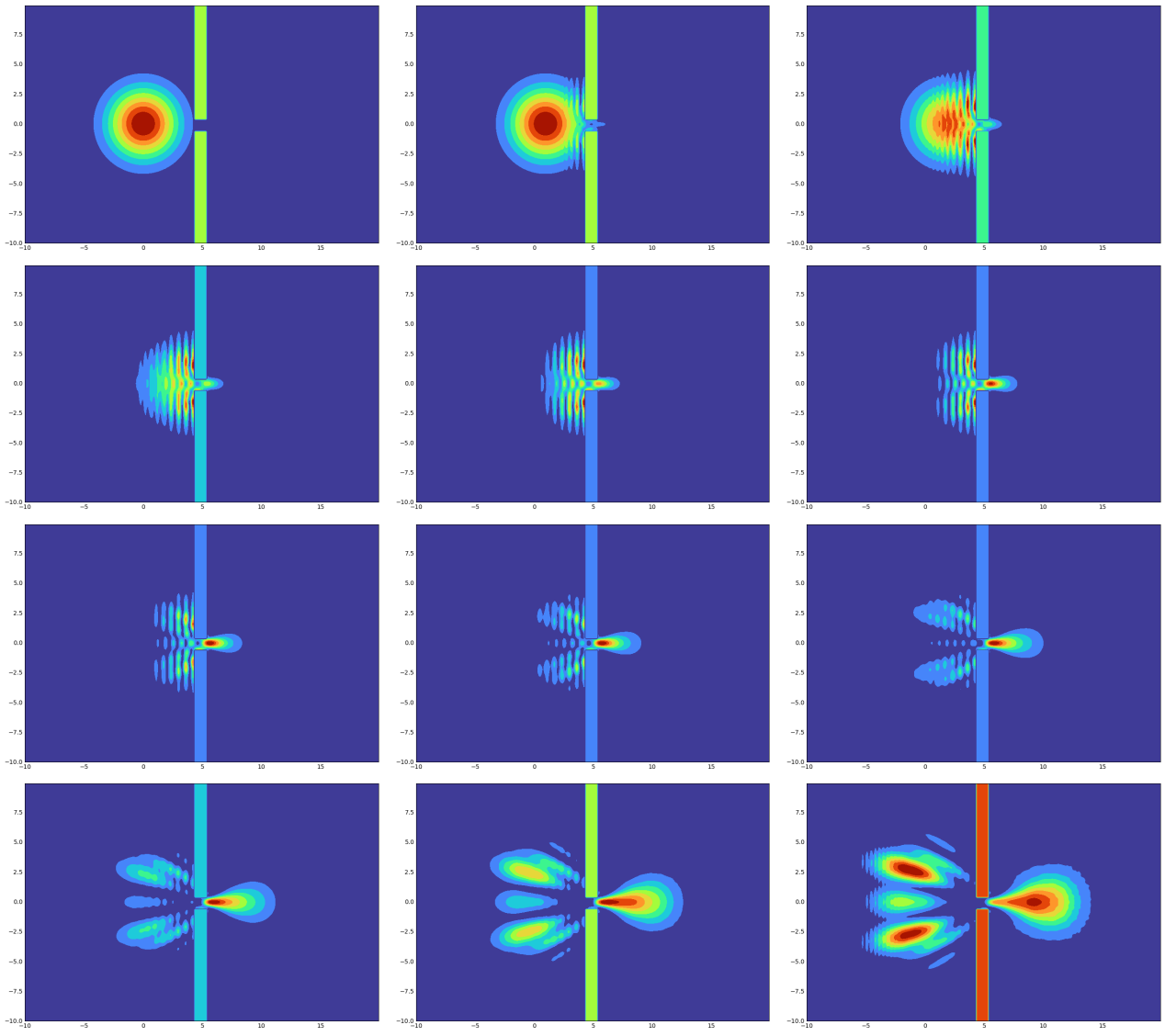

Figure 1. $\Delta y = 20\delta, t \in [0, 1100]$

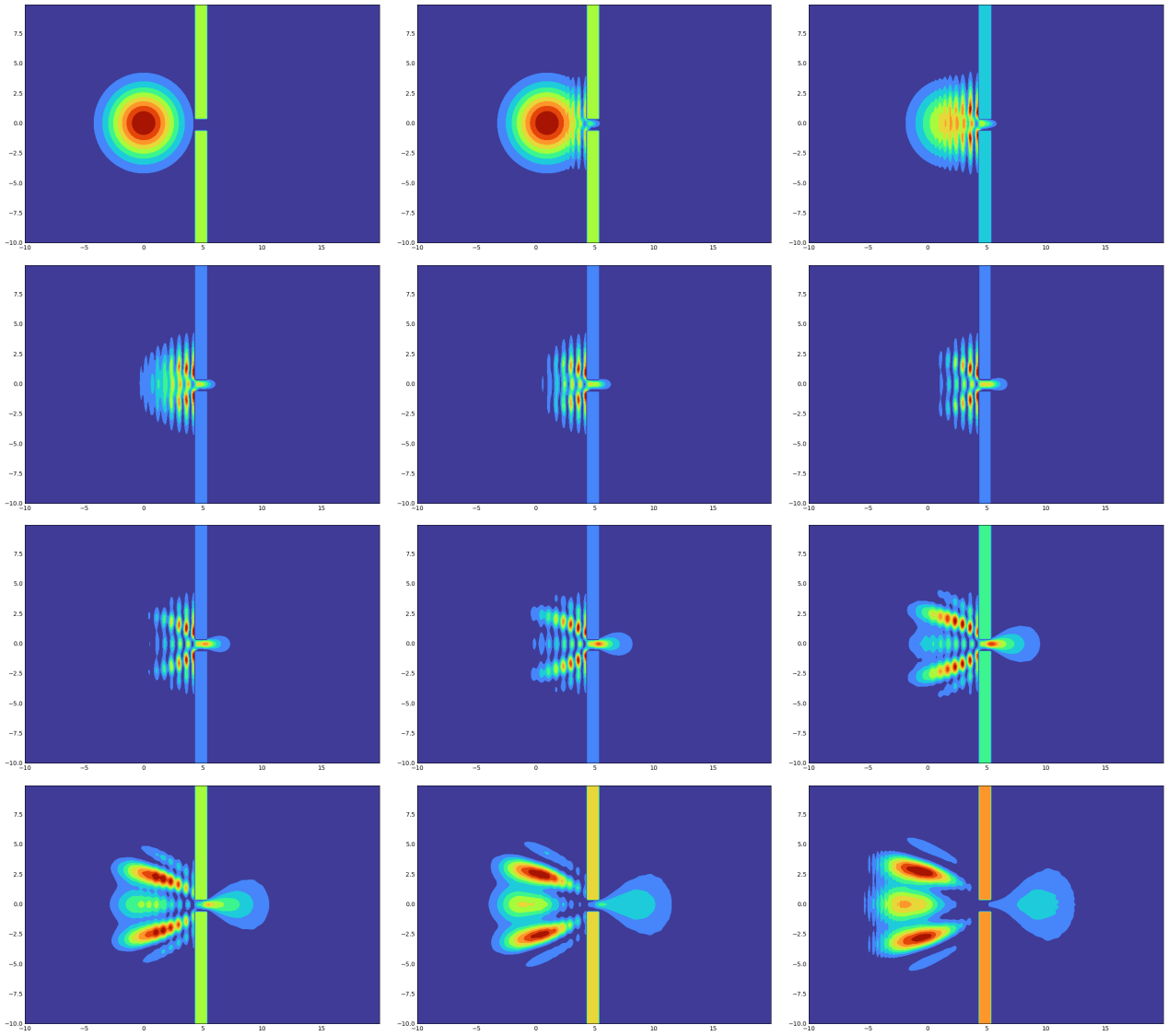

Figure 2. $\Delta y = 10\delta, t \in [0, 1100]$

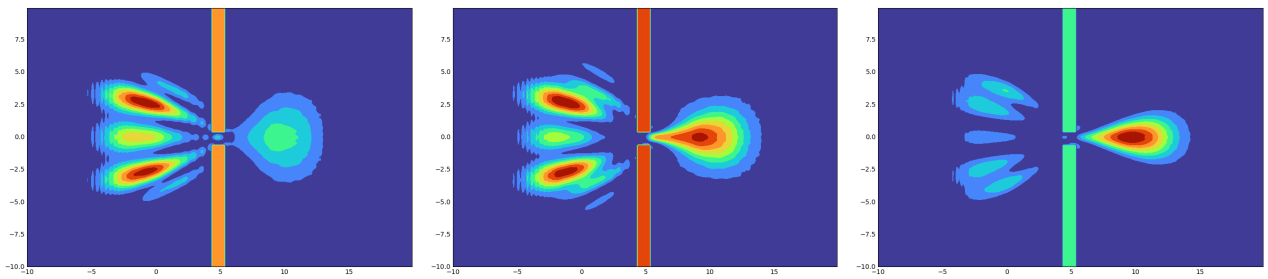

Figure 3. $\Delta y = 16\delta, 20\delta, 30\delta$,t=1100

4. Code


```cpp
#include<iostream>
#include<cmath>
#include<vector>
#include <bitset>
#include <string>//to_string()
#include <algorithm>//reverse()
#include <fstream>
#define PI 3.14159
#define NN 32768
#define Wi 15
using namespace std;
class Wave
{
    public:
        float sigma,delta,k;
        float x0=0,y0=0;
        int istart=0;
        int Nx,Ny,Nh;
        int length=5,width=5;
        //length和width分别是势垒的厚度和宽度的一半，单位是delta。需要注意，取值应该是python版本的一半。
        float Vmax,T;
        float *X;
        float *Y;
        float *potential;
        float **psi;


        int initialize(float x_min,float x_max,float y_min,float y_max,float in_delta,float k,float Vmax,float Ek,int in_istart);
        int operate(float vector[]);
        int TimeInt(float An_r[],float An_i[],int m);

};
int Wave::initialize(float x_min,float x_max,float y_min,float y_max,float in_delta,float in_k,float in_Vmax,float Ek,int in_istart)
{   T=Ek;
    istart=in_istart;
    Vmax=in_Vmax;
    delta=in_delta;
    sigma=20*delta;
    k=in_k;
    Nx=int((x_max-x_min)/delta);
    Ny=int((y_max-y_min)/delta);
    cout<<"grid number"<<Nx<<'|'<<Ny<<endl;
    Nh=Nx*Ny;
    X =new float[Nx];
    Y =new float[Ny];
    for(int i=0;i<Nx;i++)
    {
        X[i]=x_min+i*delta;
        //cout<<X[i]<<endl;
    }
    for(int j=0;j<Ny;j++)
    {
        Y[j]=y_min+j*delta;
    }
    psi=new float*[2];
    psi[0]=new float[Nh]();
    psi[1]=new float[Nh]();
    if(istart==0){
        for(int i=0;i<Ny;i++)
    {
        for(int j=0;j<Nx;j++)
        {
            psi[0][i*Nx+j]=exp(-pow((X[j]-x0),2)/(4*pow(sigma,2))-pow((Y[i]-y0),2)/(4*pow(sigma,2)))*cos(k*X[j]);
            //psi[0][i*Nx+j]=1;
           //cout<<exp(-pow(0,2)/(4*pow(sigma,2))-pow(0,2)/(4*pow(sigma,2)))<<endl;
            psi[1][i*Nx+j]=exp(-pow((X[j]-x0),2)/(4*pow(sigma,2))-pow((Y[i]-y0),2)/(4*pow(sigma,2)))*sin(k*X[j]);
        }
    }
    }
    else{
        ifstream infile;
        infile.open("Psi_r.dat");
        for(int i=0;i<Nh;i++)
```

```
        {
        infile>>psi[0][i];
        }
        infile.close();
        infile.open("Psi_i.dat");
        for(int i=0;i<Nh;i++)
        {
        infile>>psi[1][i];
        }
        infile.close();
    }

    //cout<<X[100]<<endl;
    //cout<<exp(-pow((X[100]-x0),2)/(4*pow(sigma,2))-pow((Y[100]-y0),2)/(4*pow(sigma,2)))*cos(k*X
    [100])<<endl;
    potential= new float[Nh]();
    int x1=Nx/2-length;
    int x2=Nx/2+length;
    int y1=Ny/2-width;
    int y2=Ny/2+width;
    for(int j=x1;j<x2;j++)
    {
        for(int i=0;i<Ny;i++)
        {
            potential[i*Nx+j]=Vmax;
        }
        for(int i=y1;i<y2;i++)
        {
            potential[i*Nx+j]=0;
        }
    }
    delete[] X;
    delete[] Y;
    return 0;
}
int Wave::operate(float vector[])
{
    float *Temp=new float[Nh]();
    Temp[0]=(potential[0]+4*T)*vector[0]-T*(vector[1]+vector[Nx]);
    for(int i=1;i<Nx;i++)
    {
        Temp[i]=(potential[i]+4*T)*vector[i]-T*(vector[i-1]+vector[i+1]+vector[i+Nx]);
    }
    for(int i=Nx;i<Nh-Nx;i++)
    {
        Temp[i]=(potential[i]+4*T)*vector[i]-T*(vector[i-1]+vector[i+1]+vector[i+Nx]+vector[i-Nx]);
    }
    for(int i=Nh-Nx;i<Nh-1;i++)
    {
        Temp[i]=(potential[i]+4*T)*vector[i]-T*(vector[i-1]+vector[i+1]+vector[i-Nx]);
    }
    Temp[Nh-1]=(potential[Nh-1]+4*T)*vector[Nh-1]-T*(vector[Nh-2]+vector[Nh-1-Nx]);
    for(int i=0;i<Nh;i++)
    {
        vector[i]=Temp[i];
    }
    delete[] Temp;
    return 0;
}
int Wave::TimeInt(float An_r[],float An_i[],int m)
{
    float **Temp=new float*[2];
    Temp[0]=new float[Nh](),Temp[1]=new float[Nh]();
    float **Temp0=new float*[2];
    Temp0[0]=new float[Nh](),Temp0[1]=new float[Nh]();
    float **Temp1=new float*[2];
    Temp1[0]=new float[Nh](),Temp1[1]=new float[Nh]();
    for(int i=0;i<Nh;i++)
    {
        Temp0[0][i]=psi[0][i];
        Temp0[1][i]=psi[1][i];
    }
    operate(psi[0]);
    operate(psi[1]);
    for(int i=0;i<Nh;i++)
    {
        Temp1[0][i]=psi[0][i];
        Temp1[1][i]=psi[1][i];
        psi[0][i]=0.5*(An_r[0]*Temp0[0][i]-An_i[0]*Temp0[1][i])+(An_r[1]*Temp1[0][i]-An_i[1]*Temp1
    [1][i]);
        psi[1][i]=0.5*(An_i[0]*Temp0[0][i]+An_r[0]*Temp0[1][i])+(An_r[1]*Temp1[1][i]+An_i[1]*Temp1
```

```
    [0][i]);
    }
    float *temp_r=new float[Nh];
    float *temp_i=new float[Nh];
    for(int j=2;j<m;j++)
    {
        for(int i=0;i<Nh;i++)
        {
            temp_r[i]=Temp1[0][i];
            temp_i[i]=Temp1[1][i];
        }
        operate(Temp1[0]);
        operate(Temp1[1]);
        for(int i=0;i<Nh;i++)
        {
            Temp[0][i]=2*Temp1[0][i]-Temp0[0][i];
            Temp[1][i]=2*Temp1[1][i]-Temp0[1][i];
            psi[0][i]=psi[0][i]+(An_r[j]*Temp[0][i]-An_i[j]*Temp[1][i]);
            psi[1][i]=psi[1][i]+(An_r[j]*Temp[1][i]+An_i[j]*Temp[0][i]);
            Temp1[0][i]=Temp[0][i];
            Temp1[1][i]=Temp[1][i];
            Temp0[0][i]=temp_r[i];
            Temp0[1][i]=temp_i[i];
        }
        if(j%200==0)
        {
            float fm=float(m);
            cout<<'#';
        }

    }
    delete [] Temp0;
    delete[] Temp1;
    delete[] Temp;
    return 0;

}
float DataRev(float data[])
{
    float data_rev[NN];
    int width=int(log(NN)/log(2));
    for(unsigned int i=0;i<NN;i++)
    {
        bitset<Wi> bit(i);//i转二进制数
        string iStr=bit.to_string();//二进制数转字符串
        reverse(iStr.begin(), iStr.end());//字符串倒序
        int j = stoi(iStr, nullptr, 2);//字符串转十进制数
        data_rev[i]=data[j];
        //cout<<iStr<<endl;
    }
    for(int i=0;i<NN;i++)
    {
        data[i]=data_rev[i];
    }
    return 0;
}
float Coeff_r(int N,int k,int isign)
{
    return cos(isign*2*3.1415926*k/N);
}
float Coeff_i(int N,int k,int isign)
{
    return sin(isign*2*3.1415926*k/N);
}
int FFT(float data_r[],float data_i[], int isign,int N,int Width)
{   cout<<"begin DataRev"<<endl;
    DataRev(data_r);
    DataRev(data_i);
    for(int i=0;i<Width;i++)
    {
        for(int j=0;j<N;j+=pow(2,i+1))
        {
            for(int k=0;k<pow(2,i);k++)
            {
                int x=k+j;
                int y=x+pow(2,i);
                float W_r = Coeff_r(pow(2,i+1),k,isign),W_i = Coeff_i(pow(2,i+1),k,isign);
                float temp_r=data_r[y],temp_i=data_i[y];
                data_r[y]=data_r[x]-W_r*temp_r+W_i*temp_i;
                data_i[y]=data_i[x]-W_r*temp_i-W_i*temp_r;
                data_r[x]=data_r[x]+W_r*temp_r-W_i*temp_i;
```

```
                data_i[x]=data_i[x]+W_r*temp_i+W_i*temp_r;
            }

        }
    }
    if(isign==1)
    {
        for(int i=0;i<N;i++)
        {
            data_r[i]=2*data_r[i]/N;
            data_i[i]=2*data_i[i]/N;
        }
    }
    return 0;
}
int Bessel(float data_r[],float data_i[],float z,int N,int M)
{
    for(int i=0;i<N;i++)
    {
        data_r[i]=cos(-z*cos(2*PI*i/N));
        data_i[i]=sin(-z*cos(2*PI*i/N));
    }
    cout<<"begin FFT"<<endl;
    FFT(data_r,data_i,1,N,M);
    cout<<"FFT down"<<endl;
    /*
    for(int i=0;i<N;i=i+4)

    {   data_r[i]=2*data_i[i];
        data_r[i+1]=2*data_i[i+1];
        data_r[i+2]=-2*data_r[i+2];
        data_r[i+3]=-2*data_i[i+3];
    }
    */
    return 0;
}

int main()
{
    cout<<"initialize:...";
    Wave Psi;
    Psi.initialize(-10,20,-10,10,0.1,5,0.1,0.1,0);
    cout<<"down"<<endl;
    cout<<"computing coeffient an:...";
    float An_r[NN];
    float An_i[NN];
    int m=5000,timesteps=1100;
    Bessel(An_r,An_i,timesteps,NN,Wi);
    cout<<"down"<<endl;
    cout<<"begin TimeInt:";
    //time integral
    Psi.TimeInt(An_r,An_i,m);
    cout<<endl<<"down"<<endl;
    cout<<"write dat"<<endl;
    ofstream outfile;
    outfile.open("Psi_r.dat");
    for(int i=0;i<Psi.Nh;i++)
    {

        outfile<<Psi.psi[0][i]<<endl;
    }
    outfile.close();
    outfile.open("Psi_i.dat");
    for(int i=0;i<Psi.Nh;i++)
    {
        outfile<<Psi.psi[1][i]<<endl;
    }
    outfile.close();
    cout<<"completed"<<endl;

}
```

Code 1. Propagation

```
import numpy  as np
from sys import version
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
```

```
import os
def grid(min,max,delta):
    #生成网格,从min到max, 间距为delta
    line=np.arange(min,max,delta)
    return line
def gragh(X,Y,V):
    #绘制波函数的3D图像
    fig = plt.figure()
    ax = Axes3D(fig)
    fig.add_axes(ax)
    print(len(X),len(Y),len(V))
    Z=np.zeros((len(Y),len(X)),float)
    for i in range(0,len(Y)):
        for j in range(0,len(X)):
            Z[i][j]=V[i*len(X)+j]
    X, Y = np.meshgrid(X, Y)
    ax.plot_surface(X, Y, Z,rstride=1, cstride=1, cmap=cm.viridis)
    plt.show()
def level(X,Y,V,psi):
#波函数的俯视图
    plt.style.use('_mpl-gallery-nogrid')
    Z1=np.zeros((len(Y),len(X)),float)
    Z2=np.zeros((len(Y),len(X)),float)
    #y在前, x在后的二维数组
    for i in range(0,len(Y)):
        for j in range(0,len(X)):
            Z1[i][j]=psi[i*len(X)+j]
            Z2[i][j]=V[i*len(X)+j]
    level1 = np.linspace(Z1.min(), Z1.max(),10)
    fig, ax = plt.subplots(figsize=(9, 6))
    ax.contourf(X, Y, Z1,  levels=level1,cmap=cm.turbo)
    ax.contourf(X, Y, Z2,  levels=level1,cmap=cm.turbo)
    plt.show()
def potential(X,Y,Vmax,length,width,d):
    #生成一个中间带有小孔的势垒
    Ny=len(Y)
    Nx=len(X)
    Nh=Nx*Ny
    V=np.zeros((Nh),float)
    mx=int(len(X)/2)
    my=int(len(Y)/2)
    #mx,my是整个网格的中间位置, length和width对应小孔在x和y方向的尺度
    x1=int((X[mx]-X[0]-length/2)/d)
    x2=int((X[mx]-X[0]+length/2)/d)
    y1=int((Y[my]-Y[0]-width/2)/d)
    y2=int((Y[my]-Y[0]+width/2)/d)
    for j in range(x1,x2):
        for i in range(0,Ny):
            V[i*Nx+j]=Vmax
        for i in range(y1,y2):
            V[i*Nx+j]=0.
    return V
if __name__=='__main__':
    '''
    data_r = pd.read_csv('TDSE/psi_r.dat', sep='\t', dtype=float)
    data_i = pd.read_csv('TDSE/psi_i.dat', sep='\t', dtype=float)
    data_r = np.fromfile('TDSE/psi_r.dat', dtype=np.float32)
    data_i = np.fromfile('TDSE/psi_i.dat', dtype=np.float32)

    '''
    path=os.path.join(os.getcwd(),'TDSE/psi_r.dat')
    f=open(path)
    print("read file")
    line=f.readline().strip()
    data_r=[]
    print(len(data_r))
    i=0
    while line:
        data_r.append(line)
        line=f.readline().strip()
        i+=1
    print(i)
    f.close()
    f=open('TDSE/psi_i.dat')
    print("read file")
    line=f.readline().strip()
    data_i=[]
    i=0
    while line:
        data_i.append(line)
        line=f.readline().strip()
```

```
        i+=1
    print(i)
    f.close()

    psi_r=[]
    psi_i=[]
    psi2=[]
    print(len(data_i))
    for j in range(0,len(data_i)):
            psi_r.append(float(data_r[j]))
            psi_i.append(float(data_i[j]))
            psi2.append(psi_r[j]**2+psi_i[j]**2)
    d=0.1
    X=grid(-10,20,d)
    Y=grid(-10,10,d)
    print("prepare figure")
    V=potential(X,Y,0.5,10*d,10*d,d)
    level(X,Y,V,psi2)
    #gragh(X,Y,psi2)
```

Code 2. Plot

5. Appendix：Tight-Binding propagation method and Application

紧束缚（Tight-Binding，TB）方法是一种在凝聚态物理和量子化学中常用的半经验方法，此方法利用原子轨道构建系统的哈密顿量，利用对角化或非对角化方法研究电子结构。[4] 不同于第一性原理计算方法，紧束缚方法中的哈密顿矩阵由经验参数给出，避免了耗时间较长的自洽场计算 [2]。因此紧束缚方法可以处理较大的体系。利用非对角化的 TBPM（Tight-Binding propagation method）方法，可以处理含有数十亿个原子轨道的庞大系统。

5.1. Methodology

5.1.1. Tight-binding models

一个包含 n 个原子轨道的非周期性系统的哈密顿量可以写成如下形式：

$$\hat{H} = \sum_i \epsilon_i c_i^\dagger c_i - \sum_{i\neq j} t_{ij} c_i^\dagger c_j \tag{30}$$

可以写成如下紧凑的形式：

$$\begin{aligned} \hat{H} &= \mathbf{c}^\dagger \mathbf{H} \mathbf{c} \\ \mathbb{c}^\dagger &= [c_1^\dagger, c_2^\dagger, \cdots, c_n^\dagger] \\ H_{ij} &= \epsilon_i \delta_{ij} - t_{ij}(1-\delta_{ij}) \end{aligned} \tag{31}$$

其中 ϵ_i 称为格位积分（on-site integral），t_{ij} 为跃迁积分。$c^\dagger, c$ 为产生湮灭算符。格位积分和跃迁积分的定义为：

$$\begin{aligned} \epsilon_i &= \int \psi_i^*(r) \hat{h_0} \psi_i(r)\, \mathrm{d}r \\ t_{ij} &= -\int \psi_i^*(r) \hat{h_0} \psi_j(r)\, \mathrm{d}r \end{aligned} \tag{32}$$

$\hat{h_0}$ 是单粒子的哈密顿量：

$$\hat{h_0} = -\frac{\hbar^2}{2m}\nabla^2 + V(r) \tag{33}$$

ψ_i 是单个原子的原子轨道。在实际的计算中，往往采用 Wannier 函数。对具有周期性的体系，利用布洛赫定理，我们可以只关注第一个单胞。为此引入一个额外的指标 R:

$$\psi_{iR}(r) = \psi_i(r-R) \tag{34}$$

通过傅里叶变换，定义布洛赫波函数和产生湮灭算符：

$$\begin{aligned} \chi_{ik}(r) &= \frac{1}{N}\sum_R e^{ik\cdot(R+\tau_i)} \psi_{iR}(r) \\ c_i^\dagger(k) &= \frac{1}{N}\sum_R e^{ik\cdot(R+\tau_i)} c_i^\dagger(R) \\ c_i(k) &= \frac{1}{N}\sum_R e^{-ik\cdot(R+\tau_i)} c_i(R) \end{aligned} \tag{35}$$

其中 N 表示单胞的数目。Hamiltonian 可以写作：

$$\hat{H} = N\sum_k \{\sum_{i\in uc} \epsilon_i c_i^\dagger(k) c_i(k) - \sum_{R\neq 0, i\neq j} t_{ij}(R) e^{ik\cdot(R+\tau_i-\tau_j)} c_i^\dagger(k) c_j(k)\} \tag{36}$$

5.1.2. Tight-binding propagation method

对哈密顿矩阵进行对角化，可以精确地得到特征值和特征态，最终得到所有物理量。然而，精确对角化的内存和 CPU 时间成本随着模型规模 N 的增长分别为 $O(N^2)$ 和 $O(N^3)$，对于大型的模型来说，这种方法是不可行的。相反，TBPM 方法使用一种完全不同的方法处理特征值问题，使得内存和时间消耗与 N 呈线性关系。在 TBPM 方法中，初始波函数由一组随机生成的状态组成，然后按如下方程演化：

$$|\varphi(t)\rangle = e^{-i\hat{H}t}|\varphi(0)\rangle \tag{37}$$

关联函数中包含哈密顿量的特征，只要演化时间足够长，演化步长足够小，就可以准确捕捉到哈密顿量的全部特性。最后对关联函数求平均值并进行分析，得到物理量。以 DOS 的关联函数为例，

$$C_{DOS}(t) = \langle\varphi(0)|\varphi(t)\rangle \tag{38}$$

可以证明它和特征值有以下关系：

$$\langle\varphi(0)|\varphi(t)\rangle = \sum_{ijk} U_{kj} U_{ij}^* a_i a_k^* e^{-\epsilon_j k} \tag{39}$$

ϵ_j 是第 j 个特征值。$U_k j$ 是第 j 个特征值的第 k 个分量。初始波函数写作：

$$|\varphi(0)\rangle = \sum_i a_i |\psi_i\rangle \tag{40}$$

其中 a 是符合归一化 $\sum_i |a_i|^2 = 1$ 的随机复数。ψ_i 是基矢量。关联函数可以看作是频率为 ϵ_i 的波的线性组合。利用逆傅里叶变换，可以确定特征值和 DOS。

为了演化波函数，需要对时间演化算子进行数值展开，由于 TB 的哈密顿量是稀疏的，所以可以方便地使用 Chebyshev 多项式进行展开，这种方法对求解含时薛定谔方程是无条件稳定的。假设 $x \in [-1, 1]$，

$$e^{-izx} = J_0(z) + 2\sum_{m=1}^{\infty} (-i)^m J_m(z) T_m(x) \tag{41}$$

其中 J_m 是 m 阶贝塞尔函数，T_m 是第一类切比雪夫多项式。$T_m(x)$ 可以由递归关系求解：

$$T_{m+1}(x) = 2xT_m - T_{m-1} \tag{42}$$

为了利用切比雪夫多项式方法，我们需要将 $\hat{H}$ 缩放为 $\tilde{H} = \hat{H}/\|H\|$，使 $\tilde{H}$ 的特征值分布在 $[-1, 1]$。

$$|\varphi(t)\rangle = \{J_0(\tau) + 2\sum_{m=1}^{\infty} (-i)^m J_m(\tau) \hat{T}_m(\tilde{H})\}|\varphi(0)\rangle \tag{43}$$

$\tau = t\cdot\|H\|$。在实际计算中不需要存储 $\hat{T}_m$，而是由递归关系直接计算波函数：

$$\hat{T}_{m+1}(\tilde{H})\varphi(0) = 2\tilde{H}\hat{T}_m(\tilde{H})\varphi(0) - \hat{T}_{m-1}(\tilde{H})\varphi(0) \tag{44}$$

除了时间演化算子 $e^{-it\hat{H}}$ 外，其他算子也可以使用类似方法，展开为切比雪夫多项式的级数。

$$f(x) = \frac{1}{2}c_0 T_0(x) + \sum_m c_m T_m(x) \tag{45}$$

系数定义为

$$c_m = \frac{2}{\pi}\int_{-1}^{1} \frac{\mathrm{d}x}{\sqrt{1-x^2}} f(x) T_m(x) \tag{46}$$

令 $x = \cos\theta$，并带入上式，可以得到：

$$\begin{aligned} c_m &= \frac{2}{\pi}\int_0^{\pi} f(\cos\theta)\cos(m\theta)\,\mathrm{d}\theta \\ &= Re[\frac{2}{\pi}\sum_{n=0}^{N-1} f(\cos\frac{2\pi n}{N}) e^{i\frac{2\pi n}{N}m}] \end{aligned} \tag{47}$$

因此可以使用快速傅里叶变换计算出 c_k。以 Fermi-Dirac 算符为例：

$$\begin{aligned} f(\hat{H}) &= \frac{ze^{-\beta\hat{H}}}{1+ze^{-\beta\hat{H}}}, \\ \text{where} \quad \beta &= \frac{1}{k_B T}, z = e^{\beta\mu} \end{aligned} \tag{48}$$

我们定义 $\tilde{\beta} = \beta\cdot\|\hat{H}\|$。根据前面的讨论：

$$f(\tilde{H}) = \sum_{m=0}^{\infty} c_m T_m(\tilde{H}) \tag{49}$$

5.2. Application

5.2.1. Density of states

通常计算态密度的方法基于精确对角化，在密集的 k 网格上获得哈密顿矩阵的特征值，并对特征值求和：

$$D(E) = \sum_{ik} \delta(E - \epsilon_{ik}) \tag{50}$$

ϵ_{ik} 表示 k 点的第 i 个特征值。在实际计算中，δ 函数往往用高斯分布函数或洛伦兹分布函数近似：

$$G(E - \epsilon_{ik}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{[-\frac{(E - \epsilon_{ik})^2}{2\sigma^2}]\right\} \tag{51}$$

$$L(E - \epsilon_{ik}) = \frac{1}{\pi\sigma} \frac{\sigma^2}{(E - \epsilon_{ik})^2 + \sigma^2} \tag{52}$$

TBPM 方法则通过对平均关联函数的逆傅里叶变换计算 DOS

$$D(E) = \frac{1}{2\pi S} \sum_{p=1}^{S} \int_{-\infty}^{\infty} C_{DOS} e^{iEt} \, \mathrm{d}t \tag{53}$$

S 是随意初始态的数目。式53可以通过快速傅里叶变换计算，如果需要更高的能量分辨率，也可以使用数值积分计算。TBPM 方法计算 DOS 的误差随系统增大按照 $1/\sqrt{SN}$ 减少，因此，如果想要更高的精度，可以增大模型尺寸或者计算更多的随机态平均。对于足够大的系统，例如 10^8 个轨道，只需要一组随机初始态就可以得到足够精确的结果。关于 DOS 计算的更多细节，可以参考这篇文章 [5]。

5.2.2. Local density of states

为了计算特定轨道 i 上的 LDOS，我们只将 $|\varphi(0)\rangle = \sum_i a_i \psi_i$ 中的 a_i 设为非零，然后用与 DOS 相同的方法对关联函数进行分析。

$$d_i(E) = \sum_{jk} \delta(E - \epsilon_{jk}) |U_{ijk}|^2 \tag{54}$$

$$\langle \varphi(0) | \varphi(t) \rangle = \sum_i |U_{ijk} a_i|^2 e^{-i\epsilon_j t} \tag{55}$$

另一种方法基于 Lanczos 算法 [6]，利用递归方法在实空间中评估 LDOS。特定轨道 i 上的 LDOS 是：

$$d_i(E) = \lim_{\epsilon \to 0^+} \frac{1}{\pi} \mathrm{Im} \langle \psi_i | G(E + i\epsilon) | \psi_i \rangle \tag{56}$$

然后利用迭代的方法计算出格林函数 G(E) 的对角元素：

$$\begin{aligned} G_0 &= \langle l_0 | G(E) | l_0 \rangle \\ &= 1/\{E - a_0 - b_1^2/[E - a_1 - b_2^2/(E - a_3 - b_3^2 \ldots)]\} \end{aligned} \tag{57}$$

式中的 a_n 和 b_n 由以下的迭代关系求出：

$$\begin{aligned} a_i &= \langle l_i | H_i | l_i \rangle \\ |m_{i+1}\rangle &= (H - a_i)|l_i\rangle - b_i |l_{i-1}\rangle \\ b_{i+1} &= \sqrt{\langle m_{i+1} | m_{i+1} \rangle} \\ |l_{i+1}\rangle &= \frac{|m_{i+1}\rangle}{b_{i+1}} \\ |l_{-1}\rangle &= |0\rangle \end{aligned} \tag{58}$$

5.2.3. Optical conductivity

为了计算光导率，TBPM 方法将 Kubo 公式和随机状态方法相结合。Kubo 公式通过时间关联函数（time correlation function）计算系统的宏观响应量。它将微观量（如电导率、磁化率）与系统的时间演化联系起来。对于非相互作用的电子系统，由于在 β 方向上的场，在 α 方向上的光学电导率的实部为（省略在 $\omega = 0$ 处的 Drude 贡献）

$$\begin{aligned} \mathrm{Re} = \lim_{E \to 0^+} \frac{e^{-\beta\hbar\omega} - 1}{\hbar\omega A} \int_0^{\infty} e^{-Et} sin(\omega t) \\ \times 2\mathrm{Im} \langle \psi | f(H) J_\alpha(t) [1 - f(H)] J_\beta | \psi \rangle \, \mathrm{d}t \end{aligned} \tag{59}$$

A 是系统的面积或者体积。对于紧束缚的哈密顿量，流密度算符定义为：

$$J = -\frac{ie}{\hbar} \sum_{i,j} t_{ij} \hat{r}_j - \hat{r}_i c_i^\dagger c_j \tag{60}$$

$\hat{r}$ 是位置算符。Fermi-Dirac 贡献，定义为：

$$f(H) = \frac{1}{e^\beta H - \mu - 1} \tag{61}$$

在实际的计算中，通过对实空间的所有基态的随机叠加，确保63的精度。Fermi-Dirac 算符，如前面所述，可以利用切比雪夫多项式和快速傅里叶变换计算。我们引入两个波函数：

$$\begin{aligned} |\psi_1(t)\rangle &= e^{-i\tilde{H}t} [1 - f(\tilde{H})] J_\alpha |\psi(0)\rangle \\ |\psi_2(t)\rangle &= e^{-i\tilde{H}t} f(\tilde{H}) |\psi(0)\rangle \end{aligned} \tag{62}$$

于是可得 $\sigma_{\alpha\beta}(t)$ 的实部：

$$\begin{aligned} \mathrm{Re} = \lim_{E \to 0^+} \frac{e^{-\beta\hbar\omega} - 1}{\hbar\omega A} \int_0^{\infty} e^{-Et} sin(\omega t) \\ \times 2\mathrm{Im} \langle \psi_2 | J_\alpha(t) | \psi \rangle_\beta \, \mathrm{d}t \end{aligned} \tag{63}$$

由 KK(Kramers-Kronig) 关系，推导出 $\sigma_{\alpha\beta}$ 的虚部：

$$\mathrm{Im}\sigma_{\alpha\beta}(\hbar\omega) = -\frac{1}{\pi} P \int_{-\infty}^{\infty} \frac{\mathrm{Re}\sigma_{\alpha\beta}(\hbar\omega')}{\omega' - \omega} \, \mathrm{d}\omega \tag{64}$$

5.2.4. DC conductivity

直流电导率可以通过 $\lim \omega \to 0$ 时的 Kubo 公式来计算。基于之前的 DOS 和准本征态计算方法，T=0 时，直流电导率在 α 方向上的对角线项为：

$$\begin{aligned} \sigma_{\alpha\alpha}(E) &= \lim_{\tau \to \infty} \sigma_{\alpha\alpha}(E, \tau) \\ &= \lim_{\tau \to \infty} \frac{D(E)}{A} \int_0^{\tau} \mathrm{Re}[e^{-iEt} C_{DC}(t)] \, \mathrm{d}t \end{aligned} \tag{65}$$

其中 DC 关联函数定义为：

$$C_{DC} = \frac{\langle \psi(0) | J_\alpha e^{-i\hat{H}t} J_\alpha | \tilde{\Psi}(E) \rangle}{\langle \psi(0) | \tilde{\Psi}(E) \rangle} \tag{66}$$

A 表示面积或者体积。需要强调的是，$|\psi(0)\rangle$ 必须与计算 $|\tilde{\Psi}(E)\rangle$ 时所用的随机初始态相同。不考虑安德森局域化影响的半经典直流电导率定义为：

$$\sigma^{sc} = \sigma_{\alpha\alpha}^{max}(E, \tau) \tag{67}$$

实验中测量的场效应载流子迁移率与半经典直流电导率有关：

$$u(E) = \frac{\sigma^{sc}}{e n_e(E)} \tag{68}$$

其中载流子密度可以由态密度的积分求出：

$$n_e = \int_0^E D(\varepsilon) \, \mathrm{d}\varepsilon \tag{69}$$

5.2.5. Diffusion coefficient

根据 Kubo 公式，直流电导率也可以写成扩散系数的函数：

$$\sigma_{\alpha\alpha}(E) = \frac{e^2}{A} D(E) \lim_{\tau \to \infty} D_{diff}(E, \tau) \tag{70}$$

因此，可以反推出扩散系数的计算方法：

$$D_{diff} = \frac{1}{e^2}\int_0^{\tau} e^{-iEt} C_{DC}(t)\,\mathrm{d}t \tag{71}$$

一旦求出了扩散系数，就可以立即得到载流子速率：

$$v(E) = \sqrt{D_{diff(E,\tau)}/\tau} \tag{72}$$

以及平均自由程、安德森局域长度等物理量。

References

[1] C.Cohen-Tannoudji, 量子力学, 刘家谟译, Ed. 高等教育出版社, 2014, vol. 1.

[2] 单斌, 计算材料学. 华中科技大学出版社, 2023.

[3] H. De Raedt, "Computer simulation of quantum phenomena in nanoscale devices," en, in *Annual Reviews of Computational Physics IV*. WORLD SCIENTIFIC, May 1996, pp. 107–146, isbn: 978-981-02-2728-9. doi: 10.1142/9789812830050_0004. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/9789812830050_0004.

[4] Y. Li, Z. Zhan, X. Kuang, Y. Li, and S. Yuan, "Tbplas: A tight-binding package for large-scale simulation," en, *Computer Physics Communications*, vol. 285, p. 108 632, Apr. 2023, issn: 00104655. doi: 10.1016/j.cpc.2022.108632.

[5] S. Yuan, H. De Raedt, and M. I. Katsnelson, "Modeling electronic structure and transport properties of graphene with resonant scattering centers," *Phys. Rev. B*, vol. 82, p. 115 448, 11 2010. doi: 10.1103/PhysRevB.82.115448. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevB.82.115448.

[6] R Haydock, V Heine, and M. J. Kelly, "Electronic structure based on the local atomic environment for tight-binding bands," *Journal of Physics C: Solid State Physics*, vol. 5, no. 20, p. 2845, 1972. doi: 10.1088/0022-3719/5/20/004. [Online]. Available: https://dx.doi.org/10.1088/0022-3719/5/20/004.

[7] *PGFPlots - A LaTeX package to create plots.* [Online]. Available: https://pgfplots.sourceforge.net/.