数值分析第 6 次上机作业

学号：221840189，姓名：王晨光

§1 问题

实现快速傅里叶变换 FFT，并应用于某个具体场景。如信号，图像，声音处理，多项式乘法，偏微分方程求解 (谱方法)，或其它应用场景。

§2 算法思路

2.1 快速傅里叶变换 (FFT)

FFT 快速傅里叶变换算法利用了复数单位根的性质极大了提高了离散傅里叶变换算法的效率，将原本 $O(n^2)$ 的时间复杂度降低到了 $O(n\log n)$。

DFT 的计算可归结为计算

$$c_k=\sum_{j=0}^{N-1}x_jw^{kj}(k=0,1,\cdots,N-1)$$

其中 $w=e^{-\mathrm{i}\frac{2\pi}{N}},\{x_j\}$ 为已知复数序列。

计算一个 c_k 需要 N 个复数乘法，计算全部 c_k 需要 N^2 个复数乘法。

FFT 想法：

$$e^{\mathrm{i}k\frac{2\pi}{N}j}=\cos\frac{2\pi}{N}kj+\mathrm{i}\sin\frac{2\pi}{N}kj\ (k,j=0,1,\cdots,N-1)$$

其中只有 N 个不同的值。特别地，当 $N=2^p$ 时只有 $\dfrac{N}{2}$ 个不同的值，因此，在计算 c_k 时，可合并大量同类项，从而减少乘法次数。

最终可以得到 FFT 计算公式：

$$\begin{cases}A_q\left(j2^q+k\right)=A_{q-1}\left(j2^{q-1}+k\right)+A_{q-1}\left(j2^{q-1}+k+2^{P-1}\right)\\A_q\left(j2^q+k+2^{q-1}\right)=\left[A_{q-1}\left(j2^{q-1}+k\right)-A_{q-1}\left(j2^{q-1}+k+2^{P-1}\right)\right]w^{j2^{q-1}}\end{cases}$$

$$q=1,\cdots,p,j=0,1,\cdots,2^{p-q}-1,k=0,1,\ldots,2^{q-1}-1$$

$$A_0(j2^q+k)=f(x_{j2^q+k}),A_p(j2^q+k)=c_{j2^q+k}$$

快速傅里叶变换在数字图像处理、机床噪声分析、数据采集、现代雷达、机车故障检测记录等领域都有相关应用。正因为 FFT 在那么多领域里如此有用，python 提供了很多标准工具和封装来计算它。NumPy 和 SciPy 都有经过充分测试的封装好的 FFT 库，分别位于子模块 numpy.fft 和 scipy.fftpack。下面，我们考虑如何用快速傅里叶变换求解一类偏微分方程——薛定谔方程。

2.2 薛定谔方程

薛定谔方程（Schrödinger equation），又称薛定谔波动方程（Schrödinger wave equation），是由奥地利物理学家薛定谔提出的量子力学中的一个基本方程，也是量子力学的一个基本假定。它是将物

质波的概念和波动方程相结合建立的二阶偏微分，可描述观察粒子的运动，每个微观系统都有一个相应的薛定谔方程式，通过解方程可得到波函数的具体形式以及对应的能量，从而了解微观系统的性质。薛定谔方程表明量子力学中，粒子以概率的方式出现，具有不确定性，宏观尺度下失效可忽略不计。

一维量子系统的动力学由与时间有关的薛定谔方程控制：

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

其中，描述微观粒子状态的波函数 ψ 和描述势场的势函数 V 是位置 x 和时间 t 的方程，微观粒子的质量为 m，$\hbar = 1.05457266(63) \times 10^{-34}$ J·s 为约化普朗克常数（角动量的最小衡量单位）。假设我们在一维空间跟随一个单一质点运动，这个波函数表示微观粒子在时间 t 位于位置 x 的概率. 量子力学告诉我们，与我们熟悉的经典力学不同，这个概率不是对系统认识的极限，而是反映了在非常小的领域内事件的位置和时间不可避免的不确定性。

2.3 分步傅里叶方法

使用傅里叶变换求某些微分方程的数值解是一个标准的做法。我们使用如下形式的傅里叶方程：

$$\widetilde{\psi}(k,t) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}\psi(x,t)e^{-ikx}dx$$

则相关的傅里叶逆变换为：

$$\psi(k,t) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}\widetilde{\psi}(k,t)e^{ikx}dk$$

将其带薛定谔方程并且化简就得到傅里叶空间形式的薛定谔方程：

$$i\hbar\frac{\partial\widetilde{\psi}}{\partial t} = \frac{\hbar^2k^2}{2m}\widetilde{\psi} + V\left(i\frac{\partial}{\partial k}\right)\widetilde{\psi}$$

综合两种形式的薛定谔方程，我们可以提出一个计算薛定谔方程的有效方法。

首先，求解 x 空间薛定谔方程的简单部分

$$i\hbar\frac{\partial\psi}{\partial t} = V(x)\psi$$

易知，当时间有一个小变化时，上述方程有一个形如下式的解

$$\psi(x,t+\Delta t) = \psi(x,t)e^{-iV(x)\Delta t/h}$$

其次，求解 k 空间薛定谔方程的简单部分

$$i\hbar\frac{\partial\widetilde{\psi}}{\partial t} = \frac{\hbar^2k^2}{2m}\widetilde{\psi}$$

易知，当时间有一个小变化时，上述方程有一个形如下式的解

$$\widetilde{\psi}(k,t+\Delta t) = \widetilde{\psi}(k,t)e^{-ihk^2\Delta t/2m}$$

2.4 算法思想

要想得到数值解，需要反复计算傅里叶变换 $\psi(x,t)$ 和傅里叶逆变换 $\widetilde{\psi}(k,t)$。求解傅里叶变换的最著名的算法就是快速傅里叶变换，其能有效计算如下形式的离散傅里叶变换

$$\widetilde{F_m} = \sum_{n=0}^{N-1} F_n e^{-imn\frac{2\pi}{N}}$$

以及其逆变换

$$F_n = \frac{1}{N}\sum_{m=0}^{N-1} \widetilde{F_m} e^{-imn\frac{2\pi}{N}}$$

我们需要知道以上式子和上面定义的连续傅里叶变换有怎样的联系。假设无穷积分能很好地由 a 到 b 的有限积分近似，因此我们有

$$\widetilde{\psi}(k,t) = \frac{1}{\sqrt{2\pi}}\int_a^b \psi(x,t)e^{-ikx}dx$$

这个近似最终等价于假设 $V(x)\to\infty(x\le a, x\ge b)$ ，我们可以把这个积分写成黎曼和的形式，定义 $\Delta x = (b-a)/N$ 以及 $x_n = a + n\Delta x$ ，有

$$\widetilde{\psi}(k,t) \simeq \frac{1}{\sqrt{2\pi}}\sum_{n=0}^{N-1} \psi\left(x_n,t\right) e^{-ikx_n}\Delta x$$

再定义 $\Delta k = 2\pi/(N\Delta x)$ 以及 $k_m = k_0 + m\Delta k$ ，有

$$\widetilde{\psi}(k,t) \simeq \frac{1}{\sqrt{2\pi}}\sum_{n=0}^{N-1} \psi\left(x_n,t\right) e^{-ikx_n}\Delta x$$

规定 $k_0 = -\pi/\Delta x$，将 x_n 和 k_n 的表达式代入傅里叶近似的式子中，有

$$\left[\widetilde{\psi}\left(k_m,t\right)e^{imx_0\Delta k}\right] \simeq \sum_{n=0}^{N-1}\left[\frac{\Delta x}{\sqrt{2\pi}}\psi\left(x_n,t\right)e^{-ik_0x_n}\right]e^{-imn\frac{2\pi}{N}}$$

类似地，代入傅里叶逆变换的式子中，有

$$\left[\frac{\Delta x}{\sqrt{2\pi}}\psi\left(x_n,t\right)e^{ik_0x_n}\right] \simeq \frac{1}{N}\sum_{m=0}^{N-1}\left[\widetilde{\psi}\left(k_m,t\right)e^{-imx_0\Delta k}\right]e^{-imn\frac{2\pi}{N}}$$

对于连续傅里叶对有

$$\psi(x,t) \Longleftrightarrow \widetilde{\psi}(k,t)$$

相应地，离散傅里叶对有

$$\frac{\Delta x}{\sqrt{2\pi}}\psi\left(x_n,t\right)e^{-ik_0x_n} \Longleftrightarrow \widetilde{\psi}\left(k_m,t\right)e^{-imx_0\Delta k}$$

基于此，我们得出薛定谔方程的一个快速数值估计。

§3 结果分析

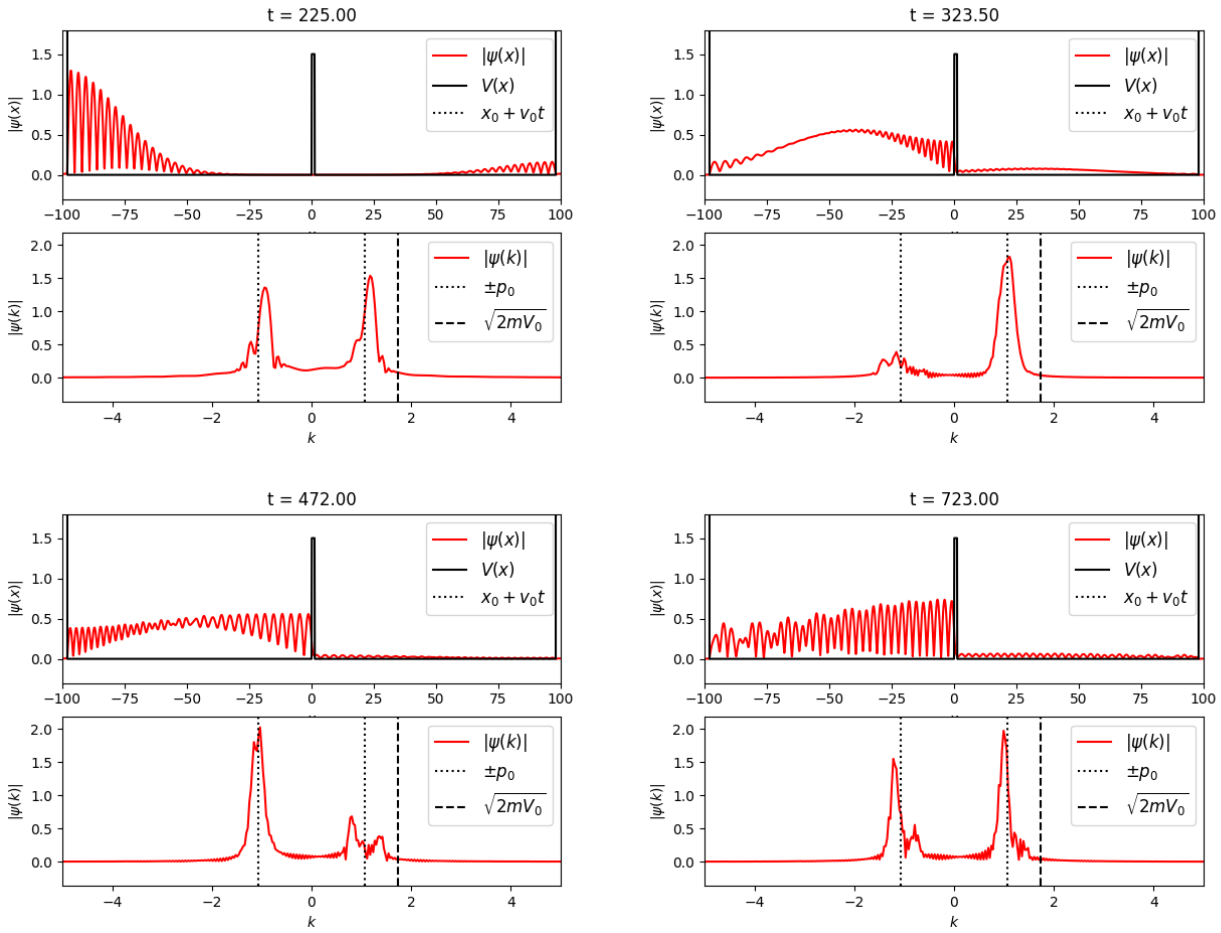

图 1: 不同状态下的快速傅里叶变换求薛定谔方程数值解结果

实验证明，快速傅里叶变换能较快且有效求解出薛定谔方程的数值解，具有非常好的应用效果。

§4 结论

快速傅里叶变换可以高效地对微分方程进行数值求解，通过合理选择傅里叶级数的展开方式和截断技术，可以在保证精度的前提下极大地提高数值求解的效率。

§5 附录: 程序完整代码

```
import numpy as np
from matplotlib import pyplot as pl
from matplotlib import animation
from scipy.fftpack import fft, ifft


class Schrodinger(object):
    """
    Class which implements a numerical solution of the time-dependent
    Schrodinger equation for an arbitrary potential
    """

    def __init__(self, x, psi_x0, V_x,
                 k0=None, hbar=1, m=1, t0=0.0):
        """
        Parameters
        ----------
        x : array_like, float
            length-N array of evenly spaced spatial coordinates
        psi_x0 : array_like, complex
            length-N array of the initial wave function at time t0
        V_x : array_like, float
            length-N array giving the potential at each x
        k0 : float
            the minimum value of k.  Note that, because of the workings of the
            fast fourier transform, the momentum wave-number will be defined
            in the range
              k0 < k < 2*pi / dx
            where dx = x[1]-x[0].  If you expect nonzero momentum outside this
            range, you must modify the inputs accordingly.  If not specified,
            k0 will be calculated such that the range is [-k0,k0]
        hbar : float
            value of planck's constant (default = 1)
        m : float
            particle mass (default = 1)
        t0 : float
            initial tile (default = 0)
        """
        # Validation of array inputs
        self.x, psi_x0, self.V_x = map(np.asarray, (x, psi_x0, V_x))
        N = self.x.size
        assert self.x.shape == (N,)
        assert psi_x0.shape == (N,)
```

```
44              assert self.V_x.shape == (N,)
45
46              # Set internal parameters
47              self.hbar = hbar
48              self.m = m
49              self.t = t0
50              self.dt_ = None
51              self.N = len(x)
52              self.dx = self.x[1] - self.x[0]
53              self.dk = 2 * np.pi / (self.N * self.dx)
54
55              # set momentum scale
56              if k0 == None:
57                  self.k0 = -0.5 * self.N * self.dk
58              else:
59                  self.k0 = k0
60              self.k = self.k0 + self.dk * np.arange(self.N)
61
62              self.psi_x = psi_x0
63              self.compute_k_from_x()
64
65              # variables which hold steps in evolution of the
66              self.x_evolve_half = None
67              self.x_evolve = None
68              self.k_evolve = None
69
70              # attributes used for dynamic plotting
71              self.psi_x_line = None
72              self.psi_k_line = None
73              self.V_x_line = None
74
75          def _set_psi_x(self, psi_x):
76              self.psi_mod_x = (psi_x * np.exp(-1j * self.k[0] * self.x)
77                                * self.dx / np.sqrt(2 * np.pi))
78
79          def _get_psi_x(self):
80              return (self.psi_mod_x * np.exp(1j * self.k[0] * self.x)
81                      * np.sqrt(2 * np.pi) / self.dx)
82
83          def _set_psi_k(self, psi_k):
84              self.psi_mod_k = psi_k * np.exp(1j * self.x[0]
85                                              * self.dk * np.arange(self.N))
86
87          def _get_psi_k(self):
88              return self.psi_mod_k * np.exp(-1j * self.x[0] *
```

```
89                                          self.dk * np.arange(self.N))
90
91          def _get_dt(self):
92              return self.dt_
93
94          def _set_dt(self, dt):
95              if dt != self.dt_:
96                  self.dt_ = dt
97                  self.x_evolve_half = np.exp(-0.5 * 1j * self.V_x
98                                              / self.hbar * dt)
99                  self.x_evolve = self.x_evolve_half * self.x_evolve_half
100                 self.k_evolve = np.exp(-0.5 * 1j * self.hbar /
101                                        self.m * (self.k * self.k) * dt)
102
103         psi_x = property(_get_psi_x, _set_psi_x)
104         psi_k = property(_get_psi_k, _set_psi_k)
105         dt = property(_get_dt, _set_dt)
106
107         def compute_k_from_x(self):
108             self.psi_mod_k = fft(self.psi_mod_x)
109
110         def compute_x_from_k(self):
111             self.psi_mod_x = ifft(self.psi_mod_k)
112
113         def time_step(self, dt, Nsteps=1):
114             """
115             Perform a series of time-steps via the time-dependent
116             Schrodinger Equation.
117
118             Parameters
119             ----------
120             dt : float
121                 the small time interval over which to integrate
122             Nsteps : float, optional
123                 the number of intervals to compute.  The total change
124                 in time at the end of this method will be dt * Nsteps.
125                 default is N = 1
126             """
127             self.dt = dt
128
129             if Nsteps > 0:
130                 self.psi_mod_x *= self.x_evolve_half
131
132             for i in range(Nsteps - 1):
133                 self.compute_k_from_x()
```

```
134                 self.psi_mod_k *= self.k_evolve
135                 self.compute_x_from_k()
136                 self.psi_mod_x *= self.x_evolve
137 
138             self.compute_k_from_x()
139             self.psi_mod_k *= self.k_evolve
140 
141             self.compute_x_from_k()
142             self.psi_mod_x *= self.x_evolve_half
143 
144             self.compute_k_from_x()
145 
146             self.t += dt * Nsteps
147 
148 
149     ##---------------------------------------------------------------------
150     # Helper functions for gaussian wave-packets
151 
152     def gauss_x(x, a, x0, k0):
153         """
154         a gaussian wave packet of width a, centered at x0, with momentum k0
155         """
156         return ((a * np.sqrt(np.pi)) ** (-0.5)
157                 * np.exp(-0.5 * ((x - x0) * 1. / a) ** 2 + 1j * x * k0))
158 
159 
160     def gauss_k(k, a, x0, k0):
161         """
162         analytical fourier transform of gauss_x(x), above
163         """
164         return ((a / np.sqrt(np.pi))**0.5
165                 * np.exp(-0.5 * (a * (k - k0)) ** 2 - 1j * (k - k0) * x0))
166 
167 
168     ##---------------------------------------------------------------------
169     # Utility functions for running the animation
170 
171     def theta(x):
172         """
173         theta function :
174           returns 0 if x<=0, and 1 if x>0
175         """
176         x = np.asarray(x)
177         y = np.zeros(x.shape)
178         y[x > 0] = 1.0
```

```
179         return y
180
181
182     def square_barrier(x, width, height):
183         return height * (theta(x) - theta(x - width))
184
185     ########################################################################
186     # Create the animation
187
188
189     # specify time steps and duration
190     dt = 0.01
191     N_steps = 50
192     t_max = 120
193     frames = int(t_max / float(N_steps * dt))
194
195     # specify constants
196     hbar = 1.0   # planck's constant
197     m = 1.9      # particle mass
198
199     # specify range in x coordinate
200     N = 2 ** 11
201     dx = 0.1
202     x = dx * (np.arange(N) - 0.5 * N)
203
204     # specify potential
205     V0 = 1.5
206     L = hbar / np.sqrt(2 * m * V0)
207     a = 3 * L
208     x0 = -60 * L
209     V_x = square_barrier(x, a, V0)
210     V_x[x < -98] = 1E6
211     V_x[x > 98] = 1E6
212
213     # specify initial momentum and quantities derived from it
214     p0 = np.sqrt(2 * m * 0.2 * V0)
215     dp2 = p0 * p0 * 1./80
216     d = hbar / np.sqrt(2 * dp2)
217
218     k0 = p0 / hbar
219     v0 = p0 / m
220     psi_x0 = gauss_x(x, d, x0, k0)
221
222     # define the Schrodinger object which performs the calculations
223     S = Schrodinger(x=x,
```

```
                        psi_x0=psi_x0,
                        V_x=V_x,
                        hbar=hbar,
                        m=m,
                        k0=-28)

    ########################################################################
    # Set up plot
    fig = pl.figure()

    # plotting limits
    xlim = (-100, 100)
    klim = (-5, 5)

    # top axes show the x-space data
    ymin = 0
    ymax = V0
    ax1 = fig.add_subplot(211, xlim=xlim,
                          ylim=(ymin - 0.2 * (ymax - ymin),
                                ymax + 0.2 * (ymax - ymin)))
    psi_x_line, = ax1.plot([], [], c='r', label=r'$|\psi(x)|$')
    V_x_line, = ax1.plot([], [], c='k', label=r'$V(x)$')
    center_line = ax1.axvline(0, c='k', ls=':',
                              label=r"$x_0 + v_0t$")

    title = ax1.set_title("")
    ax1.legend(prop=dict(size=12))
    ax1.set_xlabel('$x$')
    ax1.set_ylabel(r'$|\psi(x)|$')

    # bottom axes show the k-space data
    ymin = abs(S.psi_k).min()
    ymax = abs(S.psi_k).max()
    ax2 = fig.add_subplot(212, xlim=klim,
                          ylim=(ymin - 0.2 * (ymax - ymin),
                                ymax + 0.2 * (ymax - ymin)))
    psi_k_line, = ax2.plot([], [], c='r', label=r'$|\psi(k)|$')

    p0_line1 = ax2.axvline(-p0 / hbar, c='k', ls=':', label=r'$\pm p_0$')
    p0_line2 = ax2.axvline(p0 / hbar, c='k', ls=':')
    mV_line = ax2.axvline(np.sqrt(2 * V0) / hbar, c='k', ls='--',
                          label=r'$\sqrt{2mV_0}$')
    ax2.legend(prop=dict(size=12))
    ax2.set_xlabel('$k$')
    ax2.set_ylabel(r'$|\psi(k)|$')
```

```
269
270     V_x_line.set_data(S.x, S.V_x)
271
272     ################################################################################
273     # Animate plot
274
275
276     def init():
277         psi_x_line.set_data([], [])
278         V_x_line.set_data([], [])
279         center_line.set_data([], [])
280
281         psi_k_line.set_data([], [])
282         title.set_text("")
283         return (psi_x_line, V_x_line, center_line, psi_k_line, title)
284
285
286     def animate(i):
287         S.time_step(dt, N_steps)
288         psi_x_line.set_data(S.x, 4 * abs(S.psi_x))
289         V_x_line.set_data(S.x, S.V_x)
290         center_line.set_data(2 * [x0 + S.t * p0 / m], [0, 1])
291
292         psi_k_line.set_data(S.k, abs(S.psi_k))
293         title.set_text("t␣=␣%.2f" % S.t)
294         return (psi_x_line, V_x_line, center_line, psi_k_line, title)
295
296
297     # call the animator.  blit=True means only re-draw the parts that have changed
298     anim = animation.FuncAnimation(fig, animate, init_func=init,
299                                    frames=frames, interval=30, blit=True)
300
301
302     # uncomment the following line to save the video in mp4 format.  This
303     # requires either mencoder or ffmpeg to be installed on your system
304
305     #anim.save('schrodinger_barrier.mp4', fps=15, extra_args=['-vcodec', 'libx264'
306
307     pl.show()
```