“计算物理学”第 3 次大作业

李聪乔 1500011425

2019 年 6 月 13 日

目录

1 求解含时薛定谔方程

题目简述：考虑一维原子模型，定态薛定谔方程

$$\left(-\frac{1}{2}\frac{\mathrm{d}^2}{\mathrm{d}x^2}+V(x)\right)\psi(x)=E_n\psi(x)$$

含时薛定谔方程

$$\mathrm{i}\frac{\partial}{\partial t}\psi(x,\,t)=\left(-\frac{1}{2}\frac{\mathrm{d}^2}{\mathrm{d}x^2}+V(x)+xE(t)\right)\psi(x,\,t)$$

其中我们使用软化库伦势 $V(x)=-\frac{1}{\sqrt{x^2+2}}$，这里外加强激光场只考虑电场分量

$$E(t)=\sqrt{I}\sin^2\frac{\omega t}{2N}\sin\omega t$$

激光开始时刻为 0，结束时刻为 $t_f=2N\pi/\omega$. 求解以下问题：

(1) 求解基态波函数概率分布 $|\psi_0(x)|^2$ 和基态能量 E_0 与 -0.48 之差.

(2) 外加激光场，参数给定，计算氢原子 1s 态布居数随时间演化 $P_0(t)=|\langle\psi_0(x)|\psi(x,t)\rangle|^2$ 和电离末态电子动量谱 $P(k)=|\langle\psi_k|\psi_f\rangle|^2$，其中 $\psi_k=\mathrm{e}^{\mathrm{i}kx}$，$\psi_f=\psi(t_f)-\langle\psi_0|\psi(t_f)\rangle\,\psi_0$. 画出 $k\in[-2.5,\,2.5]$ 上线性纵坐标和对数纵坐标图 $P(k)$.

(3) 用偶极矩和加速度两种方法计算高次谐波功率谱 $\log|A(\omega)|^2$. 其中偶极矩方法为

$$d(t)=\langle\psi(x,t)|x|\psi(x,t)\rangle\,,\quad A(\omega)=\frac{1}{\sqrt{2\pi}}(-\omega^2)\int_0^{t_f}\mathrm{e}^{-\mathrm{i}\omega t}d(t)\,\mathrm{d}t\,;$$

加速度法为

$$a(t)=\langle\psi(x,t)|-\frac{\mathrm{d}V(x)}{\mathrm{d}x}+E(t)|\psi(x,t)\rangle\,,\quad A(\omega)=\frac{1}{\sqrt{2\pi}}\int_0^{t_f}\mathrm{e}^{-\mathrm{i}\omega t}a(t)\,\mathrm{d}t\,.$$

(4) 通过时频分析给出具有时间分辨率的谐波功率谱分布 $\log|A(t,\,\omega)|^2$，该谱通过加窗傅里叶变换的方法取得：

$$A(t_0,\,\omega)=\int_0^{t_f}a(t)W_{t_0,\omega}(t)\,\mathrm{d}t\,,\quad W_{t_0,\omega}(t)=\mathrm{e}^{-\mathrm{i}\omega t}\mathrm{e}^{-\frac{(t-t_0)^2}{2\sigma^2}}.$$

注：本题目第一稿只完成了第 (1) 问与第 (2) 问布居数演化图；第 (2) 问动量谱图和第 (3)、(4) 问均是在这一稿加上的.

1.1 解题思路

本题要用到计算物理知识的有两点：求解定态薛定谔方程与实现解薛定谔方程的演化. 在离散的空间、时间格点下，定态薛定谔方程的求解实际为一个三对角矩阵的本征向量和本征值的求解，而我们只需求出本征值最小的本质向量即可. 在给定参考值 -0.48 的情形下，可以用反幂法求得距离该值最近的本征值及相应本征向量. 薛定谔方程的演化则采用 Crank–Nicolson 近似，可保证传递误差仅为 $\mathcal{O}(\Delta t^3)$. 除此两点以外，其他的步骤都可由简单计算实现.

1.2 算法提纲

我们分反幂法与 Crank–Nicolson 求解偏微分方程两点来具体介绍.

反幂法求三对角矩阵本征向量与本征值

反幂法的思路是，从一个初始的向量 $\boldsymbol{x}_0$ 出发，不断求解 $\boldsymbol{A}\boldsymbol{x}_k = \boldsymbol{x}_{k-1}$ 的方程并迭代，可以使 $\boldsymbol{x}$ 当中绝对值最小本征值对应本征向量 $\boldsymbol{v}_1$ 所占组分越来越大，经历足够多次以后可认为 $\boldsymbol{x}$ 已经仅由 $\boldsymbol{v}_1$ 构成. 用此方法可以求任意给定矩阵绝对值最小的本质值及相应的本征向量.

本题哈密顿量在空间格点离散化后离散化后是恰好是一个 n 阶三对角矩阵，因此上述矩阵方程可用追赶法求解，时间复杂度仅为 $\mathcal{O}(n)$. 需要注意，为求得本征值最接近 -0.48 的本征值，我们实际需要将矩阵 $H + 0.48I$ 代入反幂法求解. 如上所述，反幂法的伪代码为（参考讲义格式，下式 $\overline{\max}(\boldsymbol{v})$ 为 $\boldsymbol{v}$ 绝对值最大的分量的值 (保留符号)）

$\boldsymbol{u} = \boldsymbol{v}$
while (不满足判停标准) **do**
    追赶法求解 $\boldsymbol{A}\boldsymbol{v} = \boldsymbol{u}$
    $\lambda = 1/\overline{\max}(\boldsymbol{v})$ (为最小特征值近似值)
    $\boldsymbol{u} = \boldsymbol{v}\lambda_n$ (规格化处理)
end while
$\boldsymbol{u}$ 即为规格化的主特征向量

对于追赶法，我们需要输入 n 阶三对角矩阵 $\boldsymbol{A}$ 的-1, 0, 1 级对角向量 $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$，以及的伪代码是列向量 $\boldsymbol{f}$，可用如下伪代码求得 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{f}$ 的解 $\boldsymbol{x}$：

for $i = 2,\, 3, \cdots,\, n$

$\quad m_i = a_i / b_i$

$\quad b_i = b_i - m_i c_{i-1}$

$\quad f_i = f_i - m_i / f_{i-1}$

end for

$x_n = f_n / b_n$

for $i = n-1,\, n-2, \cdots,\, 1$

$\quad x_i = (f_i - c_i x_{i+1} / b_i)$

end for

将其插入到反幂法中“追赶法求解 $\boldsymbol{A}\boldsymbol{v} = \boldsymbol{u}$”的步骤当中即可. 结合以上二者的求解本征值的步骤封装为函数 `eigen_solver`.

Crank–Nicolson 方法求解偏微分方程

求解含时薛定谔方程的演化本质是偏微分方程求解问题，Crank–Nicolson 的本质思想是对离散化时间格点上波函数的演化

$$\psi(x,\, t + \Delta t) = U(t + \Delta t,\, t)\, \psi(x,\, t) \approx \mathrm{e}^{-\mathrm{i}\Delta t H(t + \frac{\Delta t}{2})}\, \psi(x,\, t)$$

作以下近似

$$\mathrm{e}^{\mathrm{i}H\Delta t} = \frac{1 - \frac{1}{2}\mathrm{i}H\Delta t}{1 + \frac{1}{2}\mathrm{i}H\Delta t} + \mathcal{O}\big(\Delta t^3\big).$$

因此，从 $\psi(x,\, t)$ 推下一步波函数 $\psi(x,\, t + \Delta t)$ 需要求解

$$\left(1 + \frac{1}{2}\mathrm{i}H\Delta t\right) \psi(x,\, t + \Delta t) = \left(1 - \frac{1}{2}\mathrm{i}H\Delta t\right) \psi(x,\, t),$$

而 $\big(1 + \frac{1}{2}\mathrm{i}H\Delta t\big)$ 在离散化空间格点下也是三对角矩阵，故同样可用追赶法求以上的矩阵方程. 利用上式可对波函数进行每一步的时间演化. 程序中将每一步演化过程封装为一个函数 `time_evolve`.

除此二种方法外，其余步骤均为普通计算. 其中数值积分我们采用最简单的矩形面积相加来实现. 具体计算公式已在“题目简述”中给出，这里不再详述.

1.3 结果与讨论

第 (1) 问

运行程序 `01/Q1_1_eigenstate.py`，很快可得出以下输出

```
和 0.48 的偏差 ΔE = -0.02003440
我们只绘制 [-20, 20] 区间里的波函数啦
```

并得到如图 1 所示的 $x \in [-20, 20]$ 区间内的波函数概率密度分布. 容易看出，该分布正是氢原子的 1s 态. 此外，我们也得知了实际求得的本征能量与 -0.48 的偏差大概为 -0.02003.

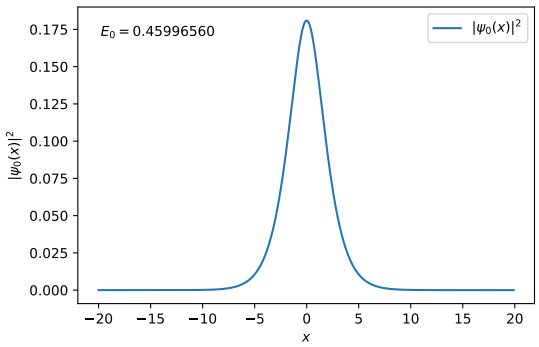

图 1: $x \in [-20, 20]$ 区间内的一维氢原子基态（1s 态）的概率密度分布

第 (2) 问

运行程序 `01/Q1_2_time_evolve.py`，需经过较长时间后才能得到结果. 这是因为每一步演化都需要进行 40001 维的矩阵运算，而我们的时间步长选取为 $\Delta t = 0.01$，相比总时间非常小. 助教师兄在测试时可不妨把 `x_max` 修改为 200 或 20，事实上可以得到非常类似的结果. 对原 $x_{\max} = 2000$ 的情形求得布居数随时间的变化请见图 2，动量谱针对线性纵坐标与对数纵坐标两种情形请见图 3. 就物理含义进行分析，激光结束作用后，电子在基态的布居数约为 0.9，说明部分电子离开的基态.

第 (3) 问

运行程序 `01/Q1_3_logA2.py`，等待程序运行较长时间后得到结果. 在本小问中频率 ω 发生了变化，大约减为原来的 6 倍. 由于影响步长的主要是外加电场中 $\sin \omega t$ 项，故我们可以适当增大步长，来保证 ωt 在三角函数宗量上有相似的采样间隔. 经过不同参数的测试，我们选取了 $\Delta t = 0.5$ 为步长，能够使程序更快地得到结果. 最终用偶极矩法和加速度法获得的两条高次谐波功率谱请见图 4. 可以发现出现了 1 倍、3 倍、5 倍 $\cdots$ 激光频率的谐波，且峰值功率逐渐递减.

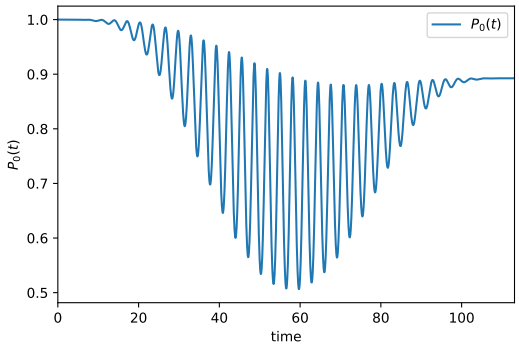

图 2: 氢原子 1s 态布居数随时间演化 $P_0(t) = |\langle\psi_0(x)|\psi(x,t)\rangle|^2$ 的曲线. 和

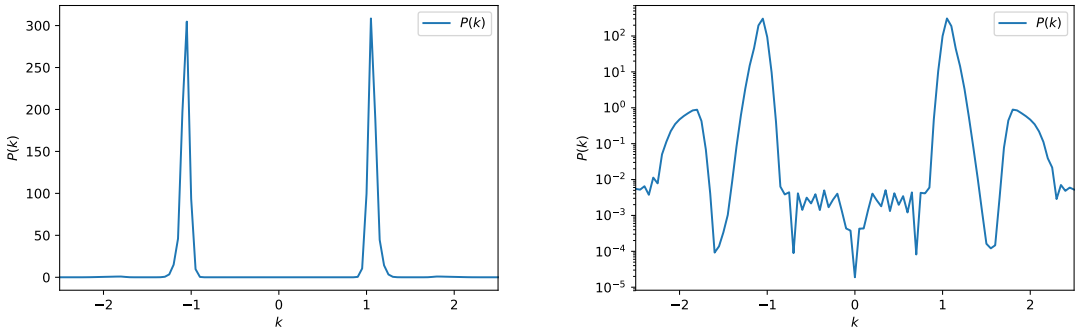

图 3: 在线性纵坐标（左图）和对数纵坐标（右图）上绘制的电离末态电子动量谱 $P(k) = |\langle\psi_k|\psi_f\rangle|^2$ 的曲线.

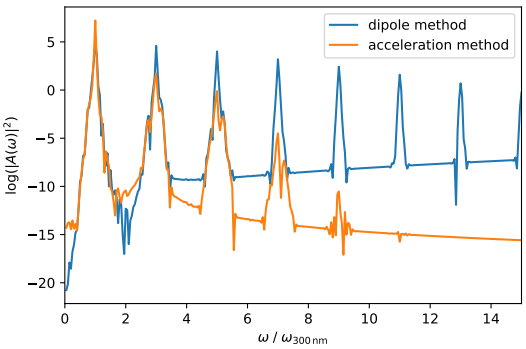

图 4: 偶极矩法和加速度法获得的两条高次谐波功率谱 $\log|A(\omega)|^2$ 曲线.

第 (4) 问

运行程序 `01/Q1_4_logA2tome.py`，可等待较短时间得到结果. 按照上文的思路，可以对改变时间步长，故我们测试了步长分别为 $\Delta t = 0.5$ 和 0.1 的结果，获得的具有

时间分辨率的谐波功率谱分布 $\log|A(t,\omega)|^2$ 的二维等高线图请见图 5. 可以发现二者间差异很小，故选取 $t=0.5$ 所引起的误差不大. 至此，本题目完全求解完毕.

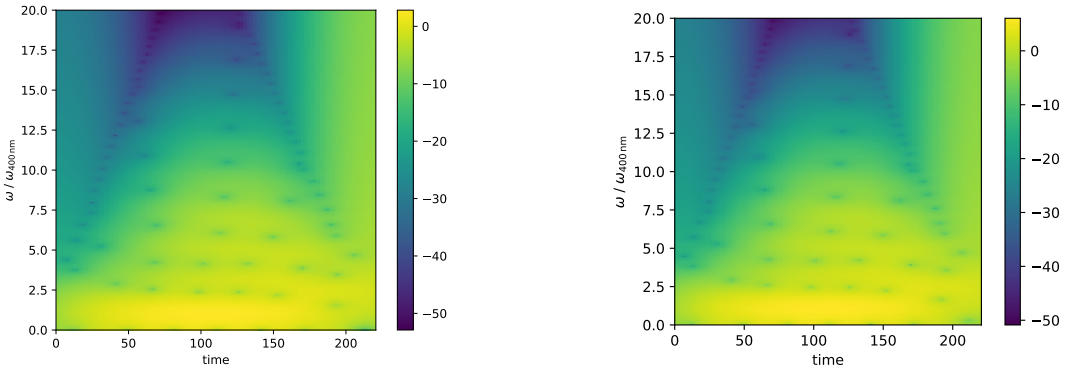

图 5: 具有时间分辨率的谐波功率谱分布 $\log|A(t,\omega)|^2$ 在选取时间步长 $\Delta t=0.5$（左图）与 $\Delta t=0.1$（右图）下的二维等高线图，其中横轴代表时间，纵轴为高次谐波频率与激光频率之比.

1.4 源代码

第 (1) 问代码

第 (1) 问代码：`01/Q1_2_time_evolve.py`

```
# -*- coding: utf-8 -*-
from cmath import sin, cos, exp, sqrt, pi #
    导入常用函数，但要从复数库里导入哦!

def eigen_solver(a, b, c, f): ## 求三对角矩阵特征向量，a,b,c
    为-1,0,1级对角线
    N = len(b)
    m = a.copy()
    for i in range(1, N): # 用追赶法求三对角矩阵方程
        m[i] = a[i] / b[i-1]
        b[i] -= m[i] * c[i-1]
    Nloop = 0
    f_pre = f.copy()
    while True: # 不断对方程的解迭代
```

```
        for i in range(1, N): # 追
            f[i] -= m[i] * f[i-1]
        f[N-1] /= b[N-1]
        for i in range(N-2, -1, -1): # 赶
            f[i] = (f[i] - c[i] * f[i+1]) / b[i]
        # 此时 f[i] 是方程的解
        f_abs_max = -1
        for i in range(N):
            if abs(f[i]) > abs(f_abs_max):
                f_abs_max = f[i] # 找到|f|最大的项
        for i in range(N):
            f[i] /= (f_abs_max) # 规格化矢量
        if sum([(abs(f_pre[i] - f[i]))**2 for i in range(N)]) < 1e-6: #
    两次迭代足够接近->判停
            break
        else:
            f_pre = f.copy()
            Nloop += 1
    return 1/(f_abs_max), f # 返回: 本征值 和 本征矢

################################################################################
## 开始第(1)小问
if __name__ == '__main__':
    x_max, dx = 2000, 0.1
    Nx = int(x_max / dx)
    x = [dx*i for i in range(-Nx, Nx+1)] # x的格点
    # 初始化-1,0,1级对角
    a = [0] + [-0.5/dx**2 for i in range(len(x)-1)]
    c = [-0.5/dx**2 for i in range(len(x)-1)]+ [0]
    b = [1./dx**2 -1 / sqrt(((i-Nx)*dx)**2 + 2) + 0.48  for i in
   range(len(x))]
    lam, u = eigen_solver(a, b, c, [1 for i in range(len(x))]) # 求解!
    print('和 0.48 的偏差 ΔE = %.8f'%lam.real)

    unorm = sqrt(sum([abs(u[i])**2 for i in range(len(x))]))
    for i in range(len(x)):
```

```
        u[i] /= unorm
    u0 = u.copy() # 拷贝一下初始波函数

    ## 概率密度 (*1/dx是考虑到格点归一化与实际归一化不同)
    u_prob = [(abs(ux)**2/dx).real for ux in u]

    ###########################################################################
    ## 使用 matlibplot 作图
    from matplotlib import pyplot as plt
    print('我们只绘制 [-20, 20] 区间里的波函数啦')
    from matplotlib import pyplot as plt
    fig, ax = plt.subplots()
    plt.plot(x[int(len(x)/2)-200:int(len(x)/2)+200],
    u_prob[int(len(x)/2)-200:int(len(x)/2)+200],
            label=r'$|\psi_0(x)|^2$')
    plt.xlabel(r'$x$')
    plt.ylabel(r'$|\psi_0(x)|^2$')
    plt.legend()
    plt.text(0.05, 0.9, r'$E_0=%.8f$'%(0.48+lam.real),
    transform=ax.transAxes)
    plt.savefig('Q1_1_psi0.pdf')
```

第 (2) 问代码

第 (2) 问代码: `01/Q1_1_eigenstate.py`, 这里仅展示除了与第 (1) 问相同的 `eigen_solver` 函数的部分.

```
## 新函数
def time_evolve(u, u_): # 按薛定谔方程演化一步的函数 u->u_
    for i in range(N): # 此时刻 H 的对角元 b[i], 为后文做准备
        b[i] = 1./dx**2 -1 / sqrt(((i-Nx)*dx)**2 + 2) +
    (i-Nx)*dx*E0*sin(ome*t/(2*Npulse))**2*sin(ome*t)
    ## 先求 u_=(1-1/2iHΔt) u
    u_[0] = (1 - 0.5j*dt*b[i]) * u[0] - 0.5j*dt*c[i] * u[1]
    u_[N-1] = - 0.5j*dt*a[i] * u[N-2] + (1 - 0.5j*dt*b[i]) * u[N-1]
    for i in range(1, N-1):
```

```
        u_[i] = - 0.5j*dt*a[i] * u[i-1] + (1 - 0.5j*dt*b[i]) * u[i] -
    0.5j*dt*c[i] * u[i+1]
    ## 再用追赶法求 (1+1/2iHΔt) u_' = u_
    be[0] = (1 + 0.5j*dt*b[0])
    for i in range(1, N): # 追
        be[i] = (1 + 0.5j*dt*b[i])
        m[i] = 0.5j*dt*a[i] / be[i-1]
        be[i] -= m[i] * 0.5j*dt*c[i-1]
        u_[i] -= m[i] * u_[i-1]
    u_[N-1] /= (1 + 0.5j*dt*b[i-1])
    for i in range(N-2, -1, -1): # 赶
        u_[i] = (u_[i] - 0.5j*dt*c[i] * u_[i+1]) / be[i]
    ## 此时u_[i] 就是演化下一刻的波函数

    ## 吸收虚假信号
    for i in range(N):
        if abs((i-Nx)*dx) > 0.75*x_max:
            u_[i] *= exp(-(abs((i-Nx)*dx)-0.75*x_max)**2/0.04)
    ## 归一化
    unorm = sqrt(sum([abs(u_[i])**2 for i in range(N)]))
    for i in range(N):
        u_[i] /= unorm

################################################################################
## 开始第(2)小问
if __name__ == '__main__':
    x_max, dx = 2000, 0.1
    Nx = int(x_max / dx)
    x = [dx*i for i in range(-Nx, Nx+1)] # x的格点
    # 初始化-1,0,1级对角
    a = [0] + [-0.5/dx**2 for i in range(len(x)-1)]
    c = [-0.5/dx**2 for i in range(len(x)-1)]+ [0]
    b = [1./dx**2 -1 / sqrt(((i-Nx)*dx)**2 + 2) + 0.48  for i in
    range(len(x))]
    lam, u = eigen_solver(a, b, c, [1 for i in range(len(x))]) # 求解!
    print('和 0.48 的偏差 ΔE = %.8f'%lam.real)
```

```
unorm = sqrt(sum([abs(u[i])**2 for i in range(len(x))]))
for i in range(len(x)):
    u[i] /= unorm
u0 = u.copy() # 拷贝一下初始波函数
## 以上是一样的内容: 求初始的氢原子 1s 态波函数

#############################################################################

E0, Npulse = sqrt(1 / 3.5094448314), 18
ome = 1
T = 2*Npulse*pi / ome
dt = 0.05
Nt = int(T/dt)
t_list = [i/Nt*T for i in range(Nt+1)]
t = dt / 2
N = len(x)
u = u0.copy() # 储存初始的 u
u_ = u.copy() # 用作储存演化下一刻的 u
m, be = a.copy(), a.copy()
P0t = [abs(sum([u0[i].conjugate()*u[i] for i in range(N)]))**2] #
储存 P_0(t)
u_list = [u[:]]
print('即将开始演化波函数, 真的很慢, 作为测试可尝试将 x_max
改为200或20')
for it in range(len(t_list)): ## 开始时间演化
    time_evolve(u, u_) # 按时间演化一步
    u, u_ = u_, u # 直接交换两个变量引用, 省储存空间的做法
    t += dt
    P0t.append(abs(sum([u0[i].conjugate()*u[i] for i in
range(N)]))**2)
    u_list.append(u[:])
    if it%100 == 0:
        print('正在演化波函数, 已经进行 %4d / %4d 步'%(it,
len(t_list)))
```

```
## 调取末态波函数, 计算 k 上投影
u_fin = u_list[-2] # 其实最后多演化了一步, 调取u_list[-2]为末态波函数
p_fin = sum([u0[i].conjugate()*u_fin[i] for i in range(N)]) #
基态投影系数
u_fin = [u_fin[i] - p_fin*u0[i] for i in range(len(u_fin))] #
减去基态函数

## 下面考虑在 k 本征态投影
k_max, dk = 2.5, 0.01
Nk = (int)(k_max/dk)
k = [i*dk for i in range(-Nk, Nk+1)]
## 下式里 _k*=1/sqrt(dx) exp(-ikx),
除以sqrt(dx)目的是考虑到x格点的归一化
Pk = [abs(sum([(1/sqrt(dx)*exp(-1j*k[ik]*(i-Nx)*dx))*u_fin[i] for i
in range(N)]))**2 for ik in range(len(k))]



##########################################################################
## 使用 matlibplot 作图
from matplotlib import pyplot as plt
## 先画 1s 态布居数随时间演化
plt.plot(t_list, P0t[:-1], label=r'$P_0(t)$')
plt.xlabel('time')
plt.ylabel(r'$P_0(t)$')
plt.xlim([t_list[0], t_list[-1]])
plt.legend()
plt.savefig('Q1_2_P0t.pdf')
plt.show()

## 再画末态动量谱
plt.plot(k, Pk, label=r'$P(k)$')
plt.xlabel(r'$k$')
plt.ylabel(r'$P(k)$')
plt.xlim([-k_max, k_max])
plt.legend()
plt.savefig('Q1_2_Pk.pdf')
```

```
    plt.show()
    plt.plot(k, Pk, label=r'$P(k)$')
    plt.yscale('log')
    plt.xlabel(r'$k$')
    plt.ylabel(r'$P(k)$')
    plt.xlim([-k_max, k_max])
    plt.legend()
    plt.savefig('Q1_2_Pk_logy.pdf')
    plt.show()
```

第 (3) 问代码

第 (3) 问代码: `01/Q1_3_logA2.py`,这里仅展示除了与第 (2) 问相同的 `eigen_solver`, `time_evolve` 函数的部分.

```
## 开始第(3)小问
if __name__ == '__main__':
    x_max, dx = 2000, 0.1
    Nx = int(x_max / dx)
    x = [dx*i for i in range(-Nx, Nx+1)] # x的格点
    # 初始化-1,0,1级对角
    a = [0] + [-0.5/dx**2 for i in range(len(x)-1)]
    c = [-0.5/dx**2 for i in range(len(x)-1)]+ [0]
    b = [1./dx**2 -1 / sqrt(((i-Nx)*dx)**2 + 2) + 0.48  for i in
   range(len(x))]
    lam, u = eigen_solver(a, b, c, [1 for i in range(len(x))]) # 求解!
    print('和 0.48 的偏差 ΔE = %.8f'%lam.real)

    unorm = sqrt(sum([abs(u[i])**2 for i in range(len(x))]))
    for i in range(len(x)):
        u[i] /= unorm
    u0 = u.copy() # 拷贝一下初始波函数
    ## 以上是一样的内容: 求初始的氢原子 1s 态波函数

   ###########################################################################

    E0, Npulse = sqrt(0.02 / 3.5094448314), 48 ## 注意参数改了! !
```

```
ome = 45.5633525316 / 300
T = 2*Npulse*pi / ome
dt = 0.5
Nt = int(T/dt)
t_list = [i/Nt*T for i in range(Nt+1)]
t = dt / 2
N = len(x)
u = u0.copy() # 储存初始的 u
u_ = u.copy() # 用作储存演化下一刻的 u
m, be = a.copy(), a.copy()
# 分别记 d 和 a 随时间演化
d_list = [sum([u[i].conjugate()*((i-Nx)*dx)*u[i] for i in range(N)])]
a_list = [sum([u[i].conjugate()*(-((i-Nx)*dx)/(((i-Nx)*dx)**2 +
2)**(3/2) + E0*sin(ome*t/(2*Npulse))**2*sin(ome*t))*u[i]
                    for i in range(N)])]
print('即将开始演化波函数，真的很慢，作为测试可尝试将 x_max
改为200或20')
for it in range(len(t_list)): ## 开始时间演化
    time_evolve(u, u_) # 按时间演化一步
    u, u_ = u_, u # 直接交换两个变量引用，省储存空间的做法
    t += dt
    d_list.append(sum([u[i].conjugate()*((i-Nx)*dx)*u[i] for i in
range(N)]))
    a_list.append(sum([u[i].conjugate()*(-((i-Nx)*dx)/(((i-Nx)*dx)**2
+ 2)**(3/2) + E0*sin(ome*t/(2*Npulse))**2*sin(ome*t))*u[i]
                        for i in range(N)]))
    if it%100 == 0:
        print('正在演化波函数，已经进行 %4d / %4d 步'%(it,
len(t_list)))

## 两种方法计算高次谐波功率
Nome = 20 # 共 15*Nome个 上采样点
ome_list = [i*ome/Nome for i in range(15*Nome+1)]
# 偶极矩法
A_d = [1/sqrt(2*pi)*(-o**2)*sum([exp(-1j*o*(it/Nt*T))*d_list[it] for
it in range(len(t_list))]) for o in ome_list]
```

```
    # 加速度法
    A_a = [1/sqrt(2*pi)*sum([exp(-1j*o*(it/Nt*T))*a_list[it] for it in
    range(len(t_list))]) for o in ome_list]



    ##########################################################################
    ## 使用 matlibplot 作图
    from matplotlib import pyplot as plt
    plt.plot([o/ome for o in ome_list[1:]], [log(abs(A_d[i])**2) for i in
    range(1, len(A_d))], label='dipole method')
    plt.plot([o/ome for o in ome_list[1:]], [log(abs(A_a[i])**2) for i in
    range(1, len(A_a))], label='acceleration method')
    plt.xlabel(r'$\omega\;/\;\omega_{300\,\mathrm{nm}}$')
    plt.ylabel(r'$\log(|A(\omega)|^2)$')
    plt.xlim([0, 15])
    plt.legend()
    plt.savefig('Q1_3_logA2.pdf')
    plt.show()
```

第 (4) 问代码

第 (4) 问代码: `01/Q1_3_logA2tome.py`, 这里仅展示除了与第 (2) 问相同的 `eigen_solver`, `time_evolve` 函数的部分.

```
## 开始第(4)小问
if __name__ == '__main__':
    x_max, dx = 600, 0.1 # 改成600啦!
    Nx = int(x_max / dx)
    x = [dx*i for i in range(-Nx, Nx+1)] # x的格点
    # 初始化-1,0,1级对角
    a = [0] + [-0.5/dx**2 for i in range(len(x)-1)]
    c = [-0.5/dx**2 for i in range(len(x)-1)]+ [0]
    b = [1./dx**2 -1 / sqrt(((i-Nx)*dx)**2 + 2) + 0.48  for i in
    range(len(x))]
    lam, u = eigen_solver(a, b, c, [1 for i in range(len(x))]) # 求解!
    print('和 0.48 的偏差 ΔE = %.8f'%lam.real)
```

```
unorm = sqrt(sum([abs(u[i])**2 for i in range(len(x))]))
for i in range(len(x)):
    u[i] /= unorm
u0 = u.copy() # 拷贝一下初始波函数
## 以上是一样的内容: 求初始的氢原子 1s 态波函数

#########################################################################

E0, Npulse = sqrt(0.01 / 3.5094448314), 4 ## 注意参数改了! !
ome = 45.5633525316 / 400
T = 2*Npulse*pi / ome
dt = 0.5
Nt = int(T/dt)
t_list = [i/Nt*T for i in range(Nt+1)]
t = dt / 2
N = len(x)
u = u0.copy() # 储存初始的 u
u_ = u.copy() # 用作储存演化下一刻的 u
m, be = a.copy(), a.copy()
# 记 a 随时间演化
a_list = [sum([u[i].conjugate()*(-((i-Nx)*dx)/(((i-Nx)*dx)**2 +
2)**(3/2) + E0*sin(ome*t/(2*Npulse))**2*sin(ome*t))*u[i]
                   for i in range(N)])]
print('即将开始演化波函数, 这个还比较快')
for it in range(len(t_list)): ## 开始时间演化
    time_evolve(u, u_) # 按时间演化一步
    u, u_ = u_, u # 直接交换两个变量引用, 省储存空间的做法
    t += dt
    a_list.append(sum([u[i].conjugate()*(-((i-Nx)*dx)/(((i-Nx)*dx)**2
+ 2)**(3/2) + E0*sin(ome*t/(2*Npulse))**2*sin(ome*t))*u[i]
                       for i in range(N)]))
    if it%100 == 0:
        print('正在演化波函数, 已经进行 %4d / %4d 步'%(it,
len(t_list)))

## 计算高次谐波功率
```

```
print('正在计算高次谐波功率, 请稍后')
Nome, Nt0, sigma = 20, 100, 15 # 共 20*Nome=400个 上采样点,
Nt0=100个t0上采样点
ome_list = [i*ome/Nome for i in range(20*Nome+1)]
t0_list = [i*T/Nt0 for i in range(Nt0+1)]
# 计算 A(t0,):
logA2 = [[log(abs(
sum([exp(-1j*o*(it/Nt*T))*exp(-(it/Nt*T-t0)**2/(2*sigma**2))*a_list[it]
for it in range(len(t_list)-1, -1, -1)]) ).real**2).real
           for t0 in t0_list] for o in ome_list]



##############################################################################
## 使用 matlibplot 作图
from matplotlib import pyplot as plt
im = plt.imshow(logA2, extent=(0, T, 0, 20), aspect=T/20)
plt.colorbar(im)
plt.xlabel('time')
plt.ylabel(r'$\omega\;/\;\omega_{400\,\mathrm{nm}}$')
plt.savefig('Q1_4_logA2tome_t0.1.pdf')
plt.show()
```

2 激光场中氢原子电离动力学的半径典模拟

题目简述：使用半径典模型描述氢原子在强场中的电离过程. 物理背景：在施加强场中的任何时间 t 电子都有一定几率隧穿出势垒成为自由电子，此后将在库仑力和激光电场力共同作用下做经典运动. 电子运行到无穷远处可被探测器接收，进而统计电离电子的动量分布. 施加激光分线偏振与圆偏振两种激光，矢势分别为

$$\boldsymbol{A}(t) = f(t)A_0 \sin\omega t\, \boldsymbol{e}_x$$
$$\boldsymbol{A}(t) = f(t)A_0 \left[\frac{1}{\sqrt{1+0.5^2}} \sin\omega t\, \boldsymbol{e}_x + \frac{0.5}{\sqrt{1+0.5^2}} \cos\omega t\, \boldsymbol{e}_y\right]$$

其中 $f(t) = \cos^2\left(\frac{\pi}{2}\frac{t}{2T}\right)$，脉冲时间 $-2T \le t \le 2T$. 取 $\omega = 0.057$, $T = 110.23$, $A_0 = 1.325$，电场由 $\boldsymbol{E} = -\partial\boldsymbol{A}/\partial t$ 计算. 数值解题的步骤如下：

(1) 在 $-2T \le t \le 2T$ 时间内电子可能发生隧穿. 隧穿电子位置 $|r_0| = I_p/E$ 沿电场反方向，无平行电场速度分量，隧穿几率

$$W \propto \frac{1}{E^2} \exp\left(-\frac{2}{3E}\right) \exp\left(-\frac{v_\perp^2}{E}\right)$$

利用蒙特卡洛方法生成电子样品.

(2) 在电子隧穿到 $t = 2T$ 时间内用牛顿方程解电子在库仑力和电场力共同作用下的运动. 只考虑二维平面运动，取软化库伦势. 最终得到激光结束电子位置 $\boldsymbol{r}_f$ 和速度 $\boldsymbol{v}_f$.

(3) 激光结束后电子只受库仑力作用. 用三个守恒量：能量、角动量、隆格楞次矢量得到无穷远渐进动量

$$\boldsymbol{p}_\infty = p_\infty \frac{p_\infty(\boldsymbol{L}\times\boldsymbol{a} - \boldsymbol{a})}{1 + p_\infty^2 L^2}$$

只考虑能量大于 *0* 的电子，且无需对库仑力软化.

(4) 记录 $-1.5 \le p_x \le 1.5$, $-1.5 \le p_y \le 1.5$ 范围内以 $\Delta p_x = \Delta p_y = 0.02$ 为格点间距的末态动量分布.

最后要求的结果：线偏光和椭圆偏振光的上述 $p_x - p_y$ 散点图分布，以及两个方向投影的线动量分布. 从线动量分布给出峰值动量大小.

注：本题的所有最终结果图和程序均已在第一稿给出，第二稿未做变动.

2.1 解题思路

本题看似诸多小问，其实只要得到最后结果即可. 求解过程中要使用以下计算物理的方法：

(1) 蒙特卡洛法生成散点的样本. 这里我们选用第一类舍选法：首先在 $-2T \leq t \leq 2T$ 范围内均匀取随机点，根据 $E(t)$ 及在 $v_\perp$ 上积分后的概率 $\int W(E, v_\perp)\,\mathrm{d}v_\perp$ 确定是否保留. 保留的即可作为样品，算出其位置坐标 $\boldsymbol{r}_0$ 和速度 $\boldsymbol{v}_0$，存入样本中.

(2) 四阶隆格库塔（R-K4）算法求解牛顿方程. 根据隧穿后 (x, y), (p_x, p_y) 按照所受软化库仑力与电场力计算出激光结束时位置 $\boldsymbol{r}_f$ 和速度 $\boldsymbol{v}_f$.

(3) $\boldsymbol{r}_f$ 和速度 $\boldsymbol{v}_f$ 代入公式求守恒量（舍去能量小于 0 的样品），进而求末态动量 $\boldsymbol{p}_\infty$，绘图.

下节将分这三步详述算法的实现.

2.2 算法提纲

第 (1) 步：蒙特卡洛生成电子样品

首先要求解不同时间下电子隧穿概率，为此要积掉隧穿概率中 $v_\perp$ 的变量. 容易看出 $v_\perp$ 呈高斯分布，故

$$\begin{aligned} W_E(E) &\propto \int_{-\infty}^{\infty} W(E, v_\perp)\,\mathrm{d}v_\perp = \int_{-\infty}^{\infty} \frac{1}{E^2} \exp\left(-\frac{2}{3E}\exp\left(-\frac{v_\perp^2}{E}\right)\right)\mathrm{d}v_\perp \\ &\propto \frac{1}{E^{3/2}} \exp\left(-\frac{2}{3E}\right) \end{aligned}$$

用第一类舍选法每一次生成样品的算法是：

(i) 生成随机数 $r \in [0, 1)$，利用 $t = 4T(r - \frac{1}{2})$ 转化为 $[-2T, 2T)$ 上随机数

(ii) 事先绘制好 $E(t) = \sqrt{E_x^2(t) + E_y^2(t)}$ 对线偏和椭圆偏振下的曲线（请见图 6），求得相应的最大值 $E_{\max} = 0.08$，相应概率密度曲线的最大值约为 $W_{E\max} = 0.010623$ (详见下节).

(iii) 生成另一随机数 $\xi \in [0, 1)$，判断是否 $\xi W_{E\max} < W_E(|\boldsymbol{E}(t)|)$. 若是则存入该样品合格，继续下步；不是重复该步直到找到合格的样品.

(iv) 利用二维变换抽样法进行高斯抽样，得到垂直速度 $v_\perp$. 具体为，取两个独立 $[0,1)$ 区间随机变量 u, v，计算

$$v_\perp = \sqrt{-E\,\ln u}\,\cos(2\pi v)$$

根据理论可知 $v_\perp$ 符合 $W_{v_\perp} = \frac{1}{\sqrt{\pi E}}\exp\left(-\frac{v_\perp^2}{E}\right)$ 的高斯型概率密度分布.

(v) 根据此时 $\boldsymbol{E}(t)$ 计算该电子样品隧穿后初始位置与初始速度

$$(x_0,\, y_0) = \left(-\frac{E_x\, I_p}{E^2},\, -\frac{E_y\, I_p}{E^2}\right), \quad (v_{x0},\, v_{y0}) = \left(-\frac{E_y}{E}v_\perp,\, \frac{E_x}{E}v_\perp\right)$$

将样品的 5 个参数 $(x_0,\, y_0,\, v_{x0},\, v_{y0},\, t_0)$ 存储即可. 随后可进行下一次采样.

重复上述步骤 N 次,即成功获取了 N 个样品. 本步骤的具体实现请参见 `02/Q2_all.py` 程序中的 `gen_sample` 函数.

第 (2) 步：R-K4 算法求解牛顿方程

本例采用定步长的四阶隆哥库塔算法 (R-K4) 求解牛顿动力学方程. 原始方程为 (采用原子单位制 $\hbar = m_e = e = 1$)

$$\ddot{\boldsymbol{r}} = -\boldsymbol{E} - \frac{\boldsymbol{r}}{r^3}$$

为方便 R-K4 算法的使用，定义向量 $\boldsymbol{y} = (x,\, y,\, v_x,\, v_y)$，方程写作

$$\begin{cases} \dot{x} = v_x \\ \dot{y} = v_y \\ \dot{v_x} = -E_x - \frac{x}{r^3} \\ \dot{v_y} = -E_y - \frac{y}{r^3} \end{cases}$$

需要注意这里 r 使用软化库伦势下的距离 $r = \sqrt{x^2 + y^2 + 0.2^2}$. 求解方程 $\boldsymbol{y}' = \boldsymbol{f}(t,\, \boldsymbol{y})$ 的隆格库塔算法的伪代码为:

while True

$\boldsymbol{k}_1 = \boldsymbol{f}(t_{n-1},\, \boldsymbol{y}_{n-1})$

$\boldsymbol{k}_2 = \boldsymbol{f}(t_{n-1} + \frac{\Delta x_n}{2},\, \boldsymbol{y}_{n-1} + \frac{\Delta t_n}{2}\boldsymbol{k}_1)$

$\boldsymbol{k}_3 = \boldsymbol{f}(t_{n-1} + \frac{\Delta x_n}{2},\, \boldsymbol{y}_{n-1} + \frac{\Delta t_n}{2}\boldsymbol{k}_2)$

$\boldsymbol{k}_4 = \boldsymbol{f}(t_{n-1},\, \boldsymbol{y}_{n-1} + \Delta t_n \boldsymbol{k}_3)$

$\boldsymbol{y}_n = \boldsymbol{y}_{n-1} + \frac{\Delta t_n}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4)$

$t_n = t_{n-1} + \Delta t_n$

if t_n 达到目标点: **break**

\# 开始下一步的计算

end while

本例中我们求解的是从隧穿时刻 t_0 到激光结束时 $t = 2T$ 期间向量 $\boldsymbol{y} = (x,\, y,\, v_x,\, v_y)$ 的演化，容易看出实际上 $\boldsymbol{f}(t,\, \boldsymbol{y})$ 在本例中是与 t 无关的. 本步骤的具体实现请参见 `02/Q2_all.py` 程序中的 `rk4_solver` 函数.

第 (3) 步：代入守恒量公式计算

在 $t > 2T$ 阶段只需简单计算即可求得无穷远动量 $\boldsymbol{p}_\infty$. 上一步末态的向量记为 $(x_f,\, y_f,\, v_{xf},\, v_{yf})$ 使用题目中给的守恒量计算公式，在 $x - y$ 平面上它们具体为

$$\begin{cases} p_\infty^2 = p_f^2 - \frac{2}{r} \\ L_z = y_f p_{xf} - x_f p_{yf} \\ a_x = y_f L_z - \frac{x_f}{\sqrt{x_f^2 + x_f^2}} \\ a_y = -x_f L_z - \frac{y_f}{\sqrt{x_f^2 + x_f^2}} \end{cases}$$

无穷远处的计算式为

$$\boldsymbol{p}_\infty = \left(\frac{-p_\infty^2\, L_z a_y - a_x}{1 + p_\infty^2 L_z^2},\ \frac{p_\infty^2\, L_z a_x - a_y}{1 + p_\infty^2 L_z^2} \right)$$

即为最终结果. 本步骤的具体实现请参见 `02/Q2_all.py` 程序中的 `coulomb_evol` 函数.

第 (4) 步：绘图

随后，我们使用 Python 的 `matplotlib.pyplot` 中 `hist2d` 与 `hist` 函数进行作图即可得到二维的 $p_x - p_y$ 分布与一维的 p_x 与 p_y 分布. 作图部分请参见程序主函数.

2.3 结果与讨论

在展示最终结果之前，又必要对过程中参数的选取做一说明. 首先是舍选法对于概率密度函数最大值的选取. 我们可以使用绘图软件事先做出线偏振和椭圆偏振电场的变化范围，请见图 6. 可认为对两种激光类型而言电场最大值都取 $E_{\max} = 0.08$. 进一步可知 $W_E(E)$ 是递增函数，故 W_E 所能达到最大值为 $W_{E\max} = W_E(0.08) = 0.010623$，用作舍选法参数.

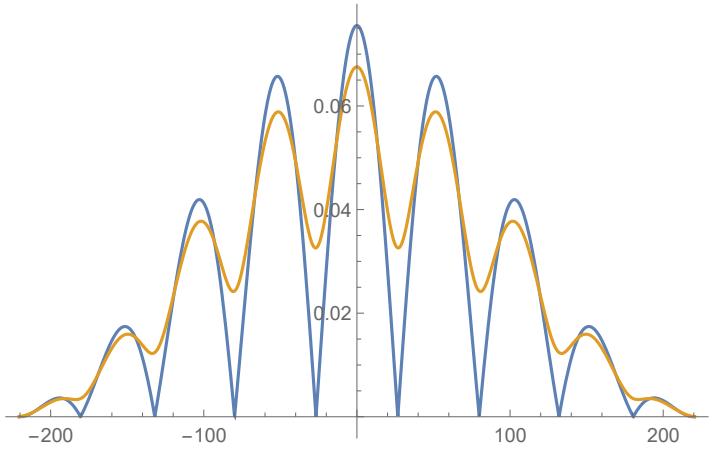

图 6: 线偏振（蓝色）与椭圆偏振（橙色）的电场大小在 $-2T < t < 2T$ 范围内的变化关系. 可以认为最大的电场值有 $E_{\max} = 0.08$.

其次是 R-K4 算法步长的选取. 根据实际的蒙特卡洛样品参数分布，可对某一样品参数 $(x_0, y_0, v_{x0}, v_{y0}, t_0) = (8, 0, 0, 0.5, 0)$ 绘制出不同步长下演化的结果，请见图 7. 由此可以发现步长选取在 $\Delta t = 0.1$ 附近时误差比较小. 但考虑到时间成本的因素，我们最终选择的步长为 $\Delta t = 0.5$. 这样可以保证误差不太大的情形下不必花费过多时间在 R-K4 的计算中.

最后是运行程序的结果. 助教师兄测试时可以通过运行测试代码 `02/Q2_all.py`，得到 1000 组样本下的无穷远动量的分布. 考虑到生成与计算样品的速度较慢，笔者对代码进行了 Python 的多线程的改造（参见文件 `02/Q2_multicore.py`，也可对此进行测试，但需要在非 *Windows* 的 *IDE* 上运行）. 笔者在 LBNL 的 Cori 服务器（24 核 CPU）上分别对线偏振与椭圆偏振获得了 10 000 000 组样品. 最终结果显示出的曲线非常平滑，且与作业提示的结果符合地很好. 其中，线偏振的二维 $p_x - p_y$ 分布、一维 p_x 和 p_y 分布请见图 8；椭圆偏振的情形请见图 9. 可以从图中读出，对线偏振的激光场，p_x 的主峰在 -0.36，次峰在 0.27；p_y 呈现单峰在 -0.03. 对椭圆偏振的情形，p_x 呈现单峰在 -0.12，而 p_y 的主峰在 -0.51，次峰在 0.42. 至此，本题的求解完毕.

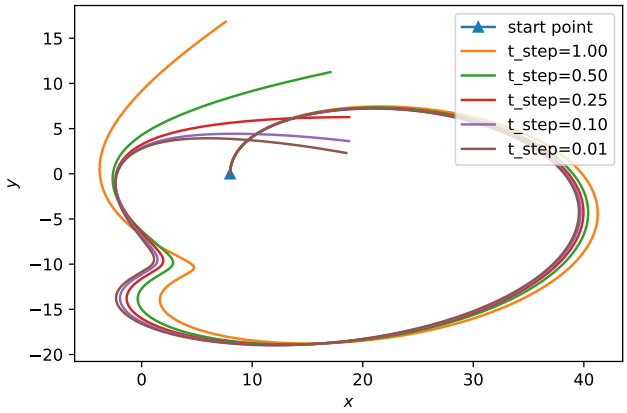

图 7: 对样品参数 $(x_0, y_0, v_{x0}, v_{y0}, t_0) = (8, 0, 0, 0.5, 0)$ 绘制的步长分别为 $\Delta t = 1$, 0.5, 0.25, 0.1, 0.01 时利用 R-K4 算法进行牛顿方程演化的结果

2.4 源代码

测试代码（小样本情形）

对生成 1000 组电子样品的测试代码，请参见：`02/Q2_all.py`

```
from numpy import sqrt, pi, cos, sin, exp, log
from random import random

###################################
# pol = 'linear'
pol = 'elliptical'
###################################

def calc_elec(t, pol): ## 输入给定时刻 t 和偏振模式 pol，输出此时电场 E_x,
    E_y
    omet = ome * t
    if pol == 'linear':
        Ex = -A0*((cos(omet/8))**2*cos(omet) - sin(omet/4)*sin(omet)/8
    )*ome
        Ey = 0
    elif pol == 'elliptical':
        Ex = -A0*((cos(omet/8))**2*cos(omet) - sin(omet/4)*sin(omet)/8
    )*ome*2/sqrt(5)
```

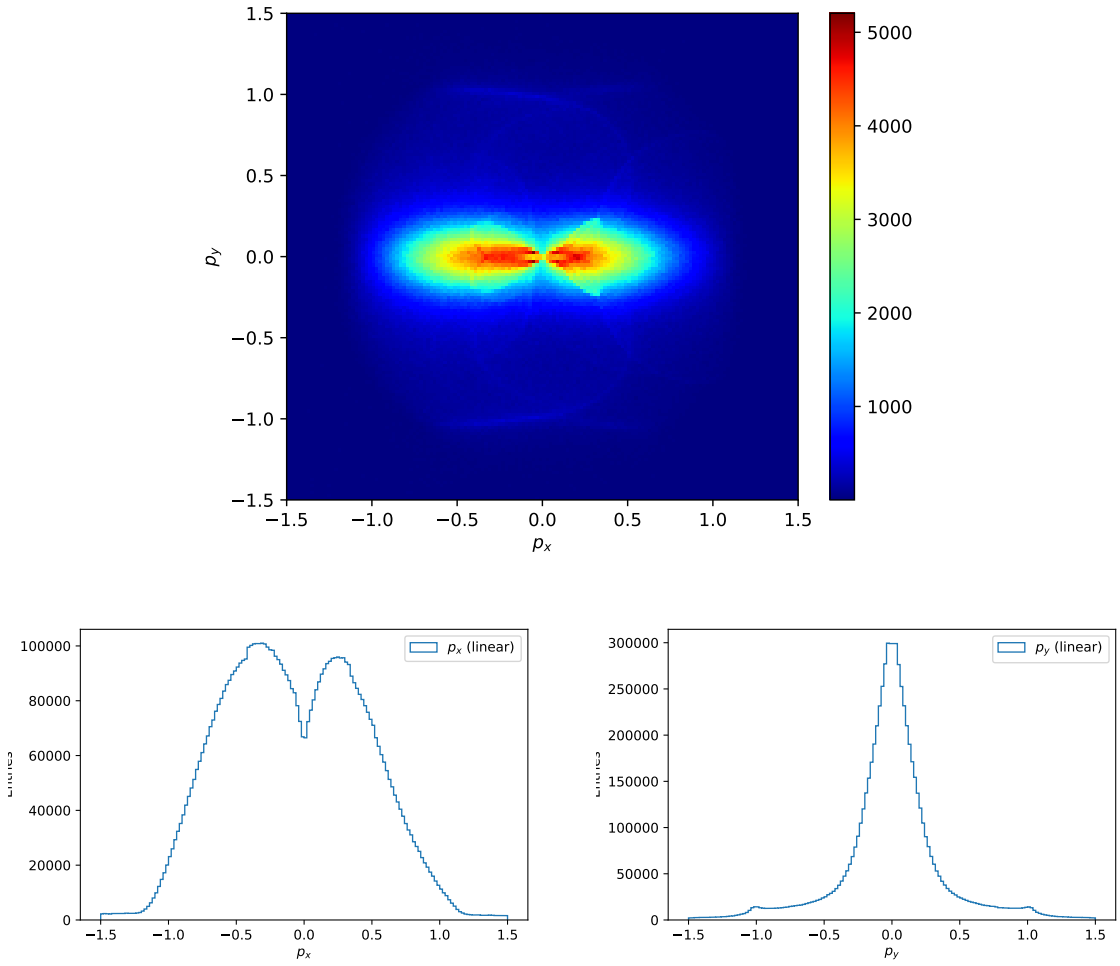

图 8: 线偏振激光场下，用 10 000 000 组电子样品得到的二维 $p_x - p_y$ 分布、一维 p_x 和 p_y 分布图. 其中动量的限制为 $-1.5 < p_x, p_y < 1.5$，间隔的要求为 $\Delta p_x = \Delta p_y = 0.02$，也即共分 150 个 bin.

```
        Ey = A0*((cos(omet/8))**2*sin(omet) + sin(omet/4)*cos(omet)/8
    )*ome/sqrt(5)
    return Ex, Ey

def gen_sample(): ## 舍选法获得一个样品
    while True:
        t = (random()-0.5) * 4*T
        Ex, Ey = calc_elec(t, pol)
        Eabs = sqrt(Ex**2 + Ey**2) # 计算当前电场大小
```

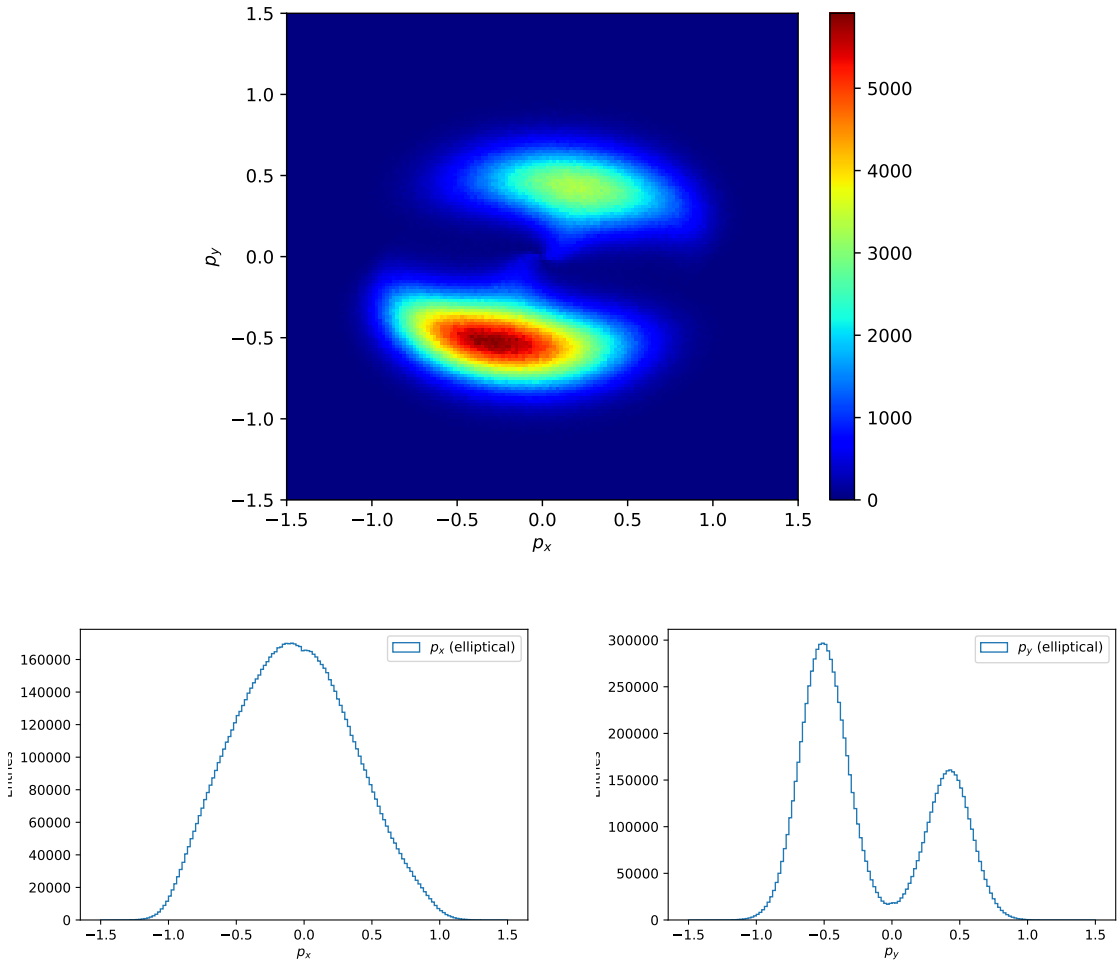

图 9: 椭圆偏振激光场下, 用 10 000 000 组电子样品得到的二维 $p_x - p_y$ 分布、一维 p_x 和 p_y 分布图. 其中动量的限制为 $-1.5 < p_x, p_y < 1.5$, 间隔的要求为 $\Delta p_x = \Delta p_y = 0.02$, 也即共分 150 个 bin.

```
        if WEmax * random() < 1/Eabs**(3/2) * exp(-2/(3*Eabs)): #
    生成随机数是否在概率密度分布以下？
            r = -Ip/Eabs
            u, v = random(), random()
            v_perp = sqrt(-Eabs*log(u)) * cos(2*pi*v)
            return t, [r*Ex/Eabs, r*Ey/Eabs, -v_perp*Ey/Eabs,
    v_perp*Ex/Eabs] # return 格式: t, [x,y,vx,vy], 生成一个样品了

def rk4_solver(t, y, dt): ## 给定微分方程的 R-K4 算法求解器
```

```
    def f(y): # y[0]=x, y[1]=y, y[2]=vx, y[3]=vy
        Ex, Ey = calc_elec(t, pol)
        r3 = sqrt(y[0]**2 + y[1]**2 + 0.04)**3  ## 软化库伦势相应的 r^3
        return [y[2], y[3], -Ex-y[0]/r3, -Ey-y[1]/r3] # 根据微分方程定出 y
    的导函数

    stop = False
    while True: # R-K4 的每次迭代
        if 2*T-t < dt: # 达到判停条件？
            dt = 2*T-t
            stop = True
        k1 = f(y)
        k2 = f([y[i] + dt/2*k1[i] for i in range(4)])
        k3 = f([y[i] + dt/2*k2[i] for i in range(4)])
        k4 = f([y[i] + dt*k3[i] for i in range(4)])
        for i in range(4):
            y[i] += dt/6 * (k1[i] + 2*k2[i] + 2*k3[i] + k4[i])
        t += dt
        if stop == True:
            break
    return t, y
def coulomb_evol(t, par): ## 激光结束后在库伦场运动，格式 par=(x,y,vx,vy)
    r = sqrt(par[0]**2 + par[1]**2)  # 不必软化库伦势
    p2 = (par[2]**2 + par[3]**2) - 2/r
    if p2 <= 0:  # 能量 < 0 则舍去
        return None
    Lz = par[0] * par[3] - par[1] * par[2]  # 角动量 z 分量
    ax = par[3] * Lz - par[0]/r  # 隆格楞次矢量 x 分量
    ay = -par[2] * Lz - par[1]/r  # 隆格楞次矢量 y 分量
    denom = 1 + p2 * Lz**2
    return [(p2*(-Lz*ay) - sqrt(p2)*ax)/denom, (p2*(Lz*ax) -
    sqrt(p2)*ay)/denom]

if __name__ == '__main__':
    A0 = 1.325
    WEmax = 0.010623
```

```
Ip = 0.5


N = 1000
ome = 0.057
T = 2*pi/ome
p_infx, p_infy = [], []
Nloop = 0
for i in range(N):
    t, par = gen_sample() # 先生成样品
    rk4_solver(t, par, dt=0.5) # R-K4演化到激光结束, 步长取为0.5
    p_inf = coulomb_evol(t, par) # 计算无穷远动量
    if p_inf == None: # 能量为负? (没有返回值) , 舍去
        continue
    if i % 100 == 0:
        print('已经生成 %-6d 个样品啦'%i)
    p_infx.append(p_inf[0])
    p_infy.append(p_inf[1])



##########################################################################
## 使用 matlibplot 作图
from matplotlib import pyplot as plt
## 首先是 px-py 2D 图
fig, ax = plt.subplots(figsize=(6.4,5))
h = ax.hist2d(p_infx, p_infy, bins=[150,
150],range=[[-1.5,1.5],[-1.5,1.5]], cmap='jet')
plt.colorbar(h[3], ax=ax)
plt.xlabel(r'$p_x$')
plt.ylabel(r'$p_y$')
plt.savefig('Q2_%s_n%d_pxpy2d.pdf'%(pol, N))
plt.show()
## px 直方图
plt.hist(p_infx, bins=150, range=[-1.5,1.5], histtype='step',
label=r'$p_x$ (%s)'%pol)
plt.legend()
plt.xlabel(r'$p_x$')
```

```
    plt.ylabel('Entries')
    plt.savefig('Q2_%s_n%d_px.pdf'%(pol, N))
    plt.show()
    ## py 直方图
    plt.hist(p_infy, bins=150, range=[-1.5,1.5], histtype='step',
    label=r'$p_y$ (%s)'%pol)
    plt.legend()
    plt.xlabel(r'$p_y$')
    plt.ylabel('Entries')
    plt.savefig('Q2_%s_n%d_py.pdf'%(pol, N))
    plt.show()
```

多线程处理代码（实际所用大样品情形）

多线程处理代码请参见：`02/Q2_multicore.py`. 由于调用函数相同，这里只展示主函数.

```
if __name__ == '__main__':
    A0 = 1.325
    WEmax = 0.010623
    Ip = 0.5

    nprocess, N = 4, 250 # 设置4线程，各250组样品
    ome = 0.057
    T = 2*pi/ome
    p_infx, p_infy = [], []
    Nloop = 0



    ##########################################################################
    ## 开始多线程处理
    import multiprocessing, pickle, time
    def worker(num): # 单个线程工作内容
        p_infx, p_infy = [], []
        for i in range(N):
            t, par = gen_sample() # 先生成样品
            rk4_solver(t, par, dt=0.5) # R-K4演化到激光结束，步长取为0.5
```

```
            p_inf = coulomb_evol(t, par) # 计算无穷远动量
            if p_inf == None: # 能量为负?(没有返回值),舍去
                continue
            p_infx.append(p_inf[0])
            p_infy.append(p_inf[1])
        with open('write%d.pkl'%num,'wb') as f: # 单进程生成完先存成pkl文件
            pickle.dump([p_infx, p_infy], f)
    jobs = []
    for i in range(nprocess): # 产生nprocess个工作
        job = multiprocessing.Process(target=worker, args=(i,))
        jobs.append(job)
        job.start()
        print('正在分配线程 %d'%i)
    print('任务已分配完毕,共将生成 %d 线程 × %d 事例共 %d
    组样本'%(nprocess, N, nprocess*N))

    ##############################################################################
    ## 等待全部处理完成
    while True: # 检查是否完成工作
        for job in jobs:
            if job.is_alive() == True:
                break
        else: ## 已全部处理完毕
            break
        time.sleep(3)
    p_infx, p_infy = [], [] # 读取所有pkl文件
    for num in range(nprocess):
        try:
            with open('write%d.pkl'%num,'rb') as f:
                arr = pickle.load(f)
                p_infx += arr[0]
                p_infy += arr[1]
        except Exception:
            print('多线程运行失败...可能是系统 IDE 不支持')
```

```
    ##############################################################################
    ## 使用 matlibplot 作图
================ 以下内容与上述相同，略去 ================
```

3 FPU 模型

题目简述：本题讨论 *FPU* 模型，即一维弹簧振子链首末固定，哈密顿量加入三次/四次非谐项可构成 α-*FPU* 与 β-*FPU* 模型. 哈密顿量分别为

$$H_\alpha = \frac{1}{2}\sum_{j=1}^{n} p_j^2 + \frac{1}{2}\sum_{j=0}^{n}(q_j - q_{j+1})^2 + \frac{\alpha}{3}\sum_{j=0}^{n}(q_j - q_{j+1})^3$$
$$H_\beta = \frac{1}{2}\sum_{j=1}^{n} p_j^2 + \frac{1}{2}\sum_{j=0}^{n}(q_j - q_{j+1})^2 + \frac{\beta}{4}\sum_{j=0}^{n}(q_j - q_{j+1})^4$$

. 回答以下问题：

(i) 对没有高次项情形解析/数值证明系统有守恒量

$$E_k = \frac{1}{2}\dot{Q}_k^2 + \frac{1}{2}\omega_k^2 Q_k^2, \quad \text{其中 } Q_k = \sqrt{\frac{2}{n}}\sum_j \sin\frac{\pi k j}{n+1} q_j, \quad \omega = 2\sin\frac{\pi k}{2(n+1)}$$

(ii) 选 α-*FPU* 模型，取一定的参数且初始能量都在 $k = 1$ 模式上 $(Q_1(0) = 4, \dot{Q}_1(0) = 0)$，研究系统演化阔成中能量再次几乎都回到 $k = 1$ 模式上的时刻和能量比例.

(iii) 给出可能解释.

(iv) 在更大的时间尺度寻找更大的回归周期.

(v) 取 $Q_1(0) = 20$，再解系统演化. 研究混沌特性、能均分定理.

(vi) 对 β-*FPU* 模型的讨论.

(vii) 对 β-*FPU* 模型奇偶模式的讨论.

(viii) β-*FPU* 模型研究 *KdV* 方程孤子解.

注：本题的所有图和程序均已在第一稿给出，第二稿仅更细致地完善了说明文档.

3.1 解题思路

本题唯一需要计算物理知识实现的就是常微分方程初值问题求解. 我们使用四阶隆格库塔算法（R-K4）实现. 不难看出，各个小问求解的差别仅在参数和方程的微小变动上，故只需一套程序框架即可完成本题. 本题更重要的在于物理图像的讨论.

3.2 算法提纲

本例使用 R-K4 算法求解常微分方程初值问题. 首先，由哈密顿量正则方程可知，α-FPU 系统的运动方程为

$$\begin{cases} \dot{q}_j = \dfrac{\partial H_\alpha}{\partial p_j} = p_j \\ \dot{p}_j = -\dfrac{\partial H_\alpha}{\partial q_j} = q_{j-1} + q_{j+1} - 2q_j + \alpha(q_{j-1} - q_j)^2 - \alpha(q_j - q_{j+1})^2 \end{cases}$$

为增加计算速度可将第 2 个等式化简为

$$\dot{p}_j = (q_{j-1} + q_{j+1} - 2q_j)[1 + \alpha(q_{j-1} - q_{j+1})];$$

β-FPU 系统的运动方程为

$$\begin{cases} \dot{q}_j = \dfrac{\partial H_\alpha}{\partial p_j} = p_j \\ \dot{p}_j = -\dfrac{\partial H_\alpha}{\partial q_j} = q_{j-1} + q_{j+1} - 2q_j + \beta(q_{j-1} - q_j)^3 - \beta(q_j - q_{j+1})^3 \end{cases}$$

为实现 R-K4 算法求解，定义 $22n$ 维数组 $\boldsymbol{y} = (q_1, \cdots, q_n, p_1, \cdots, p_n)$，可由运动方程得到微分方程的形式 $\boldsymbol{y}' = \boldsymbol{f}(t, \boldsymbol{y})$. 可以看出，本例的 $\boldsymbol{f}(t, \boldsymbol{y})$ 与 t 也无关. R-K4 算法的伪代码与上一题的相同，如下所示. 具体程序中将该算法封装在 `rk4_solver` 函数中.

while True

$\boldsymbol{k}_1 = \boldsymbol{f}(t_{n-1}, \boldsymbol{y}_{n-1})$
$\boldsymbol{k}_2 = \boldsymbol{f}(t_{n-1} + \frac{\Delta x_n}{2}, \boldsymbol{y}_{n-1} + \frac{\Delta t_n}{2}\boldsymbol{k}_1)$
$\boldsymbol{k}_3 = \boldsymbol{f}(t_{n-1} + \frac{\Delta x_n}{2}, \boldsymbol{y}_{n-1} + \frac{\Delta t_n}{2}\boldsymbol{k}_2)$
$\boldsymbol{k}_4 = \boldsymbol{f}(t_{n-1}, \boldsymbol{y}_{n-1} + \Delta t_n \boldsymbol{k}_3)$
$\boldsymbol{y}_n = \boldsymbol{y}_{n-1} + \frac{\Delta t_n}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4)$
$t_n = t_{n-1} + \Delta t_n$
if t_n 达到目标点: **break**
\# 开始下一步的计算

end while

还需要注意，由于前 7 个小问都是给定的 $Q(0)$ 初始值，故进行 R-K4 求解前需使用下面 (1) 式的逆变换

$$\boldsymbol{q}(0) = \boldsymbol{A}^{-1}\boldsymbol{Q}(0) = \boldsymbol{A}\boldsymbol{Q}(0), \quad \boldsymbol{p}(0) = \dot{\boldsymbol{q}}(0) = \boldsymbol{A}\dot{\boldsymbol{Q}}(0) = \boldsymbol{0}$$

转化为 $\boldsymbol{q}, \boldsymbol{p}$ 的初始值，从而得到数组 $\boldsymbol{y}$ 的初始值再进行演化；而第 (8) 问则直接代入已知的 $\boldsymbol{q}(0), \boldsymbol{p}(0)$ 孤子解的式子即可.

3.3 结果与讨论

第 (1) 问

首先进行理论求解. 题目已对如何进行变换做出提示，令

$$\boldsymbol{Q} = \boldsymbol{A}\boldsymbol{q}, \quad A_{ij} = \sqrt{\frac{2}{n}} \sum_{j=1}^{n} \sin \frac{\pi i j}{n+1} \tag{1}$$

我们下面证明原不含高次项的哈密顿量 H_0 可以写作

$$H_0 = \frac{n}{n+1}\left(\frac{1}{2}\dot{\boldsymbol{Q}}^{\mathrm{T}}\dot{\boldsymbol{Q}} + \frac{1}{2}\boldsymbol{Q}\boldsymbol{D}\boldsymbol{Q}\right) = \frac{n}{n+1}\left(\frac{1}{2}\dot{q}^{\mathrm{T}}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})\,\boldsymbol{q} + \frac{1}{2}\boldsymbol{q}\,(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{A})\,\boldsymbol{q}\right) \tag{2}$$

其中 $D_{ij} = \delta_{ij}\omega_i^2$.

首先计算 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$. 根据 $\boldsymbol{A}$ 对称性

$$\begin{aligned}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})_{ik} &= \sum_{j=1}^{n} A_{ij}A_{jk} = \frac{2}{n}\sum_{j=1}^{n} \sin\frac{\pi i j}{n+1}\sin\frac{\pi j k}{n+1} \\ &= \frac{1}{n}\sum_{j=1}^{n}\left(\cos\frac{\pi j(k-i)}{n+1} - \cos\frac{\pi j(k+i)}{n+1}\right)\end{aligned}$$

对上式的 cos 求和我们巧妙地用复数方法

$$\sum_{j=1}^{n}\cos\frac{\pi j(k-i)}{n+1} = \mathrm{Re}\sum_{j=1}^{n}\mathrm{e}^{\mathrm{i}\pi\frac{j(k-i)}{n+1}} = \begin{cases}\mathrm{e}^{\mathrm{i}\pi\frac{k-i}{n+1}}\frac{1-\mathrm{e}^{\mathrm{i}\pi\frac{nj(k-i)}{n+1}}}{1-\mathrm{e}^{\mathrm{i}\pi\frac{j(k-i)}{n+1}}} = -1, & (k-i \neq 2d(n+1),\ d \text{ 为整数}) \\ n, & (k-i = 2d(n+1))\end{cases}$$

因为 i, k 只能取从 1 到 n，不存在 $k+i = 2(n+1)$，则原式最终化简为

$$(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})_{ik} = \frac{n+1}{n}\,\delta_{ik}.$$

再求 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{A}$:

$$\begin{aligned}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{A})_{ik} &= \sum_{j=1}^{n}\sum_{l=1}^{n} A_{ij}D_{jl}A_{lk} = \sum_{j=1}^{n} A_{ij}A_{jk}\omega_j^2 \\ &= \tfrac{2}{n}\sum_{j=1}^{n}\sin\tfrac{\pi i j}{n+1}\sin\tfrac{\pi j k}{n+1}\left(4\sin^2\tfrac{\pi j}{2(n+1)}\right) \\ &= \tfrac{2}{n}\sum_{j=1}^{n}\left(\cos\tfrac{\pi j(k-i)}{n+1} - \cos\tfrac{\pi j(k+i)}{n+1}\right)\left(1-\cos\tfrac{\pi j}{n+1}\right) \\ &= \tfrac{2(n+1)}{n}\delta_{ik} - \tfrac{2}{n}\sum_{j=1}^{n}\left(\cos\tfrac{\pi j(k-i-1)}{n+1} + \cos\tfrac{\pi j(k-i+1)}{n+1} - \cos\tfrac{\pi j(k+i-1)}{n+1} - \cos\tfrac{\pi j(k+i+1)}{n+1}\right) \\ &= \tfrac{n+1}{n}\left(2\delta_{ik} - \delta_{i,\,k+1} - \delta_{i,\,k-1}\right)\end{aligned}$$

上式的矩阵表示为

$$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{A} = \frac{n+1}{n}\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

显然对应题中给出的哈密顿量 $\boldsymbol{q}\,(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{A})\,\boldsymbol{q} = \sum_{j=0}^{n}(q_j - q_{j+1})^2$，其中规定了 $q_0 = q_{n+1} = 0$. 因此 H_0 即为

$$H_0 = \frac{1}{2}\sum_{j=1}^{n} p_j^2 + \frac{1}{2}\sum_{j=0}^{n}(q_j - q_{j+1})^2.$$

上式根据哈密顿正则方程 $\dot{q}_j = \partial H_0/\partial p_j = p_j$，替换上面 p_j 即可. 于是 (2) 式证毕. 对于题目所给的力学量 $H_k = \frac{1}{2}\dot{Q}_k^2 + \frac{1}{2}\omega_k^2 Q_k^2 = \frac{1}{2}P_k^2 + \frac{1}{2}\omega_k^2 Q_k^2$，其随时间的变化可在正则变量 $\boldsymbol{Q}, \boldsymbol{P}$ 下由泊松括号给出：

$$\{H_k,\, H_0\}_{\mathrm{PB}} = \sum_{j=1}^{n}\frac{\partial H_k}{\partial Q_j}\frac{\partial H_0}{\partial P_j} - \frac{\partial H_k}{\partial P_j}\frac{\partial H_0}{\partial Q_j} = \omega Q_j P_j - P_j \omega Q_j = 0,$$

于是可知 H_k 为系统的守恒量.

第 (2) 问

按题中要求选定参量，运行程序 `03/Q3_2_alpha160.py`，可以得到 $k = 1, 2, 3, 4$ 的能量模式随时间演化的曲线，如图 10 所示. 可以看出，虽然为非线性系统，但各模式的能量演化存在很强的规律. 在 $t \approx 153\,\frac{2\pi}{\omega_1}$ 时能量几乎又都回到了 $k = 1$ 模式上，能量回归的比例高达 98.1%.

第 (3) 问

这一回归周期的产生或许可以从相空间入手解释. 对于经典力学我们有庞加莱重现原理，即对于当前的非线性系统，给定演化初始时相空间上的点 P 和任意小量 $\epsilon > 0$，经历足够长的时间后一定能回到相空间上距离初始状态小于 ϵ 的点上，所以给定任意高的回归比率，都存在某个时刻使得系统能量以高于该比例值回归到 $k = 1$ 模式上.

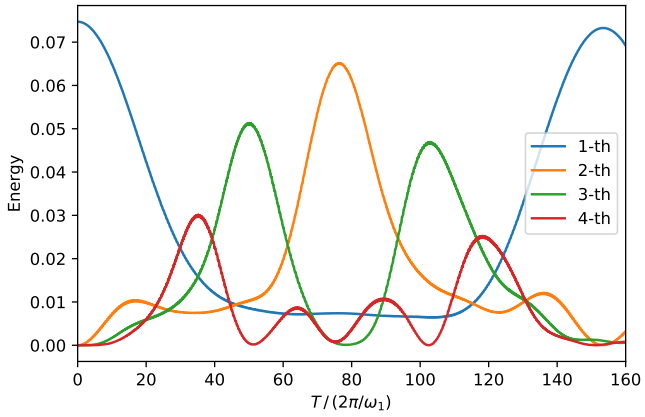

图 10: $k = 1, 2, 3, 4$ 的能量模式随时间演化的曲线.

第 (4) 问

将系统演化时间增长至 $T = 4000\,\frac{2\pi}{\omega_1}$，运行本问程序 `Q3_4_alpha4000.py`，得到的结果如图 11所示. 令人惊异的发现 $k = 1$ 模式能量的演化具有更高的回归周期，在这一尺度上看曲线像是两个有规律的波形的叠加. 模拟计算给出在 $t = 3520\,\frac{2\pi}{\omega_1}$ 附近又出现了能量的最大值，并且能量回归比率高达 99.5%.

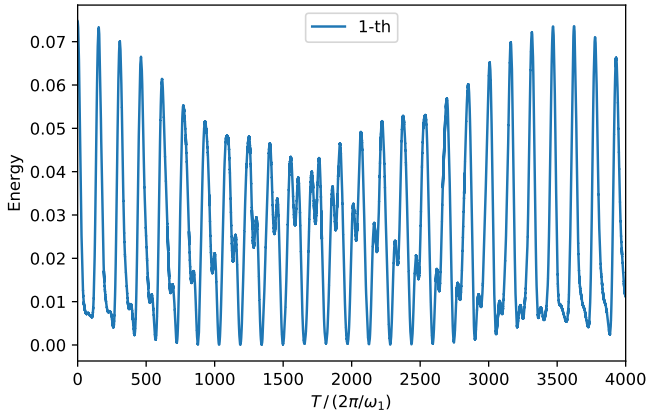

图 11: $k = 1$ 模式在 $0 < t < 4000\,\frac{2\pi}{\omega}$ 范围内的演化曲线.

第 (5) 问

本小问改变参数为 $Q = 20$，设置时间长度为 $T = 1000\,\frac{2\pi}{\omega}$. 运行本问程序 `Q3_5_alphaQ20.py` 可分别得到图 12 所示的 $k = 1$ 模式演化图和 $k = 1, \cdots, 8$ 模式

的能量平均值演化图. 可见，$k=1$ 模式的能量起初呈现出较有规律的振荡特性，但在 $t \gtrsim 200\,\frac{2\pi}{\omega_1}$ 以后就开始呈现出不规律的振荡，此时认为出现混沌现象. 若我们对 8 个模式在任意时刻下计算 $[0,t)$ 期间内能量的平均，可以发现在 $t \gtrsim 200\,\frac{2\pi}{\omega_1}$ 其平均能量近似稳定，这正好在混沌现象出现的附近，表明混沌系统的无规律性在长时间求平均下会产生抵消效应，呈现较稳定的平均值，也即达到了 Fermi 希望看到的 “热平衡” 状态. 但可以发现 8 个能量平均值并不相等，是逐渐递减的. 具体的模拟计算有

$$\langle E_1\rangle = 0.22750, \quad \langle E_2\rangle = 0.20818, \quad \langle E_3\rangle = 0.19515, \quad \langle E_4\rangle = 0.17953,$$
$$\langle E_5\rangle = 0.17040, \quad \langle E_6\rangle = 0.15779, \quad \langle E_7\rangle = 0.12722, \quad \langle E_8\rangle = 0.10189.$$

并不能反映完美的能均分定理.

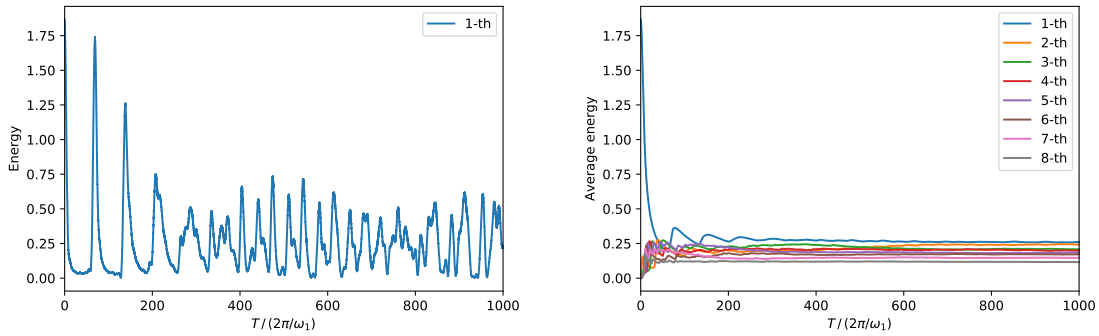

图 12: $k=1$ 模式在 $0<t<1000\,\frac{2\pi}{\omega}$ 范围内的演化，可观察到混沌现象（左图）；$k=1,\cdots,8$ 模式在 $0<t<1000\,\frac{2\pi}{\omega}$ 范围内对之前的时间所作的能量平均（右图）.

第 (6) 问

对于 β-FPU 模型，我们可以观测到相同的规律振荡与混沌现象. 经多次测试我们选择参数 $\beta=1$，$Q_1(0)=15$, $n=32$. 运行本问程序 `03/Q3_6_beta_repeat.py` 可先后获得 $T=160\,\frac{2\pi}{\omega_1}$ 尺度下前 4 个模式的能量变化以及 $T=1000\,\frac{2\pi}{\omega_1}$ 更长尺度下可以 $k=1$ 模式的能量变化，结果见图 13. 可以发现左图表明该尺度范围内系统呈现出如第 (2)、(4) 小问的规律振荡的特性（此外还会发现能量只在奇数模式间传递，与第 (7) 问讨论的现象同），而右图表明在 $t \gtrsim 200\,\frac{2\pi}{\omega_1}$ 以后演化开始混沌，呈现第 (5) 小问所述的振荡特性. 于是我们模拟证明了 β-FPU 也有类似的规律振荡与混沌现象.

特别值得说明的是，α/β-FPU 系统参数的选取会显著影响演化过程和现象. 以本问举例，若设置 β 参数过小，或 $Q_1(0)$ 过小，则在很长时间能量都保持在 $k=1$ 模式上，此时非线性项过小以至于对系统影响可以忽略；而如果设置 β 或 $Q_1(0)$ 过大，

则系统一开始就呈现混沌，毫无规律可言，说明非线性项很大，对系统影响显著. 只有在比较适中的参数选取下才能观察到先规律后混沌的特性.

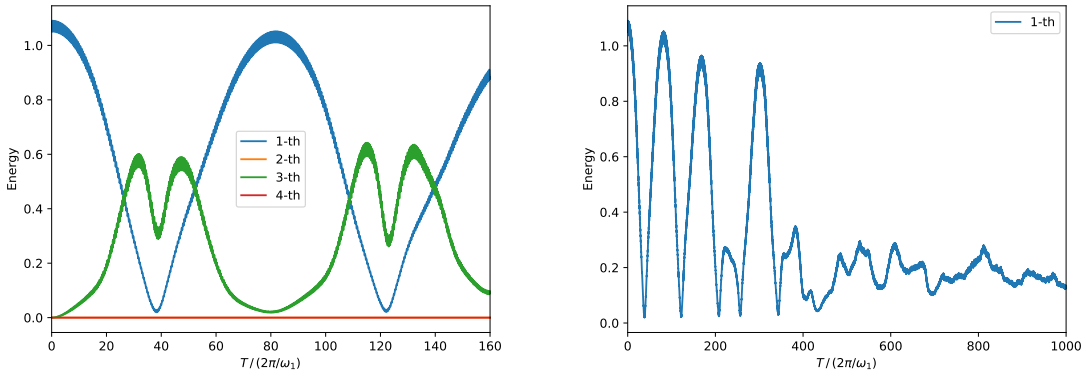

图 13: $T = 160\,\frac{2\pi}{\omega_1}$ 尺度下前 4 个模式的能量变化（左图）与 $T = 1000\,\frac{2\pi}{\omega_1}$ 尺度下 $k = 1$ 模式的能量变化（右图）.

第 (7) 问

按要求设定参数. 运行本问程序 `03/Q3_7_beta_oddeven.py` 可得到对数轴上 $k = 9, \cdots, 13$ 模式的能量演化，如图 14 所示. 可以发现只有奇数模式有振荡，偶数模式非常接近于 0（大概在 10^{-31} 量级活动）. 也即，如果初始时能量都分布在奇数模式上，则之后能量只会在奇数模式间进行分配.

这一现象可直接用理论解释. 根据 (1) 式可看出正则变量 $\boldsymbol{Q}$ 的奇数模式之于各个 q_j 的系数是关于 $i = \frac{n+1}{2}$ 轴线反对称（或奇宇称）的；$\boldsymbol{Q}$ 的偶数模式则是关于 $i = \frac{n+1}{2}$ 轴线对称（或偶宇称）的. 并且，根据哈密顿正则方程可知

$$\ddot{q}_j = -\frac{\partial H_\alpha}{\partial q_j} = q_{j-1} + q_{j+1} - 2q_j + \beta(q_{j-1} - q_j)^3 - \beta(q_j - q_{j+1})^3.$$

若将上式角标 j 替换为 $j' = n + 1 - j$，会发现上式恰好变成格点 q_{n+1-j} 的运动方程，这说明运动方程也有关于 $i = \frac{n+1}{2}$ 的对称性. 而对于 α-FPU 模型则不具有这种对称性. 因此，若初始时刻关于 α 呈现反对称，则后续演化过程中一定保持这种状态，因此系统的奇数模式与偶数模式只能在各自之间交换能量，互不相通.

第 (8) 问

此问我们直接对 $\boldsymbol{q}, \boldsymbol{p}$ 设置孤子解的初始参数. 运行 `03/Q3_8_beta_soliton.py`，可得到 $q_i, (i = 1, \cdots, 128)$ 随时间演化的图像，如图 15 所示. 由此可见孤子解的转播

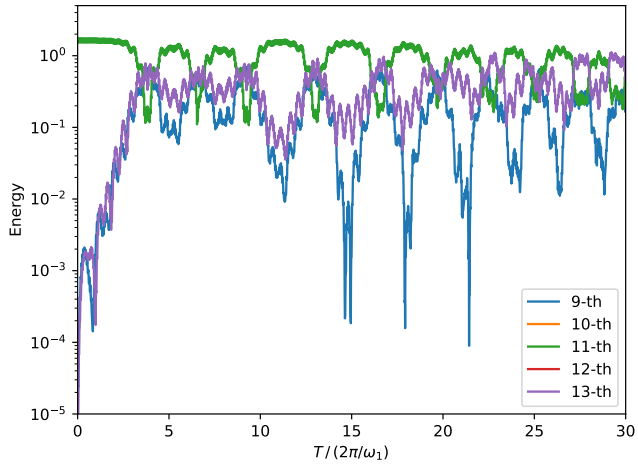

图 14: 初始条件 $Q_{11}(0) = 1$ 下的 $k = 9, 10, 11, 12, 13$ 能量模式的演化.

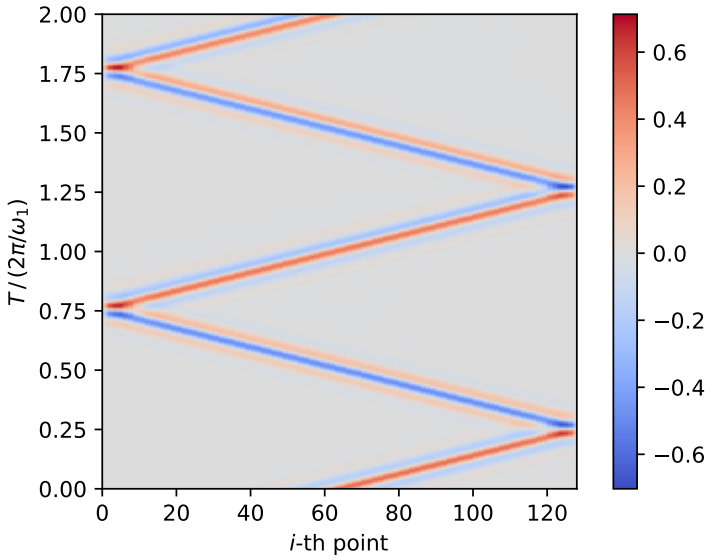

图 15: 孤子解模式下 q_i, $(i = 1, \cdots, 128)$ 随时间演化的图像. 其中横坐标为角标 i，纵坐标为时间轴. 红色表示正的振幅；蓝色表示负的振幅.

模式，即一个波包正在不断进行往返运动，周期恰好为 $T = \frac{2\pi}{\omega_1}$. 一个有趣的现象是波包向左、右运动时其振幅方向是反的. 这是因为 $i = 0$ 与 $n - 1$ 处也即系统两侧可作为墙壁，要求有 $q_0 = q_{n+1} = 0$，故波发生反射时会出现半波损失，等价于振幅反号.

3.4 源代码

第 (2) 问代码

第 (2) 问代码: `03/Q3_2_alpha160.py`.

```
# -*- coding: utf-8 -*-
from numpy import sqrt, pi, cos, sin, exp, cosh, tanh  # 只导入常用函数啦

n, Tmax, alpha, beta = 0, 0, 0, 0
def rk4_solver(t, y, dt, fpu='0', saveEt=4, saveq=0): ## 给定微分方程的
    R-K4 算法求解器

    ## 参数解释: 基本参数: t, y(t), dt; fpu='0', 'alpha',
    'beta'选择特定求解工具, saveEt: 过程中存入(前多少个)Et, saveq:
    过程中存入(前多少个)q

    def f_0(y): # y = [q1, q2, ..., qn, p1, p2, ..., pn]
    这是无高次项的微分方程
        dp = [0 for i in range(n)]
        for i in range(1, n-1):
            dp[i] = y[i-1] + y[i+1] - 2*y[i]
        dp[0] = y[1] - 2*y[0]
        dp[n-1] = y[n-2] - 2*y[n-1]
        return [y[i+n] for i in range(n)] + dp

    def f_alpha(y): # y = [q1, q2, ..., qn, p1, p2, ..., pn] 这是
    alpha-FPU 的微分方程
        dp = [0 for i in range(n)]
        for i in range(1, n-1):
            dp[i] = (y[i-1] + y[i+1] - 2*y[i]) * (1 + alpha*(y[i-1] -
    y[i+1]))
        dp[0] = (y[1] - 2*y[0]) * (1 - alpha*y[1])
        dp[n-1] = (y[n-2] - 2*y[n-1]) * (1 + alpha*y[n-2])
        return [y[i+n] for i in range(n)] + dp

    def f_beta(y): # y = [q1, q2, ..., qn, p1, p2, ..., pn] 这是 beta-FPU
    的微分方程
```

```
        dp = [0 for i in range(n)]
        for i in range(1, n-1):
            dp[i] = y[i-1] + y[i+1] - 2*y[i] + beta*(y[i-1]-y[i])**3 -
    beta*(y[i]-y[i+1])**3
        dp[0] = y[1] - 2*y[0] - beta*(y[0]-y[1])**3
        dp[n-1] = y[n-2] - 2*y[n-1] + beta*(y[n-2]-y[n-1])**3
        return [y[i+n] for i in range(n)] + dp

    if fpu == 'alpha': # 选择用哪个方程
        f = f_alpha
    elif fpu == 'beta':
        f = f_beta
    else:
        f = f_0
    stop = False
    while True: # R-K4 的每次迭代
        if Tmax-t < dt: # 判停条件
            dt = Tmax-t
            stop = True
        k1 = f(y)
        k2 = f([y[i] + dt/2*k1[i] for i in range(2*n)])
        k3 = f([y[i] + dt/2*k2[i] for i in range(2*n)])
        k4 = f([y[i] + dt*k3[i] for i in range(2*n)])
        for i in range(2*n):
            y[i] += dt/6 * (k1[i] + 2*k2[i] + 2*k3[i] + k4[i])
        t += dt
        t_list.append(t)
        if saveEt > 0: # 若要过程中储存 Et
            Et.append(calc_energy(y, lead=saveEt))
        if saveq > 0:  # 若要过程中储存 q
            q_list.append(y[:saveq])
        if stop == True:
            break
    return t, y

def calc_energy(y, lead): # 通过当前 y=(q, p) 计算能量 (要先变换到 Q, P)
```

```
    Q = [sum([Aij[(i+1)*(j+1)]*y[j] for j in range(n)]) for i in
    range(lead)]
    P = [sum([Aij[(i+1)*(j+1)]*y[j+n] for j in range(n)]) for i in
    range(lead)]
    return [n/(n+1) * (0.5*P[i]**2 + 0.5*(ome[i]*Q[i])**2) for i in
    range(lead)]

##############################################################################
## 开始第(2)小问
if __name__ == '__main__':
    alpha = 0.25
    n = 32
    Q1_0 = 4
    TN = 160
    ome = [2*sin(k*pi/(2*n+2)) for k in range(1, n+1)] # 简振模
    Aij = [sqrt(2/n)*sin(k*pi/(n+1)) for k in range(n**2+2)] #
    变换矩阵系数提前算出来
    Tmax = TN * 2*pi / ome[0]
    Q_0 = [Q1_0] + [0 for i in range(n-1)]
    t, y = 0, [0 for i in range(2*n)]
    for i in range(n):
        y[i] = sum([Aij[(i+1)*(j+1)]*Q_0[j] for j in range(n)])

    ## 初始化时间与能量 Et 列表 (存入前 4 个能量)
    t_list, Et = [], []
    t_list.append(t)
    Et.append(calc_energy(y, lead=4))

    ## 计算! (选步长 dt=0.1)
    print('开始使用 R-K4 计算, 预计耗时 <1 分钟')
    t, y = rk4_solver(t, y, dt=0.1, fpu='alpha', saveEt=4)



    ##############################################################################
    ## 使用 matlibplot 作图
    from matplotlib import pyplot as plt
```

```
    for i in range(4):
        plt.plot([t_list[i] /(2*pi/ome[0]) for i in range(len(t_list))],
    [Et[m][i] for m in range(len(Et))], label='%d-th'%(i+1))
    plt.xlim([0, TN])
    plt.legend()
    plt.xlabel(r'$T\,/\,(2\pi/\omega_1)$')
    plt.ylabel(r'Energy')
    plt.savefig('Q3_2_alpha160.pdf')
    plt.show()

    ## 计算一些回归的时间
    Et0 = [Et[m][0] for m in range(len(Et))]
    Etidx = Et0.index(max(Et0[4000:]))
    print('能量几乎回归到k=1的时间是 T/(2 / _1) = %.2f
    '%(t_list[101303]/(2*pi/ome[0])))
    print('能量比例约为   = %.4f '%(Et0[101303]/Et0[0]))
```

第 (4) 问代码

第 (4) 问代码: `03/Q3_4_alpha4000.py`, 这里仅展示除了与第 (2) 问相同的 `rk4_solver` 函数的部分.

```
## 开始第(4)小问 (运行要花好长时间)
if __name__ == '__main__':
    alpha = 0.25
    n = 32
    Q1_0 = 4
    TN = 4000
    ome = [2*sin(k*pi/(2*n+2)) for k in range(1, n+1)] # 简振模
    Aij = [sqrt(2/n)*sin(k*pi/(n+1)) for k in range(n**2+2)] #
    变换矩阵系数提前算出来
    Tmax = TN * 2*pi / ome[0]
    Q_0 = [Q1_0] + [0 for i in range(n-1)]
    t, y = 0, [0 for i in range(2*n)]
    for i in range(n):
        y[i] = sum([Aij[(i+1)*(j+1)]*Q_0[j] for j in range(n)])
```

```
    ## 初始化时间与能量 Et 列表 (存入第 1 个能量)
    t_list, Et = [], []
    t_list.append(t)
    Et.append(calc_energy(y, lead=1))

    ## 计算! (选步长 dt=0.1)
    print('开始使用 R-K4 计算, 预计耗时较长')
    t, y = rk4_solver(t, y, dt=0.1, fpu='alpha', saveEt=1)



    ##########################################################################
    ## 使用 matlibplot 作图
    from matplotlib import pyplot as plt
    for i in range(1):
        plt.plot([t_list[i] /(2*pi/ome[0]) for i in range(len(t_list))],
    [Et[m][i] for m in range(len(Et))], label='%d-th'%(i+1))
    plt.xlim([0, TN])
    plt.legend()
    plt.xlabel(r'$T\,/\,(2\pi/\omega_1)$')
    plt.ylabel(r'Energy')
    plt.savefig('Q3_4_alpha4000.pdf')
    plt.show()
```

第 (5) 问代码

第 (5) 问代码: `03/Q3_5_alphaQ20.py` ,这里仅展示除了与第 (2) 问相同的 `rk4_solver` 函数的部分.

```
## 开始第(5)小问 (运行要花好长时间)
if __name__ == '__main__':
    alpha = 0.25
    n = 32
    Q1_0 = 20
    TN = 1000
    ome = [2*sin(k*pi/(2*n+2)) for k in range(1, n+1)] # 简振模
    Aij = [sqrt(2/n)*sin(k*pi/(n+1)) for k in range(n**2+2)] #
    变换矩阵系数提前算出来
```

```
Tmax = TN * 2*pi / ome[0]
Q_0 = [Q1_0] + [0 for i in range(n-1)]
t, y = 0, [0 for i in range(2*n)]
for i in range(n):
    y[i] = sum([Aij[(i+1)*(j+1)]*Q_0[j] for j in range(n)])

## 初始化时间与能量 Et 列表 (存入前 8 个能量)
t_list, Et = [], []
t_list.append(t)
Et.append(calc_energy(y, lead=8))

## 计算! (选步长 dt=0.1)
print('开始使用 R-K4 计算, 预计耗时较长')
t, y = rk4_solver(t, y, dt=0.1, fpu='alpha', saveEt=8)



##############################################################################
## 使用 matlibplot 作图
from matplotlib import pyplot as plt
for i in range(1): ## 只做第一个能量的图, 看混沌
    plt.plot([t_list[i] /(2*pi/ome[0]) for i in range(len(t_list))],
[Et[m][i] for m in range(len(Et))], label='%d-th'%(i+1))
plt.xlim([0, TN])
plt.legend()
plt.xlabel(r'$T\,/\,(2\pi/\omega_1)$')
plt.ylabel(r'Energy')
plt.savefig('Q3_5_alphaQ20_chaos.pdf')
plt.show()

## 下面将能量从最开始取平均, 获得 Et_acc
Et_acc = [[Et[i][j] for j in range(len(Et[0]))] for i in
range(len(Et))]
for j in range(8):
    for i in range(1, len(Et_acc)):
        Et_acc[i][j] += Et_acc[i-1][j]
    for i in range(1, len(Et_acc)):
```

```
            Et_acc[i][j] /= i+1
    for i in range(8): ## 对k=1...8 做平均能量图
        plt.plot([t_list[i] /(2*pi/ome[0]) for i in range(len(t_list))],
    [Et_acc[m][i] for m in range(len(Et))], label='%d-th'%(i+1))
    plt.xlim([0, TN])
    plt.legend()
    plt.xlabel(r'$T\,/\,(2\pi/\omega_1)$')
    plt.ylabel(r'Average energy')
    plt.savefig('Q3_5_alphaQ20_aver_energy.pdf')
    plt.show()
```

第 (6) 问代码

第 (6) 问代码: `03/Q3_6_beta_repeat.py`, 这里仅展示除了与第 (2) 问相同的 `rk4_solver` 函数的部分.

```
## 开始第(6)小问 (运行要花好长时间)
if __name__ == '__main__':
    beta = 1
    n = 32
    Q1_0 = 15
    TN = 160
    ome = [2*sin(k*pi/(2*n+2)) for k in range(1, n+1)] # 简振模
    Aij = [sqrt(2/n)*sin(k*pi/(n+1)) for k in range(n**2+2)] #
    变换矩阵系数提前算出来
    Tmax = TN * 2*pi / ome[0]
    Q_0 = [Q1_0] + [0 for i in range(n-1)]
    t, y = 0, [0 for i in range(2*n)]
    for i in range(n):
        y[i] = sum([Aij[(i+1)*(j+1)]*Q_0[j] for j in range(n)])

    ## 初始化时间与能量 Et 列表 (存入前 4个能量)
    t_list, Et = [], []
    t_list.append(t)
    Et.append(calc_energy(y, lead=4))

    ## 计算! (选步长 dt=0.1)
```

```
print('开始使用 R-K4 计算, 先只算到 Tn=160, 为了看前 4
个模式的规律振荡, 预计耗时 <1 分钟')
t, y = rk4_solver(t, y, dt=0.1, fpu='beta', saveEt=4)

########################################
## 使用 matlibplot 作图
from matplotlib import pyplot as plt
for i in range(4):
    plt.plot([t_list[i] /(2*pi/ome[0]) for i in range(len(t_list))],
[Et[m][i] for m in range(len(Et))], label='%d-th'%(i+1))
plt.xlim([0, TN])
plt.legend()
plt.xlabel(r'$T\,/\,(2\pi/\omega_1)$')
plt.ylabel(r'Energy')
plt.savefig('Q3_6_beta160.pdf')
plt.show()



#########################################################################
## 第二步, 再一直算到 Tn=1000, 主要看k=1模式变混沌的现象
TN2 = 1000
Tmax = TN2 * 2*pi / ome[0]

## 计算! (选步长 dt=0.1)
print('开始使用 R-K4 计算, 再一直算到 Tn=1000, 看第 4
个模式变混沌的过程, 预计耗时 3-5 分钟')
t, y = rk4_solver(t, y, dt=0.1, fpu='beta', saveEt=1) # (这次只存入第
1个能量就好啦)

########################################
## 使用 matlibplot 作图
for i in range(1):
    plt.plot([t_list[i] /(2*pi/ome[0]) for i in range(len(t_list))],
[Et[m][i] for m in range(len(Et))], label='%d-th'%(i+1))
plt.xlim([0, TN2])
plt.legend()
```

```
    plt.xlabel(r'$T\,/\,(2\pi/\omega_1)$')
    plt.ylabel(r'Energy')
    plt.savefig('Q3_6_beta1000.pdf')
    plt.show()
```

第 (7) 问代码

第 (7) 问代码: `03/Q3_7_beta_oddeven.py`, 这里仅展示除了与第 (2) 问相同的 `rk4_solver` 函数的部分.

```
## 开始第(7)小问 (比较快)
if __name__ == '__main__':
    beta = 1
    n = 16
    Q11_0 = 1
    TN = 30
    ome = [2*sin(k*pi/(2*n+2)) for k in range(1, n+1)] # 简振模
    Aij = [sqrt(2/n)*sin(k*pi/(n+1)) for k in range(n**2+2)] #
    变换矩阵系数提前算出来
    Tmax = TN * 2*pi / ome[0]
    Q_0 = [0 for i in range(n)]
    Q_0[10] = Q11_0
    t, y = 0, [0 for i in range(2*n)]
    for i in range(n):
        y[i] = sum([Aij[(i+1)*(j+1)]*Q_0[j] for j in range(n)])

    ## 初始化时间与能量 Et 列表 (存入前 13 个能量)
    t_list, Et = [], []
    t_list.append(t)
    Et.append(calc_energy(y, lead=13))

    ## 计算! (选步长 dt=0.05)
    print('开始使用 R-K4 计算, 预计挤需 10 秒')
    t, y = rk4_solver(t, y, dt=0.05, fpu='beta', saveEt=13)



    ##########################################################################
```

```
    ## 使用 matlibplot 作图
    from matplotlib import pyplot as plt
    for i in range(8, 13):
        plt.plot([t_list[i] /(2*pi/ome[0]) for i in range(len(t_list))],
    [Et[m][i] for m in range(len(Et))], label='%d-th'%(i+1))
    plt.yscale('log')
    plt.xlim([0, TN])
    plt.ylim([1e-5, 5])
    plt.legend()
    plt.xlabel(r'$T\,/\,(2\pi/\omega_1)$')
    plt.ylabel(r'Energy')
    plt.savefig('Q3_7_beta_oddeven.pdf')
    plt.show()
```

第 (8) 问代码

第 (2) 问代码: `03/Q3_8_beta_soliton.py`, 这里仅展示除了与第 (2) 问相同的 `rk4_solver` 函数的部分.

```
## 开始第(8)小问 (比较快)
if __name__ == '__main__':
    beta = 1
    n = 128
    Q11_0 = 1
    TN = 2
    ome = [2*sin(k*pi/(2*n+2)) for k in range(1, n+1)] # 简振模
    Aij = [sqrt(2/n)*sin(k*pi/(n+1)) for k in range(n**2+2)] #
    变换矩阵系数提前算出来
    Tmax = TN * 2*pi / ome[0]

    B, k0 = 0.5, 11 ## 孤子解的特别参数
    t, y = 0, [0 for i in range(2*n)]
    for i in range(1, n+1): ## 代入达成孤子解的初条件
        y[i-1] = B*cos(pi*k0*(i-n/2)/(n+1)) /
    cosh(sqrt(3/2)*B*ome[k0-1]*(i-n/2))
        y[i+n-1] = B/cosh(sqrt(3/2)*B*ome[k0-1]*(i-n/2)) \
```

```
        * (ome[k0-1]*(1+3/16*(ome[k0-1]*B)**2)*
    sin(pi*k0*(i-n/2)/(n+1)) \
        +
    sqrt(3/2)*B*cos(pi*k0*(i-n/2)/(n+1))*sin(pi*k0/(n+1))*tanh(sqrt(3/2) \
        *B*ome[k0-1]*(i-n/2)))

    ## 初始化时间与 q 的列表 (存入前 128 个, 也即全部的 q)
    t_list, q_list = [], []
    t_list.append(t)
    q_list.append(y[:128])

    ## 计算! (选步长 dt=0.05)
    t, y = rk4_solver(t, y, dt=0.05, fpu='beta', saveEt=0, saveq=128)



    ##############################################################################
    ## 使用 matlibplot 作图
    from matplotlib import pyplot as plt
    im = plt.imshow(q_list_, extent=(0, 128, 0, TN), aspect=128/TN,
    cmap='coolwarm')
    plt.colorbar(im)
    plt.xlabel(r'$i$-th point')
    plt.ylabel(r'$T\,/\,(2\pi/\omega_1)$')
    plt.savefig('Q3_8_beta_soliton.pdf')
    plt.show()
```