Schrodinger Equation on the Computer

1 The Algorithm

The non-relativistic Schrodinger Equation describing the time-evolution of the wave function of a quantum particle is

$$i\hbar\frac{\partial\psi(\vec{r},t)}{\partial t} = -\frac{\hbar^2}{2m}\nabla^2\psi(\vec{r},t) + V(\vec{r})\psi(\vec{r},t) \tag{1}$$

where $V(\vec{r})$ is the (classical) potential energy function due to the interaction of the particle with its environment. Given the wavefunction at some instant, this first order differential equation in time can be, in principle, integrated to compute the wavefunction at any other instant of time. There are standard algorithms that treat this as a standard parabolic partial differential equation, and exploit the tools to solve such differential equations. However, such algorithms, though precise up to a high degree, hide the essence of the linear vector space structure of quantum mechanics, and Schrodinger equation being a unitary evolution in the vector space of quantum states. In this note, we look at an algorithm which, perhaps not as accurate as the others, make this connection manifest. For simplicity, we consider a particle propagationg in one-dimension (or, more physically, constrained to be confined to a very thin 'line' due to a suitable interaction). Then, the equation becomes

$$i\hbar\frac{\partial\psi(x,t)}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi(x,t)}{\partial x^2} + V(x)\psi(x,t) \tag{2}$$

The first step in solving the equation through an algorithm is to discretize the space in which the particle propagates. We visualise the discrete space as a lattice of points with lattice spacing a. Let the lattice points be labelled x_i. The wavefunction at i^{th} site is then

$$\psi(x_i,t) = \langle x_i|\ \psi(t)\rangle$$

where $|x_i\rangle$ is state of definite position with a Dirac delta normalisation $\langle x_i|\ x_j\rangle = \delta\left(x_i - x_j\right)$. The wavefunction normalisation is

$$\int_{-\infty}^{\infty} dx\,|\psi(x,t)|^2 = 1$$

For the discrete system, this reduces to

$$\sum_i a\,|\langle x_i|\ \psi(t)\rangle|^2 = 1$$

This motivates us to define normalisable states of definite position $|\tilde{x_i}\rangle = \sqrt{a}\,|x_i\rangle$. For the states $|x_i\rangle$ to satisfy the delta function normalisation, these states satisfy $\langle\tilde{x_i}|\ \tilde{x_j}\rangle = \delta_{ij}$ where δ_{ij} is the Kronecker delta. To see that is correctly gives a delta function normalisation for the states $|x_i\rangle$, we note that given $\langle\tilde{x_i}|\ \tilde{x_j}\rangle = \delta_{ij}$, it follows that $\langle x_i|\ x_j\rangle = (1/a)\delta_{ij}$. We now need to demonstrate that this is a discrete Dirac delta function. First, this is clearly zero if $x_i \neq x_j$. Since $\int dx \to \sum_i a$, we need to demonstrate that this satisfies the other property of a delta function

$$\int_{-\infty}^{\infty} dx_i\ \delta(x_i - x_j) = 1$$

whose discrete form should be

$$\sum_i a \langle x_i | \, x_j \rangle = 1$$

This is clearly true, since

$$\begin{aligned} \sum_i a \langle x_i | \, x_j \rangle &= \sum_i a \times \frac{1}{a} \delta_{ij} \\ &= \sum_i \delta_{ij} \\ &= 1 \end{aligned}$$

Then

$$\begin{aligned} \psi(x_i, t) &= \langle x_i | \, \psi(t) \rangle \\ &= (1/\sqrt{a}) \langle \tilde{x_i} | \, \psi(t) \rangle \\ &= (1/\sqrt{a}) \, \phi_i(t) \end{aligned}$$

where $\phi_i = \langle \tilde{x_i} | \, \psi \rangle$. This function is dimensionless and satisfies the normalisation condition

$$\sum_i |\phi_i|^2 = 1 \tag{3}$$

The discretised second derivative of the wavefunction, evaluated at lattice point x_i, is

$$\left. \frac{\partial^2 \psi(x,t)}{\partial x^2} \right|_{x_i} = \frac{\psi(x_i + a, t) - 2\psi(x_i, t) + \psi(x_i - a, t)}{a^2}$$

Since $\psi(x_i + a) = \psi_{i+1}$, etc., the discretised Schrodinger equation (written in terms of the dimensionless wavefunction ϕ) reduces to

$$i\hbar \, \frac{d\phi_i(t)}{dt} = -\frac{\hbar^2}{2ma^2} \left[\phi_{i+1}(t) - 2\phi_i(t) + \phi_{i-1}(t) \right] + V_i \phi_i(t)$$

where $V_i = V(x_i)$. This can be written as a matrix equation

$$i\hbar \, \frac{d\phi(t)}{dt} = H\phi \tag{4}$$

where ϕ is a column vector with entries ϕ_i and H is the Hamiltonian matrix with matrix elements

$$H_{ij} = -\frac{\hbar^2}{2ma^2} \left[\delta_{i+1,j} - 2 \, \delta_{i,j} + \delta_{i-1,j} \right] + V_i \, \delta_{i,j}$$

It is easy to verify that H is Hermitian (in fact, it is symmetric: $H_{ij} = H_{ji}$). The generic problem is as follows: the state $\phi(0)$ is given at $t = 0$ and the state $\phi(t)$ at instant t is to be determined. Since we have reduced the problem to a unitary time evolution generated by the Hermitian matrix H, we can immediately write the formal solution to eqn.(4)

$$\phi(t) = e^{-iHt/\hbar} \phi(0) \tag{5}$$

To implement this solution, we first diagonalise the Hamiltonian matrix H to determine its eigenvalues E_i and eigenvectors ϕ_{E_i}

$$H\phi_{E_i} = E_i \phi_{E_i} \tag{6}$$

Since H is Hermitian, the (normalised) eigenvectors will form an orthonormal basis for the vector space

$$\phi_{E_i}^\dagger \phi_{E_j} = \delta_{ij} \tag{7}$$

The completeness of the set $\{\phi_{E_i}\}$ allows us to express the state at $t = 0$ as a linear superposition of these states

$$\phi(0) = \sum_i c_i \phi_{E_i}$$

where $c_i = \phi^\dagger_{E_i}\phi(0)$. The state at instant t is then given by

$$\begin{aligned} \phi(t) &= e^{-iHt/\hbar}\phi(0) \\ &= \sum_i c_i \; e^{-iE_i t/\hbar}\phi_{E_i} \end{aligned}$$

This completes the algorithm. The implementation of the algorithm requires the diagonalisation of the Hamiltonian matrix. The following example discusses how this is achieved computationally.

2 Example and Dimensional Analysis

As an example, we take the quantum harmonic oscillator, for which $V(x) = (1/2)m\omega^2 x^2$, where ω is the classical (angular) frequency of the oscillator. We label the spatial lattice points as $x_i = i \times a$ where i is an integer. The Hamiltonian matrix elements are

$$\begin{aligned} H_{ij} &= -\frac{\hbar^2}{2ma^2}\left[\delta_{i+1,j} - 2\;\delta_{i,j} + \delta_{i-1,j}\right] + (1/2)m\omega^2 x_i^2\;\delta_{i,j} \\ &= -\frac{\hbar^2}{2ma^2}\left[\delta_{i+1,j} - 2\;\delta_{i,j} + \delta_{i-1,j}\right] + (1/2)m\omega^2 a^2\;i^2\;\delta_{i,j} \end{aligned}$$

We now need to do some dimensional analysis, to express variables in units of natural scales. There is a natural length scale $l_0 = \sqrt{\hbar/m\omega}$, natural energy scale $E_0 = (1/2)\hbar\omega$ and natural time scale is $t_0 = 2/\omega$ (the factor of 2 simplifies the form of the time evolution equation). We express the lattice spacing a in terms of l_0 as $a = l_0\Delta$ where Δ is dimensionless. Then the Hamiltonian simplifies to

$$H_{ij} = \frac{\hbar\omega}{2}\left[-\frac{\delta_{i+1,j} - 2\;\delta_{i,j} + \delta_{i-1,j}}{\Delta^2} + i^2\delta_{i,j}\right]$$

The eigenvalue equation is

$$H\phi_{E_i} = E_i\phi_{E_i}$$

We measure E_i in natural unit of energy E_0, so that $E_i = \epsilon_i E_0$ where ϵ_i is the dimensionless energy eigenvalue. Then the eigenvalue equation becomes

$$\tilde{H}\phi_{E_i} = \epsilon_i\phi_{E_i}$$

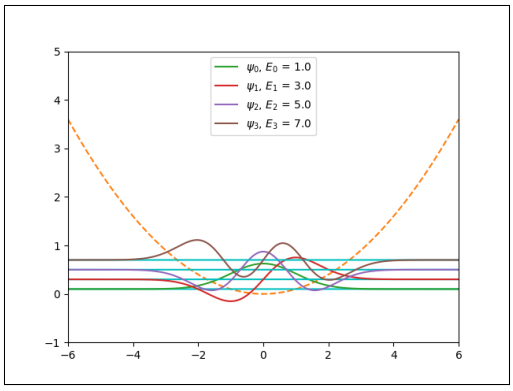
where $\tilde{H} = H/E_0$. To compute the time-evolution, we use time $\tau = t/t_0$. Then the time evolution becomes

$$\phi(\tau) = \sum_i c_i \; e^{-i\epsilon_i\tau}\phi_{E_i}$$

which solves the problem. From a computational point of view, the key is to set up the Hamiltonian matrix and determine its eigenvalues and eigenvectors.

```
import numpy as np  ## Imports module 'numpy' and uses alias 'np' to use it.
from numpy import linalg as lin   ## imports 'linalg' (a linear algebra module) and uses alias 'lin' to
    use it.
from pylab import *  ## This imports matplotlib, a plotting module.

delta = 0.01   ## Lattice spacing.
endpoint = 6.0 ## Lattice extends from x = -6.0 to +6.0
N = 600 ## Number of lattice points is 2*N+1
x = arange(-6.0,6.01,0.01)   ### Creates a list of numbers from -6.0 to 6.0 in steps of 0.1. This stores
    the lattice.

def kronecker(i,j):   ### The Kronecker Delta function.
    if i == j:
        return 1
    else:
        return 0

def v(z):
    return z**2

def h(i,j):  ### This defines the matrix element of the discretized Hamiltonian operator for this
    interaction
    return (-kronecker(i+1,j) + 2*kronecker(i,j) - kronecker(i-1,j))/delta**2 + v(delta*i) *
        kronecker(i,j)


H = np.array( [[h(i,j) for i in range(-N,N+1)] for j in range(-N,N+1)] )  ## Constructs the Hamiltonian
    matrix from its matrix elements.

H_eigenvalues, H_eigenvectors = lin.eig(H)  ## H_eigenvalues stores the eigenvalues and H_eigenvectors
    stores the eigenvectors as columns.
idx = H_eigenvalues.argsort()  ### These three lines sort the eigenvalues and eigenvectors in order of
    increasing eigenvalues.
H_eigenvalues = H_eigenvalues[idx]
H_eigenvectors = H_eigenvectors[:,idx]

eigen = [H_eigenvalues[i] for i in range(4)]  ### First 4 eigenvalues
y = [H_eigenvectors[:,i] for i in range(4)] ### first 4 eigenvectors



potential = 0.1*np.array([v(delta*i) for i in range(-N,N+1)])  ### The potential energy function to be
    plotted, suitably scaled to fit the plot.
fig = plt.figure()
ax = plt.axes(xlim=(-6,6), ylim=(-1,5))
line, = ax.plot([], [], lw=2)
plot(x,potential, '--')  ### Plot of the potential energy function.
for i in range(0,4):
    Energy = np.linspace(eigen[i],eigen[i],1201)  ### energy level to be plotted.
    plot(x,0.1*Energy, '-c')  ## Plots energy level.
    plot(x,-7*y[i]+0.1*Energy, label = '$\psi_ {%s}$, $E_{%s}$ = %s' % (i, i, round(eigen[i],2))) ##
        Plots wavefunction with origin shifted to
    #the location of the energy level. The energy is reported in units of hbar*omega/2

legend()
show()
```

The output of the program shows the ground state and the first three excited state energy eigenvalues and wavefunctions



We now modify this program to compute the time evolution. The following program takes an initial wavefunction (a Gaussian wavefunction in this example) and evolves it with time under the harmonic oscillator interaction. It displays the time evolved probability function $|\phi|^2$ as an animation and also saves the animation to a file using 'ffmpeg'

```
import numpy as np
from numpy import linalg as lin
from matplotlib import pyplot as plt
from matplotlib import animation # Animation module.

Writer = animation.writers['ffmpeg'] ## These lines set up saving the created animation.
writer = Writer(fps=15, metadata=dict(artist='Me'), bitrate=1800)

delta = 0.1   ## Lattice spacing.
endpoint = 6.0 ## Lattice extends from x = -6.0 to +6.0
N = 60
dimension = 2*N + 1 # Number of lattice points.
x = np.linspace(-6,6,2*N+1)   ### List of numbers from -6.0 to 6.0 in steps of 0.1 (lattice)

def kronecker(i,j):
    if i == j:
        return 1
    else:
        return 0
def h(i,j):
    return (-kronecker(i+1,j) + 2*kronecker(i,j) - kronecker(i-1,j))/delta**2 + delta**2 * i**2 *
        kronecker(i,j)

H = np.array( [[h(i,j) for i in range(-N,N+1)] for j in range(-N,N+1)] )
H_eigenvalues, H_eigenvectors = lin.eig(H)
idx = H_eigenvalues.argsort()
H_eigenvalues = H_eigenvalues[idx]
H_eigenvectors = H_eigenvectors[:,idx]

###### The initial Gaussian wavefunction ##########
a = 1.0 ## Initial spread in units of l0
b = 0.0 ## Initial peak of the Gaussian
p0 = 2.0 ## Initial momentum in units of hbar/l0

def psi0(y):
   return (1/pow(np.pi*(a**2),0.25))*np.exp(-((y-b)**2)/(2.0*a**2) - 1j*p0*y)

Psi0 = np.sqrt(delta)*np.array( [psi0(delta*i) for i in range(-N,N+1)], 'complex' )
#################### Time evolving state  ################################
def Psi(t): # Time evolution function
    sum = np.zeros(dimension, 'complex')
    for n in range(dimension):
        c = np.vdot(Psi0, H_eigenvectors[:,n]) # nth expansion coefficient
        E = H_eigenvalues[n]
        sum += c * np.exp(-E*t*1.0j) * H_eigenvectors[:,n]
    return sum

def Prob(t): # Probability function
    return np.array( [abs(Psi(t)[i])**2 for i in range(dimension)] )

u = np.linspace(-6, 6, 2*N+1,endpoint=True)
v = 0.003*u**2
fig = plt.figure()
ax = plt.axes(xlim=(-6,6), ylim=(0,0.2))
line, = ax.plot([], [], lw=2)
plt.plot(u,v)

def init():
    line.set_data([], [])
    return line,
def animate(i):
    x = np.linspace(-6, 6, 121)
    y = Prob(0.05*i)
    line.set_data(x, y)
    return line,

anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=200, interval=20, blit=True)
anim.save('Time_Evolution_Oscillator.mp4', writer=writer)
plt.show()
```