

计算物理第三次大作业

王台宜

学号：1500011419

2019年6月10日

目录

1 第一题	4
1.1 第一问	4
1.1.1 题目	4
1.1.2 思路	4
1.1.3 程序实现	5
1.1.4 结果展示	8
1.2 第二问	10
1.2.1 题目	10
1.2.2 思路	10
1.2.3 程序实现	10
1.2.4 结果展示	12
1.3 第三问	14
1.3.1 题目	14
1.3.2 思路	15
1.3.3 程序实现	15
1.3.4 结果展示	19
1.4 第四问	20
1.4.1 题目	20
1.4.2 思路	20
1.4.3 程序实现	21
1.4.4 结果展示	22
2 第二题	23
2.1 第一问	23
2.1.1 题目	23
2.1.2 思路	23
2.1.3 程序实现	24
2.1.4 结果展示	26
2.2 第二问	28

2.2.1 题目	28
2.2.2 思路	28
2.2.3 程序实现	28
2.2.4 结果展示	30
2.3 第三四问	32
2.3.1 题目	32
2.3.2 思路	32
2.3.3 程序实现	32
2.3.4 结果展示	37
3 第三题	40
3.1 第一问	40
3.1.1 题目	40
3.1.2 求解	40
3.2 第二问	41
3.2.1 题目	41
3.2.2 思路	41
3.2.3 程序实现	42
3.2.4 结果展示	44
3.3 第三问	44
3.3.1 题目	44
3.3.2 求解	45
3.4 第四问	45
3.4.1 题目	45
3.4.2 求解	45
3.5 第五问	46
3.5.1 题目	46
3.5.2 求解	46
3.6 第六问	49
3.6.1 题目	49
3.6.2 求解	49
3.7 第七问	50
3.7.1 题目	50
3.7.2 求解	50
3.8 第八问	52
3.8.1 题目	52
3.8.2 思路	52
3.8.3 程序实现	52

Chapter 1

第一题

1.1 第一问

1.1.1 题目

(1) 请数值求解一维氢原子的基态波函数的概率分布 $|\psi_0(x)|^2$ 和得到的基态能量 E_0 与 -0.48 (应在此值附近) 之间的相对误差 (基态的电离势 $I_p = -E_0$);

1.1.2 思路

求基态氢原子的波函数和基态能量，就是求解一个定态的薛定谔方程，势能函数为软化过的库仑势： $-\frac{1}{\sqrt{2+x^2}}$ ，薛定谔方程为：

$$[-\frac{1}{2} \frac{d^2}{dx^2} + V(x)]\psi(x) = E_n \psi(x)$$

解这个方程，需要把坐标离散化，这样波函数的求二阶导就会变成有限差分的形式：

$$-\frac{1}{2} \frac{d^2}{dx^2} \psi(x_i) = -0.5\psi(x_i + \Delta x) + \psi(x_i) - 0.5\psi(x_i - \Delta x)$$

于是整个薛定谔方程变成了一个三对角方程，解这个方程的特征值即可，即是各能态能量，特征向量就是各个态的波函数。

有限差分的步长为 h ，矩阵写出来是：

$$\begin{bmatrix} \frac{1}{h^2} + V(x_0) & \frac{-1}{2h^2} & 0 & (\text{此处忽略一项}) & 0 & \cdots & 0 & 0 \\ \frac{-1}{2h^2} & \frac{1}{h^2} + V(x_1) & \frac{-1}{2h^2} & 0 & 0 & \cdots & 0 & 0 \\ 0 & \frac{-1}{2h^2} & \frac{1}{h^2} + V(x_2) & \frac{-1}{2h^2} & 0 & \cdots & 0 & 0 \\ 0 & 0 & \frac{-1}{2h^2} & \frac{1}{h^2} + V(x_3) & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \frac{1}{h^2} + V(x_{n-2}) & \frac{-1}{2h^2} & \frac{-1}{h^2} + V(x_{n-1}) \\ 0 & 0 & 0 & 0 & \cdots & \frac{-1}{2h^2} & \frac{1}{h^2} + V(x_{n-1}) & \text{此处忽略一项} \end{bmatrix}$$

这是一个三对角系数矩阵形成的线性方程组，可以利用追赶法求解，要求基态能量，题目中已经说基态能量在 -0.48 左右，所以利用这个条件加上反幂法就可以求得基态能量和波函数。事实上，这里需要解释的是，由于第一项和最后一项（矩阵第一行和最后一行）求解时并不是严格三对角矩阵，但是做三对角矩阵近似并不影响结果，因此忽略了首尾两项的部分差分项。

1.1.3 程序实现

首先需要预定义一些解线性方程和线性代数常见的函数，例如矩阵、向量间的运算，以及追逐法求解线性方程的解法。这与此前所有大作业都是类似的，具体如下所示：

```
1 import random
2 import math
3 import matplotlib.pyplot as plt
4 import matplotlib as mpl
5 mpl.rcParams['font.sans-serif'] = ['KaiTi']
6 mpl.rcParams['font.serif'] = ['KaiTi']
7 mpl.rcParams['axes.unicode_minus'] = False # 解决保存图像是负号'-'显示
8     为方块的问题，或者转换负号为字符串
9
10
11 def listminusfunction(list1, list2): # 向量相减
12     list_return = [0] * len(list2)
13     if len(list1) != len(list2):
14         print("length does not match")
15         exit()
16     else:
17         for n in range(len(list1)):
18             list_return[n] = list1[n] - list2[n]
19     return list_return
20
21
22 def max_num(v): # 向量最大值
23     n = len(v)
24     res = 0
25     num = 0
26     for i in range(0, n):
27         if math.fabs(v[i]) > res:
28             res = math.fabs(v[i])
29             num = i
30     return v[num]
31
32 def V(x):
33     return -1/math.sqrt(2+x**2)
34
35
36
37 def listnum_multiply(n, list): # 向量数乘
38     vec = []
39     for i in range(0, len(list)):
40         vec.append(n * list[i])
```

```

    return vec

43

45 def chasesolvingfunction(n, a, b, c, f): # 定义追逐法解方程的函数
46     a1 = deepcopy(a)
47     b1 = deepcopy(b)
48     c1 = deepcopy(c)
49     f1 = deepcopy(f)
50     for i in range(1, n):
51         m = a1[i] / b1[i-1]
52         b1[i] = b1[i] - m * c1[i-1]
53         f1[i] = f1[i] - m * f1[i-1]
54     x = [0] * n
55     x[n-1] = f1[n-1]/b1[n-1]
56     for i in range(n-2, -1, -1):
57         x[i] = (f1[i] - c1[i] * x[i + 1])/b1[i]
58
59     return x

```

然后是主函数部分的处理，首先确定步长和 x_{max} ，然后就可以确定划分的区间数，根据这两个量列出三对角矩阵的三个对角向量，分别是 a,b,c，注意因为后面要用到反幂法，所以这里的对角元要加上 0.48，这样用反幂法就可以求得绝对值最小的特征值，然后再减去 0.48 就可以得到真正的基态能量。

反幂法可以求特征值最小的特征值及其对应的特征向量，如果我们队原有矩阵减去一个 λI ，其中 λ 是近似特征值，我们对这个矩阵运用反幂法，求出来的就是对于 λ 的特征向量。在求解过程中，我们使用 LU 分解即可。反幂法的迭代过程是在一个循环中，不断解 $LUv=u$ ，解出 v 后求 v 向量中最大的分量，令其为 λ_n ，然后让 $u = v\lambda_n$ ，这样不断循环就能使最终的 u 趋近于特征向量，在这里我们只是用追逐法解方程替代 LU 分解。

用反幂法求得了波函数之后，需要对波函数进行一次归一化处理，乘一个系数，最后输出基态能量，将波函数画图即可。具体的三对角系数矩阵求解线性方程组问题的处理方法见下图（课堂讲义）。

三对角方程组的追赶算法

前面的讨论是基于 $\mathbf{A} = \mathbf{L}\mathbf{U}_0$ 的，而下面给出的基于 $\mathbf{A} = \mathbf{L}_0\mathbf{U}$ 的真实算法：

输入: $n, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{f}$; 输出: \mathbf{x} :

For $i = 2, 3, \dots, n$

$$m_i := a_i/b_{i-1};$$

$$b_i := b_i - m_i c_{i-1};$$

$$f_i := f_i - m_i f_{i-1};$$

END

$$x_n := f_n/b_n;$$

For $i = n-1, n-2, \dots, 1$

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

End

追赶法(托马斯法)：解三对角线性方程组

三对角线性方程组: $\mathbf{Ax} = \mathbf{f}$, 矩阵形式为:

$$\begin{bmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ \vdots \\ f_{n-1} \\ f_n \end{bmatrix} \quad (32)$$

系数矩阵为三对角矩阵，非零元素分布在主对角线及其相邻两条次对角线上。储存时，可以用三个向量 $\mathbf{a}, \mathbf{b}, \mathbf{c}$ 分别表示非零元素：

$\mathbf{a} = (a_2, a_3, \dots, a_n); \mathbf{b} = (b_1, b_2, b_3, \dots, b_n); \mathbf{c} = (c_1, c_2, c_3, \dots, c_{n-1})$ 。

我们要寻求三对角方程组的最高效的算法。

具体代码实现如下：

```
1 h = 0.1 # 步长
2 boundary = 2000 # 求解的x_max
3 n = int(2*boundary/h)+1 # 区间数
4 a = []
5 b = []
6 c = []
7 for i in range(n): # 初始化追逐法用到的三列向量
8     b.append(1/h**2 + V(-boundary+i*h)+0.48)
9     a.append(-0.5/h**2)
10    c.append(-0.5/h**2)
11 v = []
12 x = []
13 for i in range(n): # 初始化反幂法用到的解向量v以及画图时要用到的坐标
14     向量x
15     v.append(random.random())
16     x.append(-boundary+i*h)
17 u = deepcopy(v)
18 iteration = 1
19 while iteration == 1 or math.fabs(max_num(listminusfunction(u1, u))) >
20     1E-14: # 运用反幂法求解，其中LU分解解方程的部分已经化为追逐法
21     u1 = u
22     v = chasesolvingfunction(n, a, b, c, u)
23     lam = 1 / max_num(v)
24     u = listnum_multiply(lam, v)
25     iteration += 1
26 # 得到u, 特征向量, lam是特征值
27 sum_u = 0
28 for i in range(len(u)):
29     sum_u += u[i]*u[i]
30 for i in range(len(u)): # 对得到的基态波函数进行归一化
31     u[i] = u[i]*u[i]/sum_u/h
32 print('得到的氢原子基态能量为：', lam-0.48)
33 plt.plot(x, u)
34 plt.xlabel('x')
35 plt.ylabel('概率')
36 plt.title('氢原子基态时在各处出现的概率分布图')
# plt.legend()
37 plt.show()
```

1.1.4 结果展示

得到的氢原子基态能量为： -0.5000344989700516， 相对误差大概 4.2%

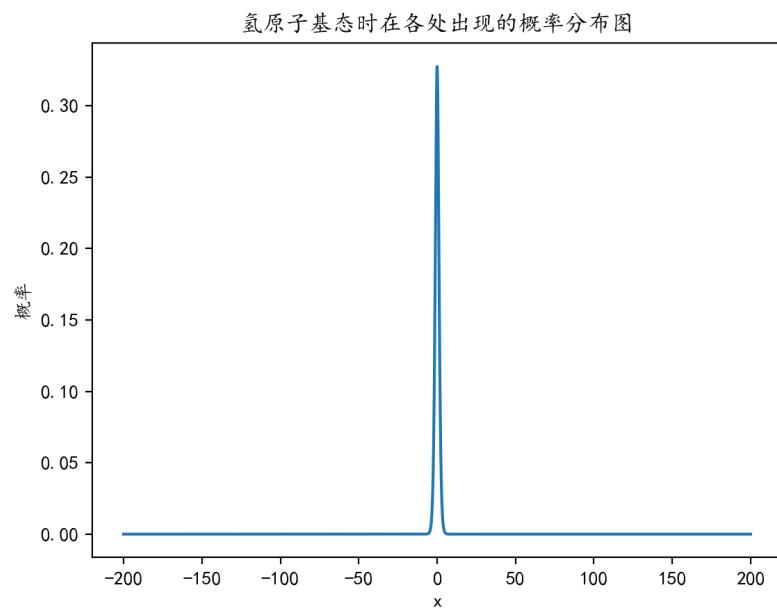


图 1.1: $x=-200-200$, 步长 =0.01

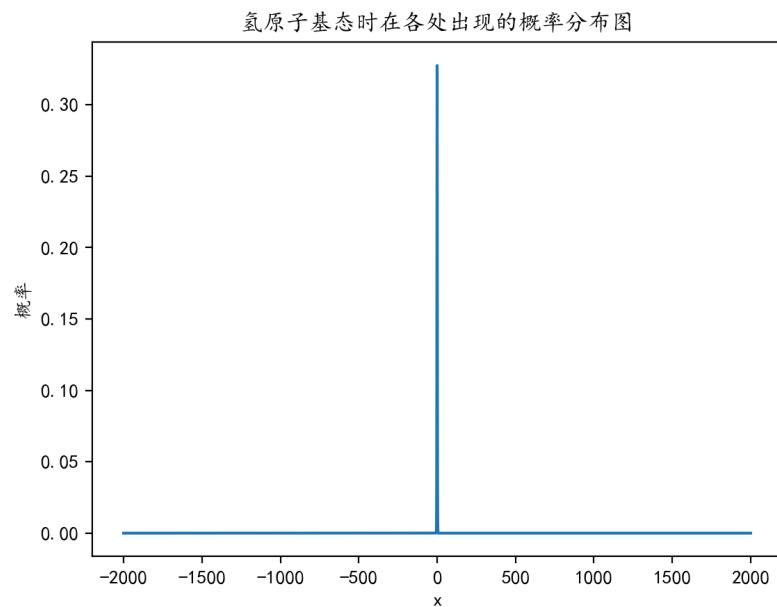


图 1.2: $x=-2000-2000$, 步长 =0.1

1.2 第二问

1.2.1 题目

(2) 在外加阿秒激光场作用下 (光强 $I = 1 \times 10^{16} W/cm^2$, 频率 $\omega = 1 a.u.$, 脉冲周期数 $N = 18$) , 请计算氢原子1s态布居数随时间的演化 $P_0(t) = |\langle \psi_0(x) | \psi(x, t) \rangle|^2$ 和其电离的末态电子动量谱 $P(k) = |\langle \psi_k | \psi_f \rangle|^2$, 其中 $\psi_f = \psi(t_f) - p\psi_0$, 而 $p = \langle \psi_0 | \psi(t_f) \rangle$ 为系统留在基态的几率幅; 当然 $\psi(t_f)$ 表示初态 ψ_0 在激光场中演化完毕后的整个波函数 (包含电离部分和留在基态的部分) , k 对应电子动量, 取值范围取为 $[-2.5, 2.5]$ 。请分别用线性纵坐标和对数纵坐标画图 $P(k)$ 。

1.2.2 思路

这一题是解强激光场下电子的行为, 就是在第一问的基础上添加了一个随时间变化的电场, 这样薛定谔方程就变成了含时的, 波函数的初始状态就是第一问中解得的基态波函数, 求解的方程为:

$$[-\frac{1}{2} \frac{d^2}{dx^2} + V(x) + xE(t)]\psi(x, t) = i \frac{\partial}{\partial t} \psi(x, t)$$

其中电场部分为:

$$E(t) = \sqrt{I} \sin^2(\frac{\omega t}{2N}) \sin(\omega t + \phi_0)$$

开始时刻为 0, 结束时刻为 $2N\pi/\omega$, 其中 $I = 1 \times 10^{16} W/cm^2$, $\omega = 1 a.u.$, $N = 18$

解含时的薛定谔方程要既满足稳定性要求又满足么正性要求, 应当采用对 e^{-iHt} 离散的 Cayley 格式, 即:

$$(1 + iH\Delta t/2)\psi_j^{m+1} = (1 - iH\Delta t/2)\psi_j^m$$

实际上就是 Crank-Nicolson 方法, 使用这个方法可以解出波函数随时间的演化, 即 $\psi(x, t)$

求氢原子的布居数随时间的演化 $P_0(t) = |\langle \psi_0(x) | \psi(x, t) \rangle|^2$

及其电离的末态电子动量谱 $P(k) = |\langle \psi_k | \psi_f \rangle|^2$, 其中 $\psi_f = \psi(t_f) - p\psi_0$, $p = |\langle \psi_0 | \psi(t_f) \rangle|$, $\psi_k = e^{ikx}$, k 对应电子动量, 取值范围为: $[-2.5, 2.5]$ 。

1.2.3 程序实现

程序的前半部分与第一问一样, 此处不过多赘述。解出波函数 u 之后的步骤为: 首先确定时间步长和包络数 N , y 是随时间变化的波函数, 赋予它初值 u , 然后定义一个 dot , 用于最后常量计算。主程序最开始需要初始化时间列表和布局数列表, 根据此前引用课堂讲义对三对角矩阵的 a, b, c 列的定义, 演化过程矩阵中 a 和 c 两列不随时间变化, 所以提前初始化好即可。而 b 列会随着时间变化, 所以在时间循环中需要不断更新该列。分为 1 和 2 两个是因为有左侧乘矩阵和右侧乘矩阵, 左侧乘矩阵标记为 1, 右侧乘矩阵定义为 2。对于演化过程中的每一时刻, 我们都需要用追逐法接触波函数, 然后求一个布局数, 加到向量尾部即可。程序如下:

```

1 a1 = []
2 c1 = []
3 a2 = []
4 c2 = []
5 dt = 0.05 # 时间步长
6 N = 18
7 t_list = [] # 初始化时间列表
8 p_list = [] # 初始化
9 y = deepcopy(u) # y的初始态是u
10 dot = complex(0, 1)*dt/2 # 后面用到的一个量i delta t/2
11 y_origin = deepcopy(u) # 基态的波函数
12
13 for i in range(n):
14     a1.append(-0.5 / h ** 2 * dot)
15     c1.append(-0.5 / h ** 2 * dot)
16     a2.append(0.5 / h ** 2 * dot)
17     c2.append(0.5 / h ** 2 * dot) # 初始化三对角矩阵, 1是更新的时间下
18     # 标, 2是原时间下矩阵,
19 for t in range(int(2*N*math.pi/dt)): # t不是真实时间
20     b1 = []
21     b2 = []
22     t_list.append(t * dt)
23     for i in range(n):
24         b1.append((1 / h ** 2 + V(-boundary + i * h) + (-boundary + i
25             * h)*E((t+1)*dt, N))*dot + 1) # 现时刻的矩阵
26         b2.append(-(1 / h ** 2 + V(-boundary + i * h) + (-boundary + i
27             * h)*E(t*dt, N))*dot + 1) # 原时刻的矩阵
28     y_old = [b2[0] * y[0] + c2[0] * y[1]]
29     for i in range(1, n-1):
30         y_old.append(a2[i] * y[i-1] + b2[i] * y[i] + c2[i] * y[i+1])
31     y_old.append(a2[n - 1] * y[n - 2] + b2[n - 1] * y[n - 1]) # 计算
32     # 演化时右侧的矩阵乘向量
33     y = chasesolvingfunction(n, a1, b1, c1, y_old) # 追逐法求y
34     p = 0 # p是最后的求和
35     for i in range(n):
36         p += y[i] * y_origin[i] * h
37     p_list.append(abs(p)**2)
38     print('布居数已完成: ', t/(2*N*math.pi/dt)*100, '%')
39 plt.plot(t_list, p_list)
40 plt.ylabel('P')
41 plt.xlabel('t')
42 plt.title('布居数随时间变化图')
43 plt.show() # 画布居数的图

```

完成上面这一部分后，只需要求末态电离动量分布， p 值直接用上一部分的就好，先求出 ψ_f ，然后确定 k 的步长，不断地根据 k 求出 P ， $P(k) = |\langle \psi_k | \psi_f \rangle|^2$ ，最后画图即

可。代码如下：

```
1 psi_f = []
2 for i in range(n):
3     psi_f.append(y[i]-p*y_origin[i])
4 P_k = []
5 dk = 0.01
6 k_list = []
7 for k in range(int(5/dk)+1):
8     sum_k = 0
9     k_list.append(k*dk-2.5)
10    for i in range(n):
11        sum_k += (math.cos((k*dk-2.5)*(-boundary + i * h))-complex
12            (0,1)*math.sin((k*dk-2.5)*(-boundary + i * h)))*psi_f[i]*h
13    P_k.append(abs(sum_k)**2)
14    print('动量谱已完成：', k / (5/dk) * 100, '%')
15 plt.plot(k_list, P_k)
16 plt.ylabel('P(K)')
17 plt.xlabel('k')
18 plt.title('电离电子动量谱')
19 plt.show() # 画动量谱的图
20 for i in range(0, len(P_k)):
21     P_k[i] = math.log10(P_k[i])
22
23 plt.plot(k_list, P_k)
24 plt.ylabel('logP(K)')
25 plt.xlabel('k')
26 plt.title('电离电子动量谱（对数坐标）')
27 plt.show() # 画动量谱的图
```

1.2.4 结果展示

得到布局数随时间变化的图：

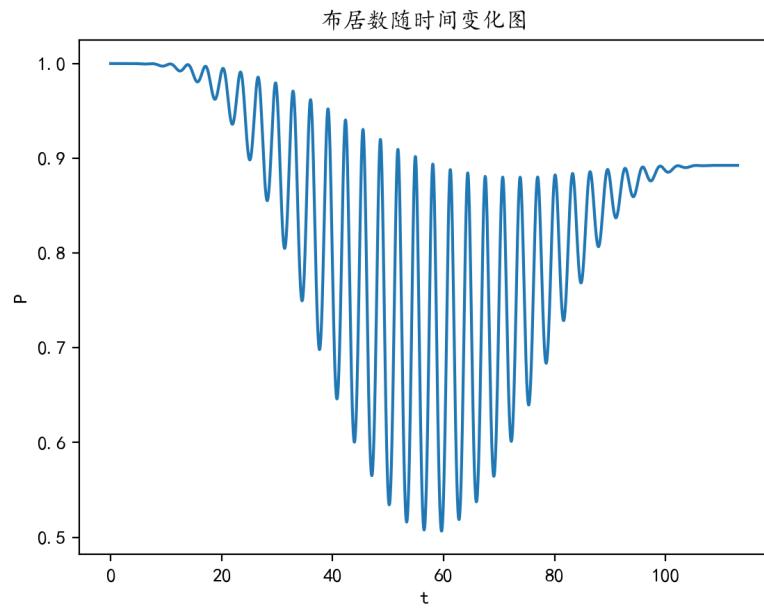


图 1.3: 布局数随时间变化图

得到末态电子动量谱（普通坐标）的图：

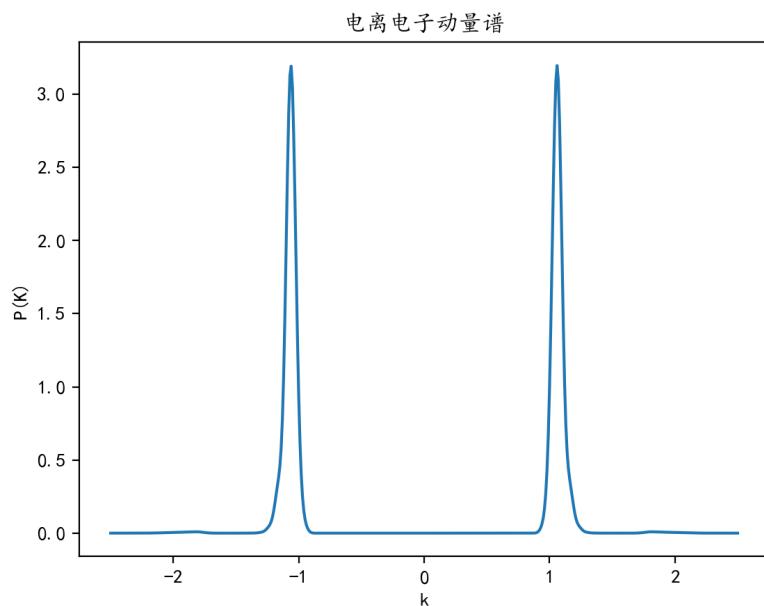


图 1.4: 电离电子动量谱（普通坐标）

得到末态电子动量谱（对数坐标）的图：

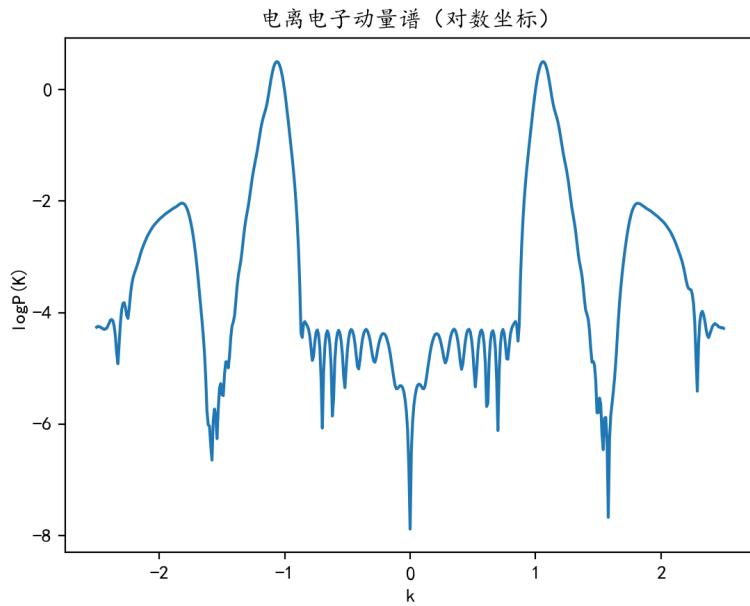


图 1.5: 电离电子动量谱 (对数坐标)

1.3 第三问

1.3.1 题目

(3) 在外加激光场作用下 (光强 $I = 2 \times 10^{14} W/cm^2$, 波长 $\lambda = 300 nm$, 脉冲周期数 $N = 48$) , 给出用偶极矩和加速度两种方法分别计算的高次谐波功率谱 (谐波谱横坐标单位统一为谐波阶次, 即 $\frac{\omega}{\omega_{300nm}}$, 到 15 阶即可, 纵坐标为 $\log_{10}(|A(\omega)|^2)$), 参考提示 (4) 的公式) ;

1.3.2 思路

在外加激光场作用下，求解高次谐波功率谱，外加场仍然为电场，只不过参数与上一问有所不同，光强 $I = 2 \times 10^{14} W/cm^2$, 波长 $\lambda = 300nm$, 脉冲周期 $N=48$, 载波包络相位 $\phi_0 = 0$, 将新的参数输入进原方程，求谐波功率谱时用两种方法，一种是偶极矩：

$$d(t) = \langle \psi(x, t) | x | \psi(x, t) \rangle$$

$$A(\omega) = \frac{1}{\sqrt{2\pi}} (-\omega^2) \int_0^{t_f} e^{-i\omega t} d(t) dt$$

另外一种是加速度：

$$a(t) = \langle \psi(x, t) | -\frac{d}{dx} V(x) + E(t) | \psi(x, t) \rangle$$

$$A(\omega) = \frac{1}{\sqrt{2\pi}} \int_0^{t_f} e^{-i\omega t} a(t) dt$$

为了减少运算，我首先计算各个时间下的波函数，做成一个波函数序列，然后再用后面提供的公式进行计算，算出 $A(\omega)$ ，最后将 $\frac{\omega}{\omega_{300nm}}$ 当横坐标，将 $\log_{10}(|A(\omega)|^2)$ 当纵坐标，求出谐波功率谱分布。

由于高次谐波的计算中，边界会进行反射从而掺杂一些虚假信号，所以在边界处引入高斯型吸收函数 $f(x)$ 与波函数相乘

$$f(x) = \begin{cases} \exp\left[-\frac{(|x| - x_0)^2}{\sigma_x^2}\right] & x_0 < |x| < x_{max} \\ 1 & |x| < x_0 \end{cases}$$

其中 $\sigma_x = 0.2$, $x_0 = 0.75x_{max}$ 每次传播都要进行吸收。

1.3.3 程序实现

得到基态波函数之前的部分不再赘述，和第一问中的做法完全相同。下面是一些第三问中新加入的关键函数。 $a_function$ 和 $d_function$ 是用来计算 $a(t)$ 和 $d(t)$ 两个序列的函数，输入时间和位置后，可以得到用来积分计算的核心量，即位于右矢和左矢中间的函数表达，这个量将会被带入在 d_or_a 函数中，具体体现为：

$$d(t) = \langle \psi(x, t) | -\frac{d}{dx} V(x) + E(t) | \psi(x, t) \rangle$$

$$d(t) = \langle \psi(x, t) | x | \psi(x, t) \rangle$$

我们通过 d_or_a 函数对狄拉克表示进行积分，从而得到 $a(t)$ 和 $d(t)$ 两个序列。具体是 a 还是 d ，取决于输入函数变量是 $d_function$ 还是 $a_function$ 。后面的 $gauss_absorb$ 函数是用来求高斯吸收序列的，乘在求出的波函数序列上。 psi_xt 函数用来求各个时刻的波函数并保存起来以方便后面继续算 d_or_a 函数。代码如下：

```

1 def a_function(t, i): # 求a (t) 的函数
2     global h
3     global boundary
4     global dt
5     global N

```

```

    global omega_light
7   x = (-boundary + i * h)
8   t_real = t*dt
9   return (-x/((2+x**2)**(1.5)))+E2(t_real, N, omega_light)

11
11 def d_function(t, i): # 求d(t)的函数
12   global h
13   global boundary
14   return -boundary + i * h

17 def cons_vector(lam, v):
18   re = [lam * v[i] for i in range(len(v))]
19   return re

21 def normalization(v, dx): # 归一
22   sum = 0
23   for i in range(len(v)):
24     sum += v[i]**2
25   con = 1/math.sqrt(sum * dx)
26   return cons_vector(con, v)
27

29 def d_or_a(y, function, N, dt): # 本函数用来求解d或者a
30   global omega_light
31   global h
32   d_list = [] # 初始化, d是t的函数
33   d = 0 # d是最后的求和
34   for t in range(int(2 * N * math.pi / omega_light / dt)):
35     for i in range(n):
36       d += y[t][i].conjugate() * function(t, i) * y[t][i] * h
37   d_list.append(d) # 得到t关于时间的函数
38   return d_list
39

41 def gauss_absorb(u): # 高斯吸收序列
42   global h
43   global boundary
44   gauss = []
45   for i in range(len(u)):
46     x = -boundary + i * h
47     if abs(x) > 0.75 * boundary:
48       gauss.append(math.exp(-(abs(x)-0.75*boundary)**2/0.04))
49     else:
50       gauss.append(1)
51   return gauss

```

```

53
def psi_xt(u, dt, N, omega_light, gauss): # 本函数用来求解波函数,输出
    的长度是int(2 * N * math.pi /omega_light/ dt)
55    global boundary
56    y = [[] for t in range(int(2 * N * math.pi /omega_light/ dt))]
57    y[0] = deepcopy(u) # 引入第1问中得到的波函数
58    dot = complex(0, 1) * dt / 2 # 后面用到的一个量i delta t/2
59    a1 = []
60    c1 = []
61    a2 = []
62    c2 = []
63    for i in range(n):
64        a1.append(-0.5 / h ** 2 * dot)
65        c1.append(-0.5 / h ** 2 * dot)
66        a2.append(0.5 / h ** 2 * dot)
67        c2.append(0.5 / h ** 2 * dot) # 初始化三对角矩阵, 1是更新的时
68        间下标,2是原时间下矩阵
69    for t in range(int(2 * N * math.pi /omega_light/ dt) - 1): # t不
70        是真实时间
71        sum_y=0
72        b1 = []
73        b2 = []
74        for i in range(n):
75            b1.append((1 / h ** 2 + V(-boundary + i * h) + (-boundary
76                + i * h) * E2((t + 1) * dt, N, omega_light)) * dot + 1)
77            # 现时刻的矩阵
78            b2.append(-(1 / h ** 2 + V(-boundary + i * h) + (-boundary
79                + i * h) * E2(t * dt, N, omega_light)) * dot + 1) # 原
80                时刻的矩阵
81        y_old = [b2[0] * y[t][0] + c2[0] * y[t][1]]
82        for i in range(1, n - 1):
83            y_old.append(a2[i] * y[t][i - 1] + b2[i] * y[t][i] + c2[i]
84                * y[t][i + 1])
85        y_old.append(a2[n - 1] * y[t][n - 2] + b2[n - 1] * y[t][n -
86            1]) # 计算演化时右侧的矩阵乘向量
87        y[t+1] = chasesolvingfunction(n, a1, b1, c1, y_old) # 追逐法
88        求y(时间t时)
89        for i in range(n): # 计算吸收
90            y[t+1][i] = (y[t+1][i] * gauss[i])
91        print('波函数求解已完成', t/(2*N*math.pi/omega_light/dt)*100,
92            '%')
93    return y

```

后面是主函数部分,首先取一些参量,时间步长取0.05,x间隔(空间间隔)0.1,取然后求出高斯吸收序列以便后续使用。接着求出波函数矩阵,将波函数矩阵输入到d_or_a函数序列,通过计算求出a(t)和d(t)两个序列,然后分别求解两种方法下的谐波功率

谱。求解时根据公式对 t 进行积分即可得到 ω 下的功率，对不同的 ω 画图（频率间隔为 0.005）即可。两种方法（偶极矩和电子加速）具体积分公式如下：

$$A(\omega) = \frac{1}{\sqrt{2\pi}}(-\omega^2) \int_0^{t_f} e^{-i\omega t} d(t) dt \quad \text{偶极矩方法}$$

$$A(\omega) = \frac{1}{\sqrt{2\pi}} \int_0^{t_f} e^{-i\omega t} a(t) dt \quad \text{加速度方法}$$

```

1 omega_300 = 45.5633525316/300
2 N = 48
3 lam = 300
4 omega_light = 45.5633525316/lam
5 dt = 0.05 # 时间步长
6 gauss = gauss_absorb(u)
7 y = psi_xt(u, dt, N, omega_light, gauss)# 得到的y是xt的函数，即各个时
刻的波函数

9 # 下面部分是画偶极矩下的谐波功率谱
10 d_list = d_or_a(y, d_function, N, dt)
11 d_ome = 0.005 # ome的步长
12 ome_list = []
13 A_list = []

15 for ome in range(1, int(15*omega_300/d_ome)): # 对不同频率积分
16     ome_real = ome*d_ome
17     ome_list.append(ome_real/omega_300)
18     A = 0
19     for t in range(int(2*N*math.pi/omega_light/dt)):
20         A += complex(math.cos(ome_real * t * dt), -math.sin(ome_real*t
21                         * dt))*d_list[t]*dt
22     A_list.append(math.log10((A.real**2+A.imag**2)*abs((1/math.sqrt(2*
23                         math.pi))*(-ome_real**2))**2))
24     print('偶极矩已完成', ome/(15*omega_300/d_ome)*100, '%')
25 plt.plot(ome_list, A_list, label='偶极矩')
26 plt.ylabel('lg |A(omega)|^2')
27 plt.xlabel('omega/omega300')
28 plt.title('谐波辐射频谱')

29 # 下面部分是画电子加速度的谐波功率谱
30 a_list = d_or_a(y, a_function, N, dt)
31 print(a_list[2])
32 d_ome = 0.005 # ome的步长
33 ome_list = []
34 A_list = []
35 for ome in range(1, int(15*omega_300/d_ome)): # 对不同频率积分
36     ome_real = ome*d_ome
37     ome_list.append(ome_real/omega_300)
38     A = 0

```

```

37     for t in range(int(2*N*math.pi/omega_light/dt)):
38         A += (math.cos(ome_real * t * dt)-complex(0,1)*math.sin(
39             ome_real*t * dt))*a_list[t]*dt
40     A_list.append(math.log10((A.real**2+A.imag**2)/(2 * math.pi)))
41     print('电子加速已完成', ome / (15 * omega_300 / d_ome) * 100, '%')
42 plt.plot(ome_list, A_list, label='电子加速')
43 plt.legend()
44 plt.yticks([])
45 plt.show() # 画两个的图

```

1.3.4 结果展示

以下是两种方法计算的谐波功率谱：（分别对应于 $x_{max} = 10$ 和 $x_{max} = 2000$ ）

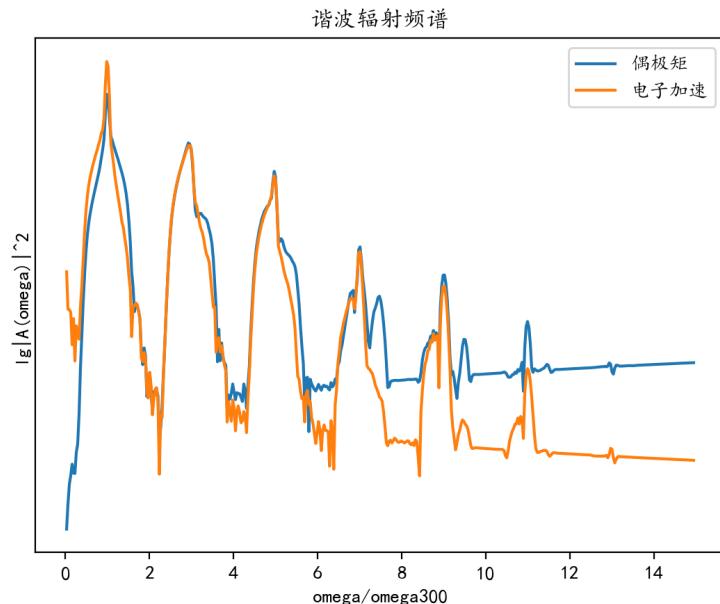


图 1.6: $x_{max} = 10$

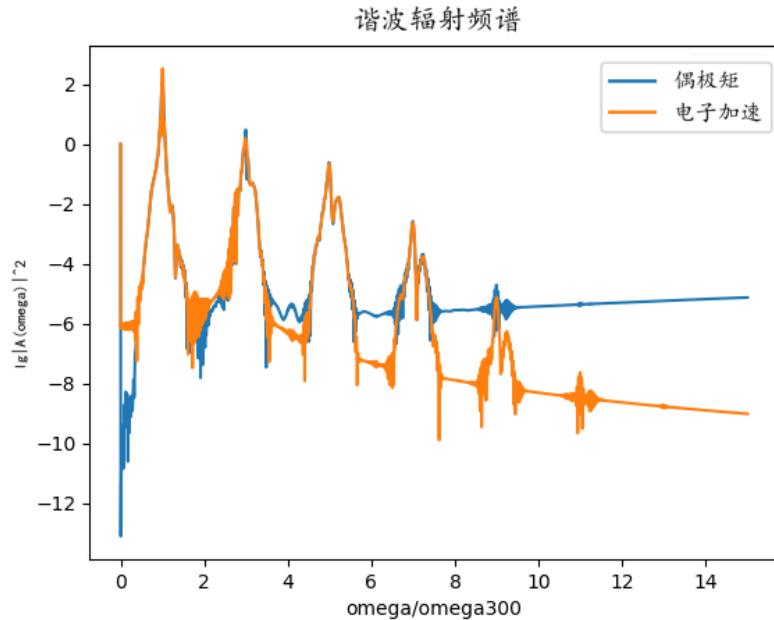


图 1.7: $x_{max} = 2000$

1.4 第四问

1.4.1 题目

(4) 在外加激光场作用下 (光强 $I = 1 \times 10^{14} W/cm^2$, 波长 $\lambda = 400 nm$, 脉冲周期数 $N = 4$, 载波包络相位 $\phi_0 = 0$) , 通过时频分析给出具有时间分辨率的谐波功率谱分布 $\log_{10}(|A(t, \omega)|^2)$ (谐波阶次最高到 20 阶即可) , 参考提示 (5) , 此小题 x_{max} 取 600 即可。

1.4.2 思路

这一问利用了第三问中的结果, 求具有时间分辨率的谐波功率谱分布: 首先是参数稍有变动, 光强 $I = 1 \times 10^{14} W/cm^2$, 波长 $\lambda = 400 nm$, 脉冲周期数 $N=4$, 载波包络相位 $\phi_0 = 0$, 将参数重新输入, 然后根据前面的步骤求得 $a(t)$, x_{max} 取 600, 求有时间分辨率的谐波功率谱时, 利用:

$$A(t_0, \omega) = \int_0^{t_f} a(t) W_{t_0, \omega}(t) dt$$

其中窗函数:

$$W_{t_0, \omega}(t) = e^{-i\omega t} e^{-\frac{(t-t_0)^2}{2\sigma^2}}$$

根据老师建议, σ 取 15, 计算结果并画图, 由于有两个纬度的变量, 因此我采用等高线类色度图, 经过两个 for 循环, 在不同 ω 和 t 下求 $\log(|A(t, \omega)|^2)$ 。在这里由于要画图的

原因所以用到了 numpy。

1.4.3 程序实现

前面已有的步骤不再放上来了，以下程序从计算出 $a(t)$ 开始，接着在一个 ω 循环和 t_0 循环中直接计算 $\log_{10}(|A(t, \omega)|^2)$ ， t_0 的步长取 0.2，从 0 到激光结束。程序如下：

```
omega_300 = 45.5633525316/300
2 N = 4
lam = 400
4 omega_light = 45.5633525316/lam
dt = 0.1 # 时间步长
6 gauss = gauss_absorb(u)
y = psi_xt(u, dt, N, omega_light, gauss) # 得到的y是xt的函数，即各个
    时刻的波函数
8 a_list = d_or_a(y, a_function, N, dt)
t_0_list = []
10 d_ome = 0.01 # ume的步长
dt_0 = 2
12 A_list = [[ for t_0 in range(int(2 * N * math.pi / omega_light/dt_0)) ]
ome_list = []
14 for ume in range(1, int(20*omega_300/d_ome)):
    ume_real = ume * d_ome
    ume_list.append(ume_real / omega_300)
    for t_0 in range(int(2 * N * math.pi / omega_light / dt_0)): # 对不同
        时刻循环
        t_0_list.append(t_0*dt_0)
        for ume in range(1, int(20*omega_300/d_ome)): # 对不同频率循环
            ume_real = ume*d_ome
            A = 0
            for t in range(int(2*N*math.pi/omega_light/dt)):
                A += complex(math.cos(ume_real * t * dt), -math.sin(
                    ume_real*t * dt))*a_list[t]*dt*math.exp(-(t*dt-t_0*dt_0)
                )**2/15**2)
            A_list[t_0].append(math.log10(abs(A)**2))
            print('功率谱已完成', t_0 / (2 * N * math.pi / omega_light/dt_0) *
100, '%')
26 fig2 = plt.figure(2)
surf2 = plt.contourf(ome_list, t_0_list, A_list)
fig2.colorbar(surf2)
plt.title('具有时间分辨率的谐波功率谱分布')
30 plt.xlabel('omega/omega300')
plt.ylabel('t')
32 plt.show()
```

1.4.4 结果展示

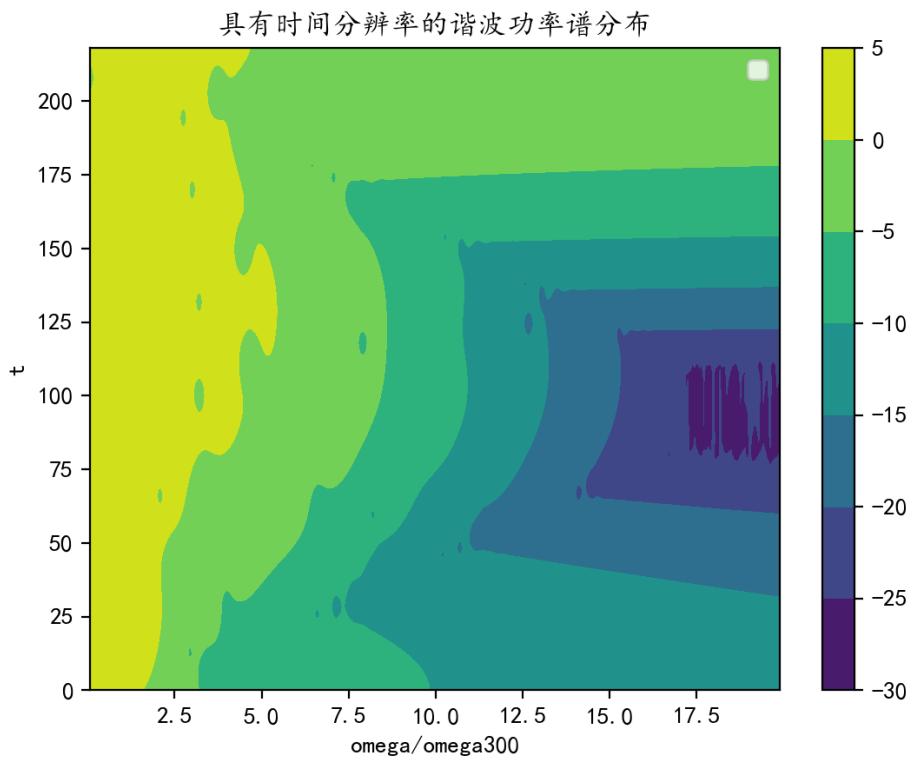


图 1.8: 有时间分辨率的谐波功率谱, 每一个点颜色深浅按照色标表示这个点 (ω,t) 对应的功率谱幅值大小关系

Chapter 2

第二题

2.1 第一问

2.1.1 题目

第一步，在电场存在的任何时刻，束缚电子会遂穿成为自由电子，该自由电子的初始位置距离坐标原点反比于电场强度大小， $|r_0| = I_p/E$ ，其中氢基态原子电离势 $I_p = 0.5$ ，方向为激光电场的反方向；电子平行于电场方向初速度为0，但是具有垂直于电场方向的初速度 v_\perp 。电子的遂穿几率 W 与电场大小以及横向初速度相关，

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

利用蒙特卡洛方法，生成具有上式分布的电子样品。

2.1.2 思路

这个题目的思路就是从根据随机数生成一系列电子样品，他们的概率分布符合隧穿几率，即：

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

采用舍选法即可首先确定变量的范围， $E = -\frac{\partial A}{\partial t}$ ，根据题目所给的A，可以求出E的最大值在t=0处取得，大小为 $\omega A_0 = 0.076$ ，所以E在0到0.076取值，垂直于电场速度没有给取值范围，但是因为是束缚原子，动能小于势能，所以初速度应小于1，所以我们选择v在0到1之间。

根据舍选法，我们先求出这个范围上隧穿几率最大的点，求出他的几率，然后随机生成E和v，再额外生成一个0到1之间的随机变量，如果这个变量小于归一化的隧穿几率，那么选取，反之舍弃。W的极值在 $E = 1/3, v = 0$ 时取得，但是无法到达 $E = 1/3$ ，在给定范围内最大几率在 $E = 0.076, v = 0$ 处取得，用此值进行归一化即可。

为了表示电子样品，我用了两种图，一种是概率密度随E和v变化的情况，另外一个是v随时间的分布结果。

2.1.3 程序实现

此题程序实现上比较简单，E_list 和 v_list 是用来储存选取的 E 和 v 的列表，进行 500000 次循环后，将得到的电子画在 E-v 图上，另外为了表示在不同时刻的电子样品，我进行一个时间循环，将不同时间电场求出来进行速度抽样，就可以获得速度样品随时间的变化结果。

```
def E(t): # 线偏振电场函数
    omega = 0.057
    E = -1.325 * omega * (
        0.5 * math.cos(omega * t) + 0.5 * math.cos(omega * t) *
        math.cos(omega * t / 4) - 1 / 8 * math.sin(
            omega * t) * math.sin(omega * t / 4))
    return E

def E_ellipse(t): # 椭圆偏振电场函数
    omega = 0.057
    E_x = -1/math.sqrt(1.25) * 1.325 * omega * (
        0.5 * math.cos(omega * t) + 0.5 * math.cos(omega * t) *
        math.cos(omega * t / 4) - 1 / 8 * math.sin(
            omega * t) * math.sin(omega * t / 4))
    E_y = -0.5 / math.sqrt(1.25) * 1.325 * omega * (
        -0.5 * math.sin(omega * t) - 0.5 * math.sin(omega * t) *
        math.cos(omega * t / 4) - 1 / 8 * math.cos(
            omega * t) * math.sin(omega * t / 4))
    return E_x,E_y

def species(E): # 对于固定的E选取一些v的样本
    max = 1 / 0.076 ** 2 * math.exp(-(2 / 3) / 0.076) # max是全局的
    v_list = [] # 初始化列表
    for iter_a in range(1000):
        Chisi = random.random() # 随机生成的参考向量
        v = random.uniform(-1, 1) # 随机选取v, 在-1到1之间
        W = 1 / E ** 2 * math.exp(-(2 / 3 + v ** 2) / E) # 隧穿几率
        if W / max > Chisi: # 判定是否选取
            v_list.append(v)
    return v_list # 返回一系列v的样本

t1_list = []
t2_list = []
v1_list = []
v2_list = []
omega = 0.057
```

```

38 resolution = 10000
t_0 = -4*math.pi/omega
40 for n in range(resolution): # 对于时间t做循环, 计算末状态
    t = t_0 + 8*math.pi/omega * n / resolution
    print('已完成计算: ', n/resolution*100, '%')
    E_1 = abs(E(t))
44 E_2x, E_2y = E_ellipse(t)
E_2 = math.sqrt(E_2x**2+E_2y**2)
46 if E_1 > 0.03:
    v_ver1 = species(E_1)
48 else:
    v_ver1 = []
49 if E_2 > 0.03:
    v_ver2 = species(E_2)
52 else:
    v_ver2 = []
54 for i in range(len(v_ver1)):
    t1_list.append(t)
56 v1_list = v1_list + v_ver1
58 for i in range(len(v_ver2)):
    t2_list.append(t)
v2_list = v2_list + v_ver2
60 plt.figure(1)
plt.scatter(t1_list, v1_list, s=1)
62 plt.xlabel('t')
plt.ylabel('v')
64 plt.title('line')
plt.show()
66 plt.figure(2)
plt.scatter(t2_list, v2_list, s=1)
68 plt.xlabel('t')
plt.ylabel('v')
70 plt.title('ellipse')
plt.show()
72
max = 1/0.076**2*math.exp(-(2/3)/0.076) # 最大概率
74 E_list = []
v_list = [] # 初始化列表
76 for iter_a in range(500000):
    Chisi = random.random() # 随机生成的参考向量
78     E = random.uniform(0, 0.076)
    v = random.uniform(0, 1)
80     W = 1/E**2*math.exp(-(2/3+v**2)/E) # 隧穿几率
    if W/max > Chisi: # 判定是否选取
82         E_list.append(E)
            v_list.append(v)
84 plt.figure(3)

```

```
86 plt.scatter(E_list, v_list, s=1)
87 plt.xlabel('E')
88 plt.ylabel('v')
89 plt.title('电子随电场和垂直电场速度分布图')
90 plt.show()
```

2.1.4 结果展示

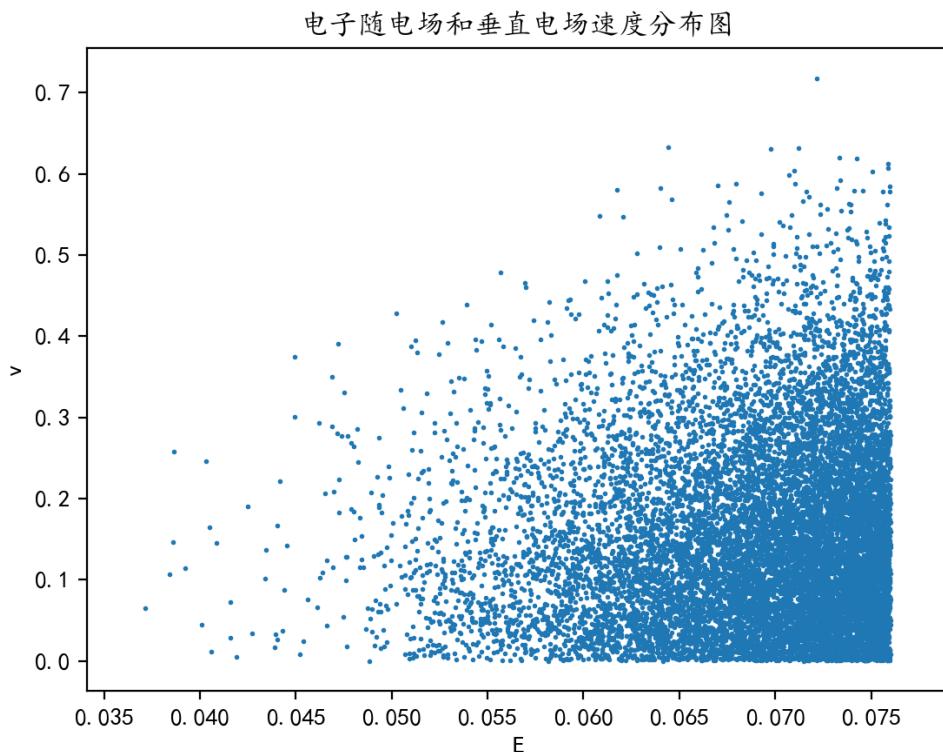


图 2.1: 电子隧穿样品的 E 和 v_{\perp}

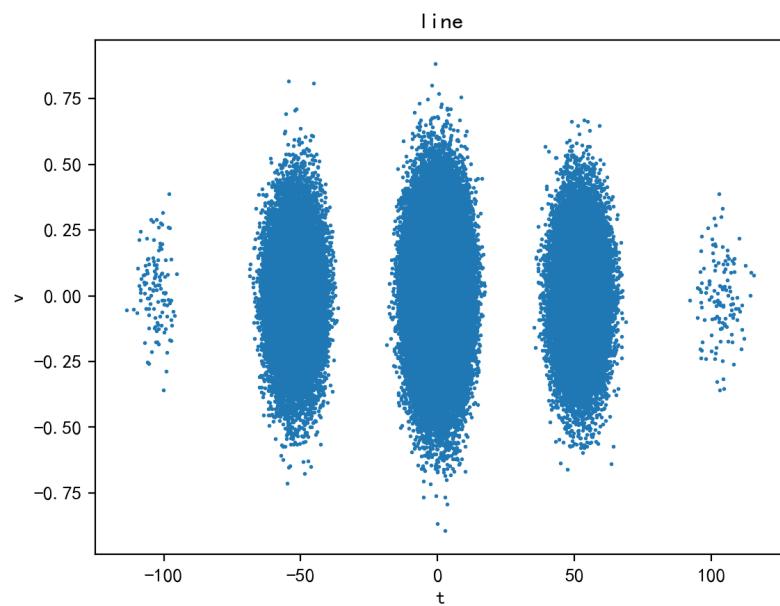


图 2.2: 电子隧穿样品线偏振的 v_{\perp} 随 t 分布

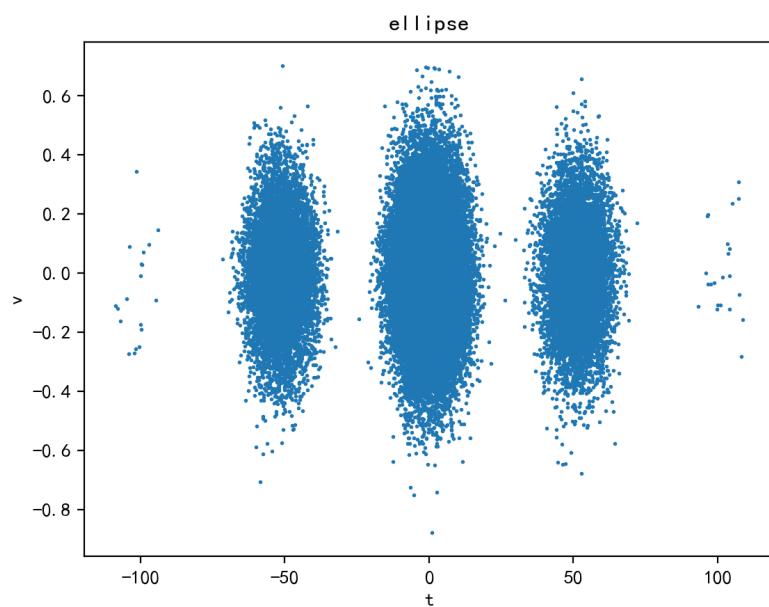


图 2.3: 电子隧穿样品椭圆偏振的 v_{\perp} 随 t 分布

2.2 第二问

2.2.1 题目

第二步，每个电子的初始条件前面已经给出，接下来求解牛顿方程来计算电子在库仑力和电场力共同作用下的运动，请自行写出微分方程。氢原子的库伦势为 $-1/r$ ，做为练习我们仅仅考虑二维的情况，电子仅在xy平面内运动（z方向的位置以及速度均为零），且为了避免库伦势的奇点对你们的数值求解造成一定困难，我们将库伦势做软化，即取 $r = \sqrt{x^2 + y^2 + 0.2^2}$ ，避免了部分电子运动到接近坐标轴原点时库仑力的发散。从电离时刻开始，求解给定初始条件的微分方程组，得到激光结束时刻的电子位置 \vec{r}_f 和速度 \vec{v}_f 。

2.2.2 思路

在第一步中已经给定了电子电离后，初始位置为： $r_0 = I_p/E$ ，具有垂直于电场方向的初速度 v_\perp ，然后按照牛顿力学受电场力（分为激光电场合库仑力）作用积分即可，直到激光时间结束。

2.2.3 程序实现

总的思路很清晰，首先写四个函数，分为线偏振和椭圆偏振，首先有两个函数，来输入一个时间计算电场强度，线性偏振的输出一个值，椭圆偏振的输出两个值，计算公式就是根据：

$$E = -\frac{\partial A}{\partial t}$$

然后又有两个函数分别计算两种偏振中的粒子运动过程，为line和ellipse函数，在这两个函数中，首先计算时间t中的电场强度大小，然后根据电场强度大小计算初始距离 $r_0 = 0.5/E$ ，然后根据初始坐标与电场方向是反方向得到x和y，输入数据里面有一个垂直速度大小，是垂直于速度方向的，根据电场的方向确定 v_x, v_y ，然后要设置一个时间步长dt，初始化x_list和y_list，然后进入时间的while循环，每次增加一个时间步长，然后x, y根据 v_x, v_y 更新， v_x, v_y 根据电场大小更新，电场大小分为两个部分，一部分是库仑场一部分是恒定外场，恒定外场部分随时间变化，每一次要进行更新，将每次计算完的x 和y坐标添加到x和y的list中，计算到t 到 $4 * \pi / \omega$ 时停止。然后根据x_list和y_list画图即可。

```
1 def E(t):
2     omega = 0.057
3     E = -1.325 * omega * (
4         0.5 * math.cos(omega * t) + 0.5 * math.cos(omega * t) *
5             math.cos(omega * t / 4) - 1 / 8 * math.sin(
6                 omega * t) * math.sin(omega * t / 4))
7
8     return E
```

```

7

9 def E_ellipse(t):
10    omega = 0.057
11    E_x = -1/math.sqrt(1.25) * 1.325 * omega * (
12        0.5 * math.cos(omega * t) + 0.5 * math.cos(omega * t) *
13        math.cos(omega * t / 4) - 1 / 8 * math.sin(
14        omega * t) * math.sin(omega * t / 4))
15    E_y = -0.5 / math.sqrt(1.25) * 1.325 * omega * (
16        -0.5 * math.sin(omega * t) - 0.5 * math.sin(omega * t) *
17        math.cos(omega * t / 4) - 1 / 8 * math.sin(
18        omega * t) * math.cos(omega * t / 4))
19
20    return E_x,E_y

21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49 def ellipse(t_0, v_ver):
50     E_x,E_y = E_ellipse(t_0)

```

```

51 E_0 = math.sqrt(E_x**2+E_y**2)
52 omega = 0.057
53 # 得到初始时的电场
54 r_0 = abs(0.5 / E_0)
55 x = -r_0*E_x/E_0 # 初始位置
56 y = -r_0*E_y/E_0
57 v_y = v_ver * E_x/E_0
58 v_x = -v_ver * E_y/E_0
59 dt = 0.001
60 t = t_0
61 x_list = []
62 y_list = []
63 while t < 4 * math.pi / omega:
64     x = x + v_x * dt
65     y = y + v_y * dt
66     x_list.append(x)
67     y_list.append(y)
68     E_x, E_y = E_ellipse(t)
69     v_x = v_x + (-E_x - x / (x ** 2 + y ** 2 + 0.2 ** 2) ** (1.5))
70         * dt
71     v_y = v_y + (-E_y - y / (x ** 2 + y ** 2 + 0.2 ** 2) ** (1.5))
72         * dt
73     t = t + dt
74 r_final = math.sqrt(x ** 2 + y ** 2)
75 v_final = math.sqrt(v_x ** 2 + v_y ** 2)
76 print('椭圆偏振光作用下的末态', 'x=', x, 'y=', y, 'v_x=', v_x, 'v_y='
77     =', v_y)
78 plt.plot(x_list, y_list)
79 plt.title('ellipse')
80 #plt.legend()
81 plt.show()

82 t_1 = float(input('输入线偏振光初始时间: '))
83 v_1 = float(input('输入线偏振光垂直速度: '))
84 t_2 = float(input('输入椭圆偏振光初始时间: '))
85 v_2 = float(input('输入椭圆偏振光垂直速度: '))
86 line(t_1, v_1)
87 ellipse(t_2, v_2)

```

2.2.4 结果展示

我们可以选取两个特殊情况来做例子: $t=0$ 和 $v=0.3$, 得到线偏振结果:

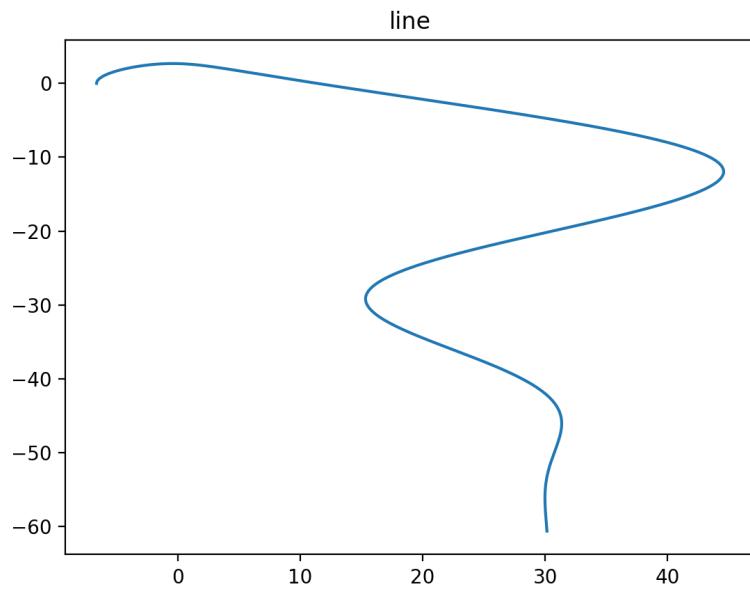


图 2.4: 线偏振作用下的粒子运动

得到椭圆偏振结果:

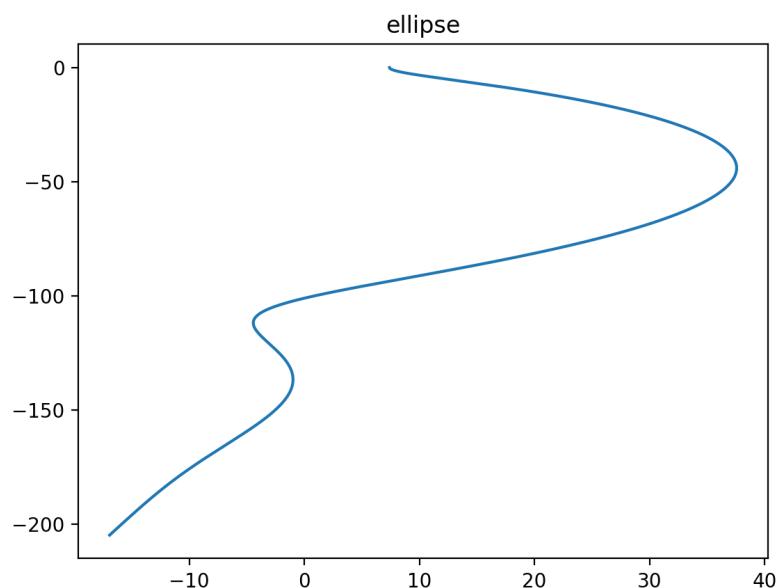


图 2.5: 椭圆偏振作用下的粒子运动

2.3 第三四问

2.3.1 题目

第三步，激光结束之后电子仅在库仑力的作用下运动，电子的运动可以解析得到，我们利用下面式子求得电子在无穷远处的渐进动量。从三个守恒量：能量，角动量，洛格楞次矢量

$$\begin{aligned}\frac{p_\infty^2}{2} &= \frac{p_f^2}{2} - \frac{1}{r_f}, \\ \vec{L} &= \vec{r}_f \times \vec{p}_f, \\ \vec{a} &= \vec{p}_f \times \vec{L} - \frac{\vec{r}_f}{r_f},\end{aligned}$$

可以得到无穷远处的动量：

$$\vec{p}_\infty = p_\infty \frac{\vec{L} \times \vec{a}}{1 + p_\infty^2 L^2}. \quad (2)$$

这里我们仅考虑总能量大于0的电子，物理上能量小于零的电子未被电离，而是被激发到了高里德堡态。注意，利用上式计算库伦势无需做软化，直接有 $r = \sqrt{x^2 + y^2}$ （由于激光结束后一般电子距离坐标原点较远，软化项可以忽略）。

最后一步，统计不同渐进动量的电子数目。我们统一记录 $-1.5 \leq p_x \leq 1.5, -1.5 \leq p_y \leq 1.5$ 范围内的电子，格点间隔 $\Delta p_x = \Delta p_y = 0.02$ 。我们把渐进动量落在 $(p_x \pm \Delta p_x/2, p_y \pm \Delta p_y/2)$ 区域内的电子都记录在动量格点 (p_x, p_y) 上，最后得到电离电子的末态动量分布。

本题要求给出的最终结果是分别在线偏光和椭圆偏振光作用下的电离电子末态动量分布，以及两个方向的线动量分布（例如，求 p_x 的分布，将相同 p_x 的所有不同 p_y 格点的电子数目加起来，得到动量为 p_x 的电子数目）。最后从线动量分布中给出峰值位置的动量大致大小。

2.3.2 思路

这一部分依赖于前两步的结论，首先我们需要通过第一步来生成电子样品，用第二步来计算电子末态位置和速度，然后在这一步中添加一个末状态计算函数，如果能电离的话，那么根据第二步的结果计算电离后的 p_x 和 p_y 。遍历整个有激光的时间最终得到这样一个分布即可，选择一个时间步长，每个时间选取一定次数的随机个数，因为第一问中的概率已经归一化，所以它在不同的时间是通用的。

2.3.3 程序实现

首先定义两个函数： $p_{infinite}$ 和 cac，前者用来计算到无穷远处的动量大小和方向，用到的公式就是：

$$\begin{aligned}\frac{p_{infinite}^2}{2} &= \frac{p_f^2}{2} - \frac{1}{r_f} \\ \vec{L} &= \vec{r}_f \times \vec{p}_f \\ \vec{a} &= \vec{p}_f \times \vec{L} - \frac{\vec{r}_f}{r_f} \\ \vec{p}_{infinite} &= p_{infinite} \frac{\vec{L} \times \vec{a}}{1 + p_{infinite}^2 L^2}\end{aligned}$$

将得到的动量填入到一个矩阵中，这个矩阵是 150×150 的，动量每隔 0.2 位一格，cac 就是这个计算过程，最后会输出一个矩阵。注意这里为了防止除数为 0，在 E 过小时就当做无法隧穿，根据第一步中所做的实验，将这个阈值设置为 0.3。

```

1 import random
2 import math
3 import matplotlib.pyplot as plt
4 import matplotlib as mpl
5 mpl.rcParams['font.sans-serif'] = ['KaiTi']
6 mpl.rcParams['font.serif'] = ['KaiTi']
7 mpl.rcParams['axes.unicode_minus'] = False # 解决保存图像是负号'-'显示
8         为方块的问题，或者转换负号为字符串
9
10 def E(t): # 线偏振电场函数
11     omega = 0.057
12     E = -1.325 * omega * (
13         0.5 * math.cos(omega * t) + 0.5 * math.cos(omega * t) *
14             math.cos(omega * t / 4) - 1 / 8 * math.sin(
15                 omega * t) * math.sin(omega * t / 4))
16
17
18 def E_ellipse(t): # 椭圆偏振电场函数
19     omega = 0.057
20     E_x = -1/math.sqrt(1.25) * 1.325 * omega * (
21         0.5 * math.cos(omega * t) + 0.5 * math.cos(omega * t) *
22             math.cos(omega * t / 4) - 1 / 8 * math.sin(
23                 omega * t) * math.sin(omega * t / 4))
24     E_y = -0.5 / math.sqrt(1.25) * 1.325 * omega * (
25         -0.5 * math.sin(omega * t) - 0.5 * math.sin(omega * t) *
26             math.cos(omega * t / 4) - 1 / 8 * math.cos(
27                 omega * t) * math.sin(omega * t / 4))
28
29
30 def species(E): # 对于固定的E选取一些v的样本
31     max = 1 / 0.076 ** 2 * math.exp(-(2 / 3) / 0.076) # max是全局的
32     v_list = [] # 初始化列表
33     for iter_a in range(1000):
34         kesai = random.random() # 随机生成的参考向量
35         v = random.uniform(-1, 1) # 随机选取v，在-1到1之间
36         W = 1 / E ** 2 * math.exp(-(2 / 3 + v ** 2) / E) # 隧穿几率
37         if W / max > kesai: # 判定是否选取
38             v_list.append(v)
39
40 return v_list # 返回一系列v的样本

```

```

40
41     def line(t_0, v_y): # 对于线偏振, 给定t和v计算末状态
42         v_x = 0 # x方向速度为0
43         E_0 = E(t_0)
44         omega = 0.057
45         # 得到初始时的电场
46         r_0 = (0.5/E_0)
47         x = -r_0
48         y = 0
49         dt = 1
50         t = t_0
51         x_list = []
52         y_list = []
53         while t < 4*math.pi/omega:
54             x = x + v_x * dt
55             y = y + v_y * dt
56             x_list.append(x)
57             y_list.append(y)
58             v_x = v_x + (-E(t) - x/(x**2+y**2+0.2**2)**(1.5))*dt
59             v_y = v_y - y/(x**2+y**2+0.2**2)**(1.5)*dt
60             t = t + dt
61         return [x, y, v_x, v_y]
62
63
64     def ellipse(t_0, v_ver): # 对于椭圆偏振, 给定t和v计算末状态
65         E_x,E_y = E_ellipse(t_0)
66         E_0 = math.sqrt(E_x**2+E_y**2)
67         omega = 0.057
68         # 得到初始时的电场
69         r_0 = (0.5 / E_0)
70         x = -r_0*E_x/E_0 # 初始位置
71         y = -r_0*E_y/E_0
72         v_y = v_ver * E_x / E_0
73         v_x = -v_ver * E_y / E_0
74         dt = 1
75         t = t_0
76         x_list = []
77         y_list = []
78         while t < 4 * math.pi / omega:
79             x = x + v_x * dt
80             y = y + v_y * dt
81             x_list.append(x)
82             y_list.append(y)
83             E_x, E_y = E_ellipse(t)
84             v_x = v_x + (-E_x - x / (x ** 2 + y ** 2 + 0.2 ** 2) ** (1.5))
85                         * dt
86             v_y = v_y + (-E_y - y / (x ** 2 + y ** 2 + 0.2 ** 2) ** (1.5))

```

```

    * dt
86   t = t + dt
87   return [x, y, v_x, v_y]
88

90 def p_infinite(x, y, v_x, v_y): # 对于给定末状态, 计算无穷远处动量
91     p_inf = math.sqrt(v_x**2 + v_y**2 - 2/math.sqrt(x**2+y**2))
92     L = x * v_y - y * v_x
93     a_x = L * v_y - x / math.sqrt(x**2+y**2)
94     a_y = -L * v_x - y / math.sqrt(x**2+y**2)
95     p_infx = p_inf/(1+p_inf**2*L**2)*(-p_inf*L*a_y-a_x)
96     p_infy = p_inf/(1+p_inf**2*L**2)*(p_inf*L*a_x-a_y)
97     return p_infx, p_infy
98

100 def cac(E_1, t, p_inf1_num, function): # function是用来计算到运动的函
101    数
102        if E_1 > 0.03: # E很小时不会发生隧穿
103            v_ver1 = species(E_1) # 生成样品
104            for i in range(len(v_ver1)):
105                result_1 = function(t, v_ver1[i])
106                if result_1[2]**2 + result_1[3]**2 - 2 / math.sqrt(
107                    result_1[0]**2 + result_1[1]**2) > 0:
108                    p_infx1, p_infy1 = p_infinite(result_1[0], result_1
109                        [1], result_1[2], result_1[3])
110                    if abs(p_infx1) < 1.5 and abs(p_infy1) < 1.5:
111                        p_inf1_num[round((p_infy1 + 1.49) / 0.02)][round((
112                            p_infx1 + 1.49) / 0.02)] += 1
113
114    return p_inf1_num

```

下面是主函数部分，即把各个部分串起来，完成一些初始化工作即可。resolution 是时间分辨率，每个时刻进行 1000 次随机，p_inf1_num 和 p_inf2_num 是计数的矩阵，p_x_list 和 p_y_list 是画图用到的横纵坐标。

```

1 omega = 0.057
2 resolution = 10000
3 t_0 = -4*math.pi/omega
4 p_inf1_num = [[0 for i in range(150)] for j in range(150)]
5 p_inf2_num = [[0 for i in range(150)] for j in range(150)]
6 for n in range(resolution): # 对于时间t做循环, 计算末状态
7     t = t_0 + 8*math.pi/omega * n / resolution
8     print('已完成计算: ', n/resolution*100, '%')
9     E_1 = E(t)
10    E_2x, E_2y = E_ellipse(t)
11    E_2 = math.sqrt(E_2x**2+E_2y**2) # 计算E
12    p_inf1_num = cac(E_1, t, p_inf1_num, line)
13    p_inf2_num = cac(E_2, t, p_inf2_num, ellipse)

```

```

15 p_x_list = []
16 p_y_list = []
17 for t in range(150): # 初始化x和y坐标
18     p_x_list.append(-1.49 + t * 0.02)
19     p_y_list.append(-1.49 + t * 0.02)

21 p_inf1_num_x = [0] * len(p_x_list) # 分到得到x和y分布的频数
22 p_inf1_num_y = [0] * len(p_x_list)
23 p_inf2_num_x = [0] * len(p_x_list)
24 p_inf2_num_y = [0] * len(p_x_list)
25 for i in range(len(p_x_list)):
26     for j in range(len(p_y_list)):
27         p_inf1_num_x[j] += p_inf1_num[i][j]
28         p_inf1_num_y[i] += p_inf1_num[i][j]
29         p_inf2_num_x[j] += p_inf2_num[i][j]
30         p_inf2_num_y[i] += p_inf2_num[i][j]
31 line_x_max1 = p_inf1_num_x.index(max(p_inf1_num_x[0:int(len(
32             p_inf1_num_x)/2)]))
33 line_x_max2 = p_inf1_num_x.index(max(p_inf1_num_x[int(len(p_inf1_num_x
34             )/2):len(p_inf1_num_x)]))
35 line_y_max = p_inf1_num_y.index(max(p_inf1_num_y))
36 ellipse_x_max = p_inf2_num_x.index(max(p_inf2_num_x))
37 ellipse_y_max1 = p_inf2_num_y.index(max(p_inf2_num_y[0:int(len(
38             p_inf2_num_y)/2)])
39 ellipse_y_max2 = p_inf2_num_y.index(max(p_inf2_num_y[int(len(
40             p_inf2_num_y)/2):len(p_inf2_num_y)]))
41 print('线偏振下x方向动量峰值为:', 'p_x1=', line_x_max1*0.02-1.49, 'p_x2=
42     ', line_x_max2*0.02-1.49)
43 print('线偏振下y方向动量峰值为:', 'p_y=' ,line_y_max*0.02-1.49)
44 print('椭圆偏振下x方向动量峰值为:', 'p_x=' ,ellipse_x_max*0.02-1.49)
45 print('椭圆偏振下y方向动量峰值为:', 'p_y1=' ,ellipse_y_max1*0.02-1.49, '
46     'p_y2=' ,ellipse_y_max2*0.02-1.49)
47
48 fig1 = plt.figure(1)
49 surf1 = plt.contourf(p_x_list, p_y_list, p_inf1_num)
50 fig1.colorbar(surf1)
51 plt.title('线偏振光下动量分布')
52 plt.xlabel('px')
53 plt.ylabel('py')
54 plt.show()

55
56 fig2 = plt.figure(2)
57 surf2 = plt.contourf(p_x_list, p_y_list, p_inf2_num)
58 fig2.colorbar(surf2)
59 plt.title('椭圆偏振光下动量分布')
60 plt.xlabel('px')

```

```

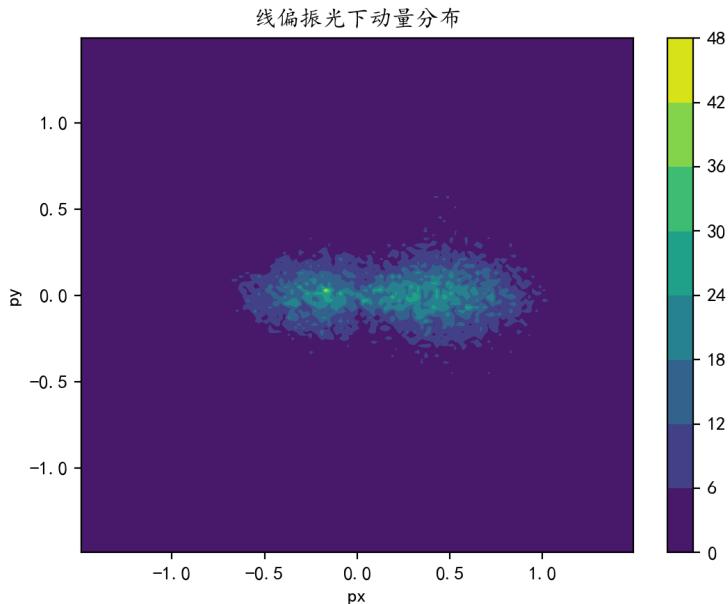
55 plt.ylabel('py')
56 plt.show()
57
58 fig3 = plt.figure(3)
59 plt.plot(p_x_list, p_inf1_num_x, label = 'px')
60 plt.plot(p_y_list, p_inf1_num_y, label = 'py')
61 plt.xlabel('p')
62 plt.ylabel('frequency')
63 plt.title('line_p')
64 plt.legend()
65 plt.show()

66
67 fig4 = plt.figure(4)
68 plt.plot(p_x_list, p_inf2_num_x, label = 'px')
69 plt.plot(p_y_list, p_inf2_num_y, label = 'py')
70 plt.xlabel('p')
71 plt.ylabel('frequency')
72 plt.title('ellipse_px')
73 plt.legend()
74 plt.show()

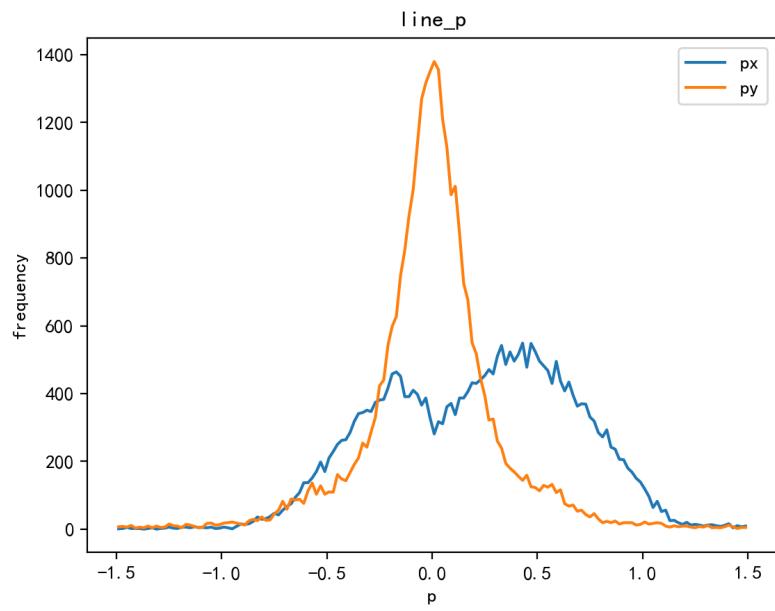
```

2.3.4 结果展示

线偏振下最后的动量分布如图所示：

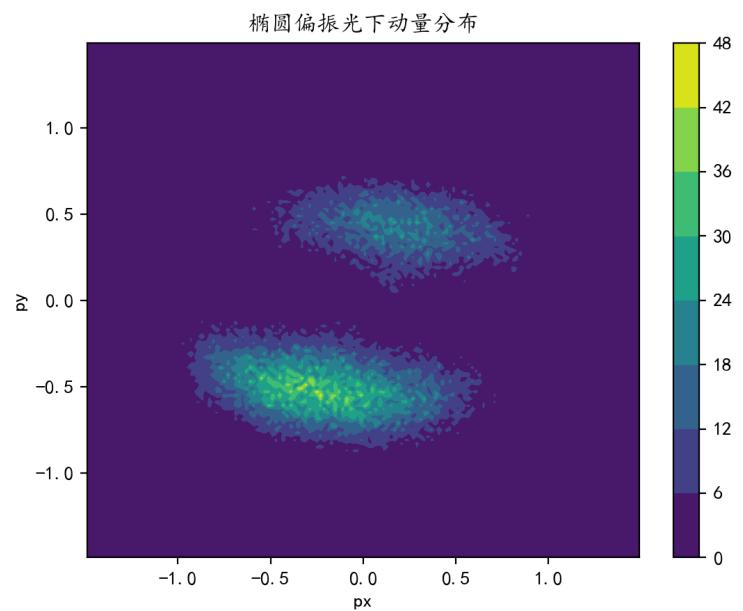


线偏振下，在 x 方向和 y 方向的动量分布

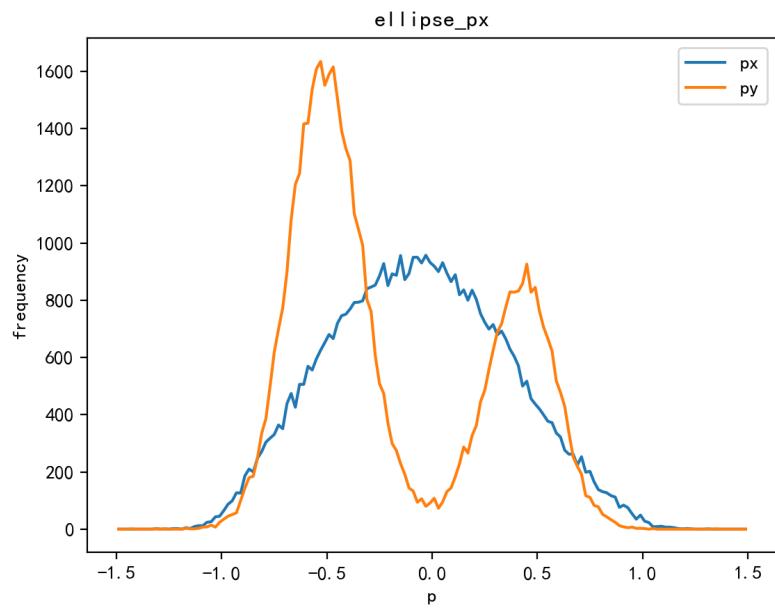


线偏振下 x 方向动量峰值为: $p_x1 = -0.1899999999999995$ $p_x2 = 0.4299999999999994$
 线偏振下 y 方向动量峰值为: $p_y = 0.010000000000000009$ (这些量可能没有过多的参考价值, 可随机取点有很大关系, 每次运行程序都不一样)

椭圆偏振下最后的动量分布如图所示:



椭圆偏振下, 在 x 方向和 y 方向的动量分布



椭圆偏振下 x 方向动量峰值为: $p_x = -0.030000000000000027$ 椭圆偏振下 y 方向动量峰值为: $p_y1 = -0.53$ $p_y2 = 0.3900000000000001$

Chapter 3

第三题

3.1 第一问

3.1.1 题目

1. 解析或数值证明对于没有高次项的情形，系统具有守恒量

$$E_k = \frac{1}{2} \dot{Q}_k^2 + \frac{1}{2} \omega_k^2 Q_k^2, \quad (3)$$

其中有

$$Q_k = \sqrt{\frac{2}{n}} \sum_j \sin \frac{\pi k j}{n+1} q_j, \quad \omega_k = 2 \sin \frac{\pi k}{2(n+1)}. \quad (4)$$

我们通常将不同的 k 称为不同的声子模式。

3.1.2 求解

下面给出数理方程的解析求解来证明能量守恒，忽略高阶项后，根据哈密顿方程，它满足的动力学关系为：

$$\ddot{q}_j = -2(q_j - q_{j+1} - q_{j-1})$$

$$\therefore q_j = \sum_k A_k(t) \cdot \sin k j$$

代入方程可得：

$$\sum_k \ddot{A}_k(t) \sin k j = -(2 \sum_k A_k(t) \sin k j - \sum_k A_k(t) \sin k(j+1) - \sum_k A_k(t) \sin k(j-1))$$

从而得到： $\ddot{A}_k(t) + \omega^2 A_k(t) = 0$ ，其中 $\omega(k) = 2 \sin \frac{k}{2}$ ，并由周期性条件 $q_0 = q_{n+1} = 0$ ，可知 $k \times (n+1) = t\pi$ ，故：

$$\omega(t) = 2 \sin \frac{t\pi}{2(n+1)}$$

$$\omega(k) = 2 \sin \frac{k\pi}{2(n+1)}$$

对 $\ddot{A}_k(t) + \omega^2 A_k(t) = 0$ 积分可以得到 $\frac{1}{2} \dot{A}_k^2 + \frac{1}{2} \omega_k^2 A_k^2 = E_k$ 为常数利用傅里叶变换可以得到：

$$A_k = Q_k = \sqrt{\frac{2}{n}} \sum_{j=1}^n q_j \cdot \sin \frac{k\pi j}{n+1}$$

故有：

$$\frac{1}{2} \dot{Q}_k^2 + \frac{1}{2} \omega_k^2 Q_k^2 = E_k$$

为常数

3.2 第二问

3.2.1 题目

2. Fermi 相信，引入非谐项后，系统的能量可以从一个声子模式扩散到全体声子模式上去，从而实现对热平衡过程的模拟。在 FPU 最初的报告中，他们选取 α -FPU 模型，令 $\alpha = 0.25, n = 32, Q_1(0) = 4$ 而其他 $Q_k(0)$ 以及全部 $\dot{Q}_k(0)$ 都为零，计算了 $t \in [0, 160 \times 2\pi/\omega_1]$ 的时间内系统的运动²。试重复这个计算，在同一张图上画出 E_1, E_2, E_3, E_4 随时间的变化，时间轴的单位取 $2\pi/\omega_1$ 。说明这段时间内有一个时刻几乎全部能量都回到了 $k = 1$ 的模式上，并给出具体时刻和能量比例。

3.2.2 思路

这个题目实质上是解一个哈密顿正则方程，哈密顿量采用 $\alpha - FPU$ 的量，然后根据题目所给的初始条件确定各 p 和 q ，然后根据哈密顿方程：

$$\begin{aligned}\frac{\partial q}{\partial t} &= \frac{\partial H}{\partial p} \\ \frac{\partial p}{\partial t} &= -\frac{\partial H}{\partial q}\end{aligned}$$

求解这个方程即可，这里我采用了保辛的蛙跳显式算法，但是这个算法有两个问题：

- (1) 无法自启动
- (2) 得到的 p 和 q 不是同时刻的

针对 (1) 我在头半个步长设置了一个初始化算法，在这半个步长中加密取点，尽量减少误差，得到一个初始化的 q ，然后再进行下面的运算。

针对 (2) 我在后面计算时采用了线性插值，因为整个算法稳定保辛，所以这个插值的精度不会随时间变大，这在长时间的解方程中是必须的。

3.2.3 程序实现

首先根据题目定义两个函数求 Q 以及它的导数，求解初值时用到了离散傅里叶变换，因为这在后面求解初始条件以及各声子能量时要用到。由于题目中给的初始条件是 Q 的，但是计算时要用到 p 和 q，因此还要在程序开始加上 p 和 q 的初始化。同时定义了计算 E 的函数。具体如下：

```
1 def Q(k, q, n):
2     res = 0
3     for i in range(n+2):
4         res += math.sin(math.pi*k*i/(n+1))*q[i]
5     res = res * math.sqrt(2 / n)
6     return res
7
8
9 def Q_dot(p, k, n):
10    res = 0
11    for i in range(n + 2):
12        res += math.sin(math.pi * k * i / (n + 1)) * p[i]
13    res = res * math.sqrt(2 / n)
14    return res
15
16
17 def E(n, k, q, p): # 求能量
18     E_1 = 0.5 * Q_dot(p, k, n) ** 2
19     E_2 = 0.5 * Q(k, q, n) ** 2 * (2 * math.sin(math.pi * k / (2*(n+1))
20         )) ** 2
21     return E_1 + E_2
```

下面初始化 p 和 q，求解初始条件，根据傅里叶变换：

$$q_j = \sqrt{\frac{2}{n}} \sum_k Q_k \sin\left(\frac{k j \pi}{n+1}\right) \times \frac{n}{n+1}$$

初始化如下：

```
n = 32
1 p = [0] * (n + 2)
2 q = []
3 for t in range(n+2):
4     q.append(4*math.sin(t*math.pi/(n+1))/math.sqrt(n/2) * n/(n+1))
```

下面是主要的函数部分，把 p,q,E_list 各量初始化后，将 t=0 的能量加上去，设置时间分辨率 resolution，dt 是步长，ddt 是第一次进行初始化的时间步长，初始化后得到 q[0.5] 然后就可以与 p[0] 进行交错蛙跳求解，并且每到 $2\pi/\omega$ 整数倍时计算能量并添加到 list 后面，最后进行画图即可，同时求出恢复的时间。

```

1 q_real = q # 真实的q
2 p_origin = p # 记录初始p
3 E_1_list = []
4 E_2_list = []
5 E_3_list = []
6 E_4_list = []
7 E_1_list.append(E(n, 1, q_real, p))
8 E_2_list.append(E(n, 2, q_real, p))
9 E_3_list.append(E(n, 3, q_real, p))
10 E_4_list.append(E(n, 4, q_real, p)) # 将t=0的能量添加
11 t_list = [i for i in range(161)]
12 resolution = 160000 # 分辨率要是160的倍数
13 dt = 2*math.pi/(2*math.sin(math.pi/2/(n+1)))*160/resolution
14 ddt = dt / 100 # 第一次计算的时间分辨率
15 for i in range(50): # 初始化计算
16     q_old = q
17     for i in range(n):
18         q[i + 1] += ddt * p[i + 1]
19     for i in range(n): # 计算p和q的变化，假设得到的q是k+0.5的
20         p[i + 1] += -ddt * (q_old[i + 1] - q_old[i + 2] + 0.25 * (
21             q_old[i + 1] - q_old[i + 2]) ** 2
22             - q_old[i] + q_old[i + 1] - 0.25 * (q_old[
23                 i] - q_old[i + 1])) ** 2)
24     for j in range(resolution):
25         print('完成', j/resolution*100, "%")
26     q_old = q
27     p = p_origin
28     for i in range(n): # 计算p和q的变化，得到的q是j+0.5的
29         p[i + 1] += -dt * (q_old[i + 1] - q_old[i + 2] + 0.25 * (
30             q_old[i + 1] - q_old[i + 2]) ** 2
31             - q_old[i] + q_old[i + 1] - 0.25 * (q_old[
32                 i] - q_old[i + 1])) ** 2)
33     for i in range(n):
34         q[i + 1] += dt * p[i + 1]
35     for u in range(len(q)): # 插值计算q
36         q_real[u] = (q[u]+q_old[u])/2
37     if (j+1)%(resolution/160) == 0: # 在一定的位置再计算E
38         E_1_list.append(E(n, 1, q_real, p))
39         E_2_list.append(E(n, 2, q_real, p))
40         E_3_list.append(E(n, 3, q_real, p))
41         E_4_list.append(E(n, 4, q_real, p))
42 E_1_recover = max(E_1_list[80:160])
43 t = E_1_list.index(E_1_recover)
44 print('E1恢复到极大值时的时间为: ', t, '能量为: ', E_1_recover, '占总
45 能量比例为E1:E2:E3:E4=', E_1_list[t]/E_1_list[0], ':', E_2_list[t]/
46 E_1_list[0], ':',
47 E_3_list[t]/E_1_list[0], ':', E_4_list[t]/E_1_list[0])

```

```

43 plt.figure(1)
44 plt.plot(t_list,E_1_list,label='E1')
45 plt.plot(t_list,E_2_list,label='E2')
46 plt.plot(t_list,E_3_list,label='E3')
47 plt.plot(t_list,E_4_list,label='E4')
48 plt.xlabel('t')
49 plt.ylabel('E')
50 plt.title('energy change over time')
51 plt.legend()
52 plt.show()

```

3.2.4 结果展示

E1 恢复到极大值时的时间为: 156

能量为: 0.07086502897185665 占总能量比例为 E1:E2:E3:E4= 0.9781335196289244
: 0.0005268547436596408 : 0.013459630566928954 : 0.0022667313519497364

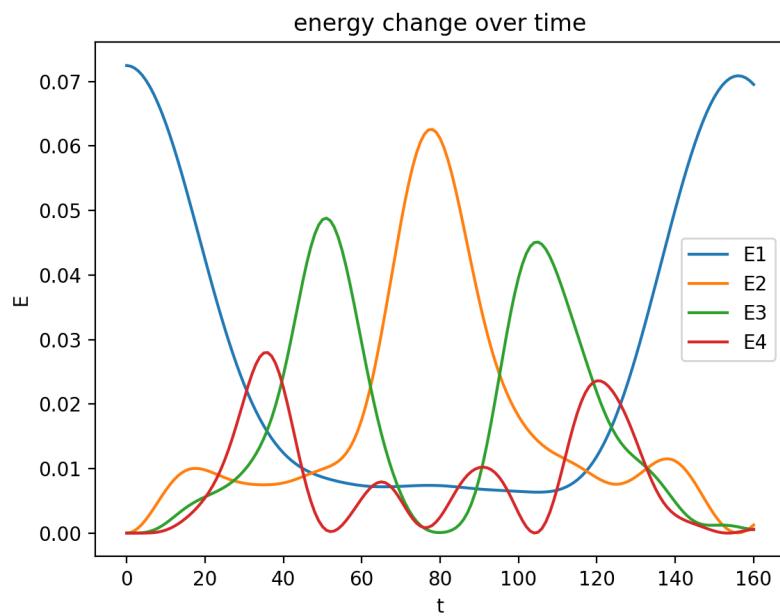


图 3.1: 能量随时间变化图

3.3 第三问

3.3.1 题目

给出上个结果的可能解释

3.3.2 求解

假如第一问的结果不忽略高次项的话

$$\ddot{q}_j = -2(q_j - q_{j+1} - q_{j-1}) - \beta(q_j - q_{j+1})^2 + \beta(q_j - q_{j-1})^2$$

代入 $q_j = \sum_k A_k(t) \cdot \sin k j$

$$\begin{aligned} \sum_k \ddot{A}_k \sin k j &= -\left(2 \sum_k A_k \sin k j - \sum_k A_k(t) \sin k(j+1) - \sum_k A_k(t) \sin k(j-1)\right) \\ &\quad - \beta\left(2 \sum_k A_k \sin k j - \sum_k A_k(t) \sin k(j+1) - \sum_k A_k(t) \sin k(j-1)\right) \\ &\quad \left(\sum_k A_k \sin k(j-1) - \sum_k A_k \sin k(j+1)\right) \end{aligned}$$

作离散化内积：

$$\frac{d^2 A_k}{dt^2} = -\omega^2 A_k - \beta \omega^2 A_k \sum_m \sum_k A_m \cos m j \sin m \sin k j$$

这时候 $\frac{1}{2} \dot{A}_k^2 + \frac{1}{2} \omega^2 A_k^2$ 不再是守恒项

$$\frac{d^2 A_k}{dt^2} = -\alpha A_k - \gamma A_k^2$$

这个方程的解 $A_k = a + b(\cos(\omega t) + c \cos(2\omega t))$, $E_k = \frac{1}{2} \dot{A}_k^2 + \frac{1}{2} \omega^2 A_k^2$ 会出现多个周期，且周期与 k 相关。从而能量会发生周期性变化，形成如图所示的结果。

3.4 第四问

3.4.1 题目

4. 1961 年，Tuck 等人将计算的时间长度进一步向前推进。采用前述参数，给出 $t \in [0, 4000 \times 2\pi/\omega_1]$ 的时间中 E_1 的变化图像。说明这段时间内存在一个更大的回归周期。给出具体的回归时刻和能量比例。

3.4.2 求解

这一部分直接沿用第二题程序即可，只需要将时间设置为 4000，观察 E_1 随时间的变化，然后输出最大值的时间指标和能量情况即可。如图 3.4.2 所示

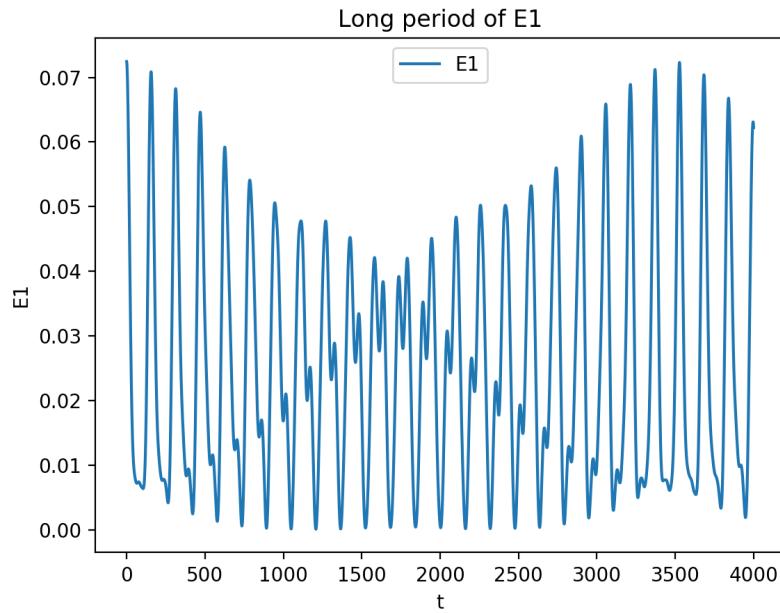


图 3.2: 能量 E_1 随时间变化图

E_1 恢复到极大值时的时间为: 3527

能量为: 0.07230423907524403 比例为: 0.9979986020868865

3.5 第五问

3.5.1 题目

5. 非线性系统的动力学行为往往强烈依赖于参数和初始条件的选取。取 $Q_1 = 20$, 其他参数不变。计算并从模式能量的角度描述系统的演化。在多长的时间后系统表现出混乱的特性? 讨论此时 $\langle E_k \rangle$ 与 k 的关系, 是否和能均分定理相符?

3.5.2 求解

取 $Q_1 = 20$, 其他参数不变, 计算模式能量在多长时间后表现出混乱的特性。这里即按照要求, 同样沿用前几问的程序, 考虑各声子能量随时间变化图, 以及 E_1, E_2, E_3, E_4 平均能量随时间变化。最后画出 E_k 随 k 变化曲线。

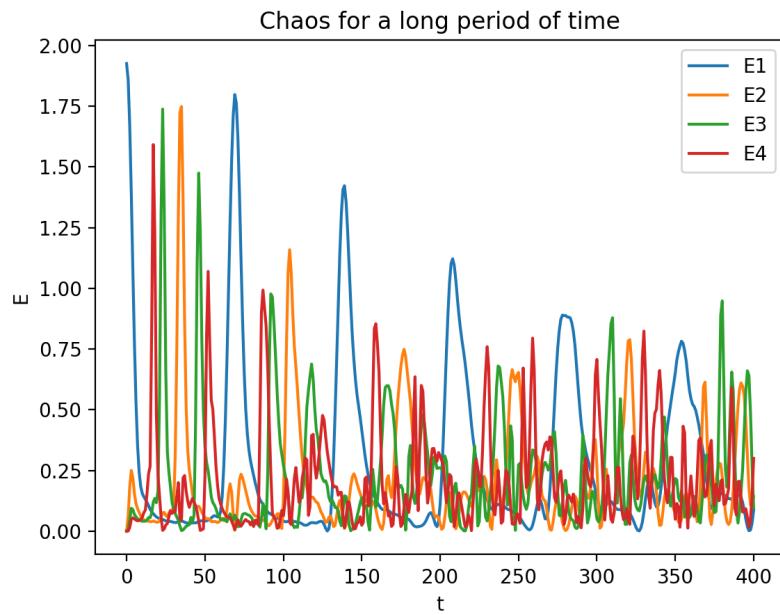


图 3.3: 各声子随时间变化图,t=0-400

从图中可以看到,一开始各声子的能量还比较有规律,但是后面逐渐趋向于混乱,在 t 等于 300 左右表现出混乱的特性。系统的演化是每个声子的能量从定值开始互相传递,最后达到热平衡。时间可以取到 4000 附近,能量已经处于较为混乱的状态,这里考虑前四个模态的能量。

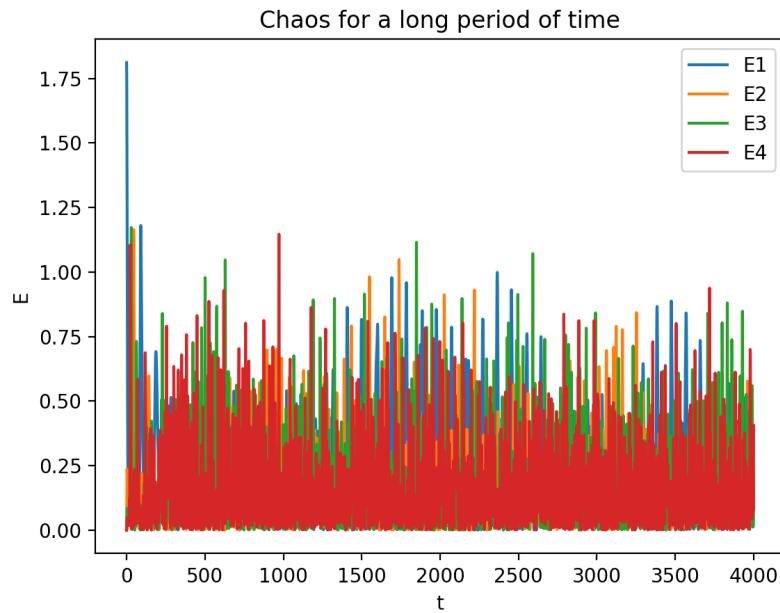


图 3.4: 各声子随时间变化图,t=0-4000

可以看到相当混乱，可以认为达到了热平衡，此时画出 1-4 个模态上能量随时间变化以及所有能态上能量在 k 维度的平均随 k 的变化：

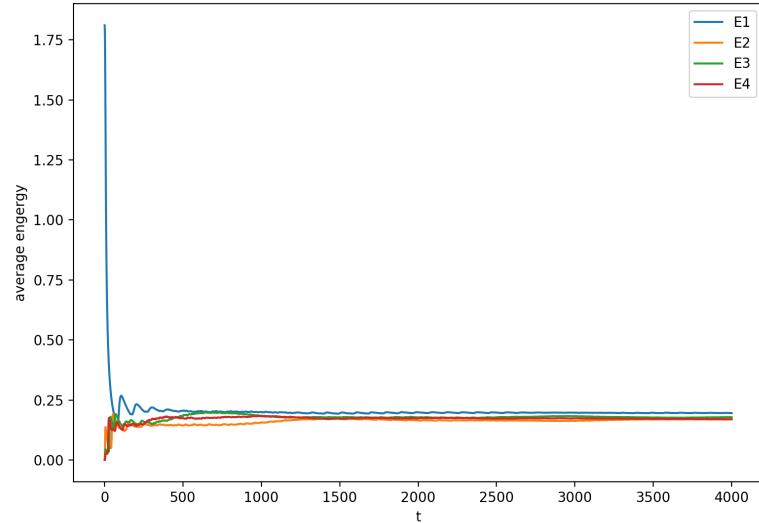


图 3.5: $k=1-4$, 各态声子能量平均值随时间变化图, $t=0-4000$

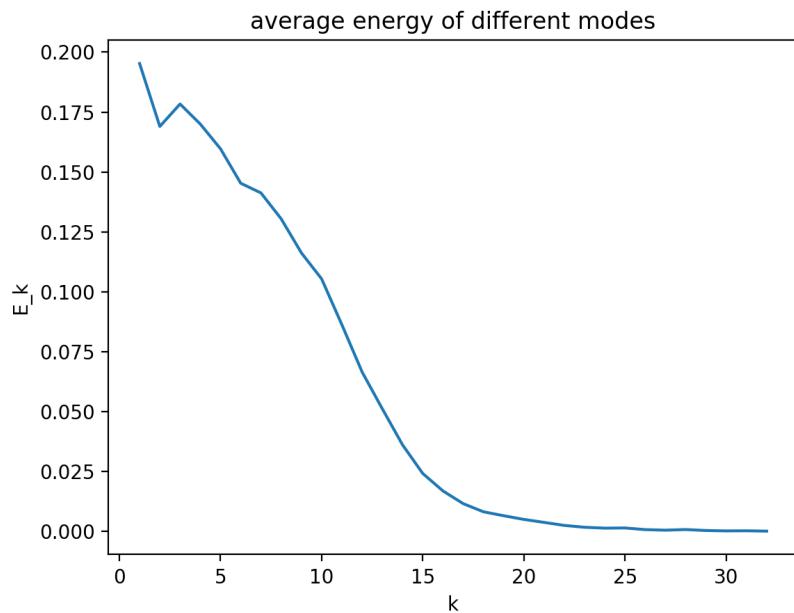


图 3.6: E_k 对 k 的平均随 k 的变化

我们发现，大约在 $t=1800$ 之后，各个模式平均能量基本不再变化，能量相近但并不相等。

3.6 第六问

3.6.1 题目

6. 我们将目光转向 β -FPU 模型。自行选择参数，验证 β -FPU 模型中也有与之前类似的现象。

3.6.2 求解

将于 α 模型完全类似的代码应用于此题中，这里就不再赘述了。此处 $\beta = 2, Q_1(0) = 4$ ，观察各个模式能量随时间的变化。

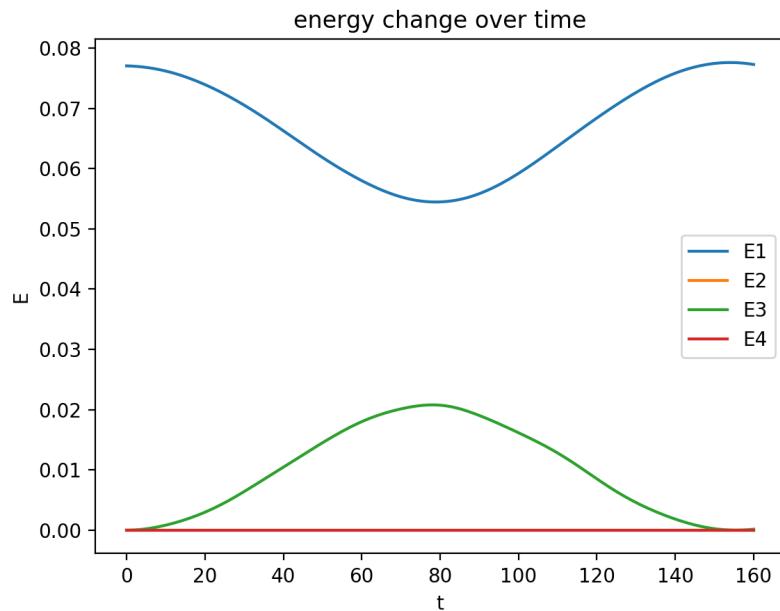


图 3.7: β 模型下各声子随时间变化图

可以看到 1 与 3 模式间也有能量传递，选取 $\beta = 0.6, Q_1(0) = 20$ ，观察各个模式能量随时间的变化。需要说明的是，这两种情况和图分别对应于 HW3-3(6) 和 (6.chaos) 的 python 文件。

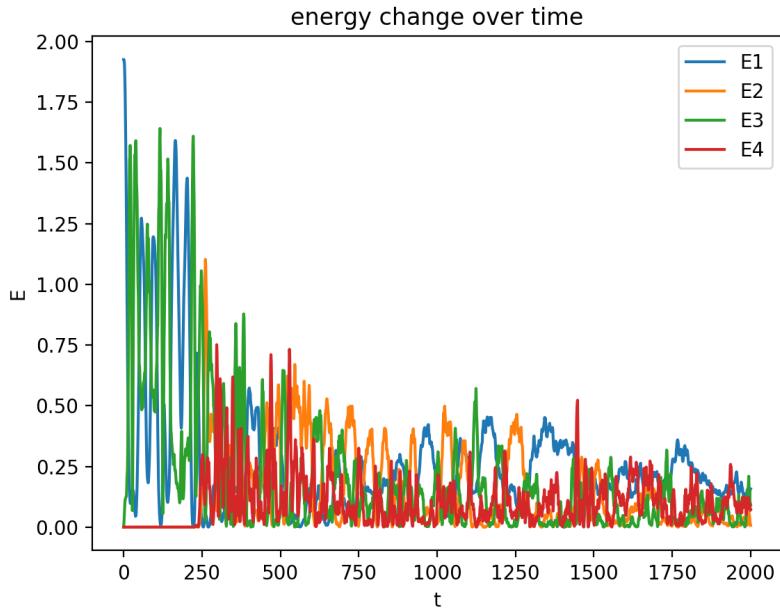


图 3.8: β 模型下各声子随时间变化图

可以看到也会出现混沌的现象。

3.7 第七问

3.7.1 题目

7. 论证对于 n 为偶数的 β -FPU 模型，若初始能量均分布在偶/奇次模式上，则随时间演化后仍然分布在偶/奇次模式上。取 $n = 16, \beta = 1$ ，初始条件为 $Q_{11} = 1$ 而其余为零，在对数标度上画出 9 至 13 次模式上能量的演化。解释你所观察到的现象。

3.7.2 求解

讨论初始能量在奇数态上，偶数态 Q_k 及 \dot{Q}_k 全部为 0：

$$q_{n+1-j} = \sum_k Q_k \cdot \sin \frac{k\pi(n+1-j)}{n+1} = \sum_k Q_k \sin(k\pi - \frac{j\pi}{n+1})$$

k 为奇数，有 $\sin(k\pi - \frac{j\pi}{n+1}) = \sin \frac{j\pi}{n+1}$ ， $q_{n+1-j} = \sum_k Q_k \sin \frac{j\pi}{n+1} = q_j$

所以在运动过程中一直有 $q_{n+1-j} = q_j$, $\dot{q}_{n+1-j} = \dot{q}_j$

对于偶数态，

$$Q_k = \sqrt{\frac{2}{n}} \sum_j \sin \frac{\pi k j}{n+1} q_j = \sqrt{2} n \times \frac{1}{2} \times \sum_j (\sin \frac{\pi k j}{n+1} q_j + \sin \frac{\pi k(n+1-j)}{n+1} q_{n+1-j})$$

k 为偶数, 有 $\sin(k\pi - \frac{j\pi}{n+1}) = -\sin \frac{j\pi}{n+1}$, $q_{n+1-j} = q_j$, $\sin \frac{\pi k j}{n+1} q_j + \sin \frac{\pi k(n+1-j)}{n+1} q_{n+1-j} = 0$ 故 $Q_k = 0, \dot{Q}_k = 0$, 偶数态上能量一直为 0, 对于初始能量在偶数态, 同理可证明能量不传向奇数态。

各个能量态的能量对数随时间变化如下图所示:

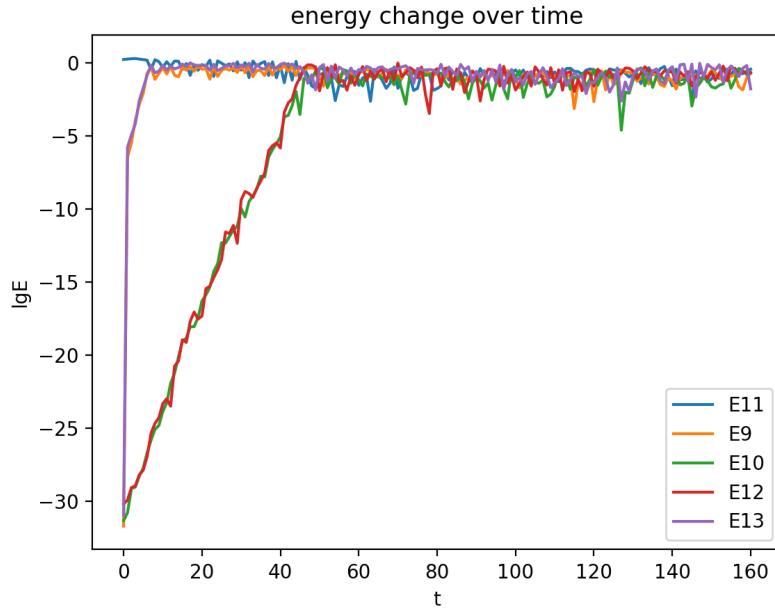


图 3.9: 各态能量随时间变化图

可以看到一开始奇数态的能量很快增长了上来, 偶数态的能量仍然比较低, 但是随着时间的演化, 误差慢慢累积, 偶数态的能量也在慢慢增长, 并且在一段时间以后与奇数态能量已经差不多了, 达到热平衡, 这一事实可能和理论不符合, 我认为这可能由于方程本身的不稳定造成的, 在机器精读下, 即使采用不同算法, 误差也会逐渐扩大, 最终达到混乱。

3.8 第八问

3.8.1 题目

8. β -FPU 模型可以看做是著名的 KdV 方程

$$\partial_t u + u \partial_x u + \delta^2 \partial_{xxx} u = 0, \quad (5)$$

的离散形式。从而，在恰当的参数下我们能在 β -FPU 模型中看到 KdV 方程特有的孤子解。取 $n = 128$, 分别对 $\beta = 0, 1$, 考虑如下形式的初始条件

$$\begin{aligned} q_i &= B \cos \frac{\pi k(i - n/2)}{n+1} / \cosh \left[\sqrt{\frac{3}{2}} B \omega_k (i - n/2) \right], \\ p_i &= \frac{B}{\cosh \left[\sqrt{\frac{3}{2}} B \omega_k (i - n/2) \right]} \left\{ \omega_k \left(1 + \frac{3}{16} \omega_k^2 B^2 \right) \sin \frac{\pi k(i - n/2)}{n+1} \right. \\ &\quad \left. + \sqrt{\frac{3}{2}} B \cos \frac{\pi k(i - n/2)}{n+1} \sin \frac{\pi k}{n+1} \tanh \left[\sqrt{\frac{3}{2}} B \omega_k (i - n/2) \right] \right\}, \end{aligned} \quad (6)$$

其中取 $B = 0.5, k = 11$, 计算系统的运动。用合适图表展示波包的运动、反射、扩散等过程并讨论之。

3.8.2 思路

根据 p, q 的运动方程求解 p, q 的运动，不需要计算能量，将所有振子坐标在一张图表现出来，画色度图，从而表现波包的扩散，反射和运动。

3.8.3 程序实现

程序里面不再包含原有的函数每只包括上几步中用到的主函数，初始化后进行运算画图即可，这里面 t_max 是时间最大值， t_inter 是绘图间隔， t_run 是运行的时间间隔， t_start 是开始画图的时间， t_stop 是结束画图时间，一个时间单位是 $2 \times \pi / (2 \times \sin(\pi/2/(n+1)))$ ，然后在不同时间上画图可以表现波包的扩散，反射和运动。程序如下：

```
1 def draw(beta, t_max, t_inter, t_run, t_start, t_stop):
2     n = 128
3     p = [0]
4     q = [0]
5     B = 0.5
6     k = 11
7     ome = 2*math.sin(math.pi*k/2/(n+1))
8     for i in range(1, n+1):
```

```

9    q.append(B * math.cos(math.pi * k * (i - n / 2) / (n + 1)) /
   10   math.cosh(math.sqrt(3 / 2) * B * ome * (i - n / 2)))
11   p.append(B / math.cosh(math.sqrt(3 / 2) * B * ome * (i - n /
   12   2)))
13   * (ome * (1 + 3 * (ome ** 2) * (B ** 2) / 16)
   14   * math.sin(math.pi * k * (i - n / 2) / (n + 1)) +
   15   math.sqrt(3 / 2)
   16   * B * math.cos(math.pi * k * (i - n / 2) / (n + 1))
   17   * math.sin(math.pi * k / (n + 1)) * math.tanh(math.
   18   sqrt(3 / 2) * B * ome * (i - n / 2)))
19   p.append(0)
20   q.append(0)
21   q_real = q # 真实的q
22   p_origin = p # 记录初始p
23   q_record = []
24   q_record.append(q)
25   t_list = [t_start+i*t_inter for i in range(int((t_stop-t_start)/
   26   t_inter)+1)]
27   x_list = [i for i in range(130)]
28   dt = 2*math.pi/(2*math.sin(math.pi/2/(n+1)))*t_run
29   ddt = dt / 1000 # 第一次计算的时间分辨率
30   for t in range(500): # 初始化计算
31       q_old = q
32       for i in range(n):
33           q[i + 1] += ddt * p[i + 1]
34       for i in range(n): # 计算p和q的变化,假设得到的q是k+0.5的
35           p[i + 1] += - ddt * (q_old[i + 1] - q_old[i + 2] + beta *
36           (q_old[i + 1] - q_old[i + 2])) ** 3
37           - q_old[i] + q_old[i + 1] - beta * (
38           q_old[i] - q_old[i + 1]) ** 3
39       p = p_origin
40       for j in range(int(t_max/t_run)):
41           print('完成', j / int(t_max/t_run) * 100, "%")
42       q_old = q
43       for i in range(n):
44           p[i + 1] += - dt * (q_old[i + 1] - q_old[i + 2] + beta * (
45           q_old[i + 1] - q_old[i + 2])) ** 3
46           - q_old[i] + q_old[i + 1] - beta * (
47           q_old[i] - q_old[i + 1]) ** 3
48       for i in range(n):
49           q[i + 1] += dt * p[i + 1]
50       for u in range(len(q)):
51           q_real[u] = (q[u] + q_old[u]) / 2
52       if j/int(t_max/t_run) > t_start/t_max and j/int(t_max/t_run) <
53           t_stop/t_max and (j + 1) % (int(t_inter/t_run)) == 0:
54           q_u = deepcopy(q_real)
55           q_record.append(q_u)

```

```

47 fig1 = plt.figure(1)
surf1 = plt.contourf(x_list, t_list, q_record)
fig1.colorbar(surf1)
49 if beta == 0:
    plt.title('beta=0')
51 else:
    plt.title('beta=1')
53 plt.xlabel('x')
55 plt.ylabel('t')
57 plt.show()

57 draw(0, 1, 0.005, 0.0001, 0, 1)
59 draw(0, 10, 0.005, 0.0001, 9, 10)
draw(1, 1, 0.005, 0.0001, 0, 1)
61 draw(1, 10, 0.005, 0.0001, 9, 10)

```

3.8.4 结果展示

以下各图的横坐标都是振子的编号，纵坐标为时间，时间的单位是 $\frac{2 \times \pi}{2 \times \sin(\frac{\pi}{2(n+1)})}$ ，
 t_{\max} 设置为 1，观察时间 $t=0-1$ 之间粒子的运动，颜色表示振子的振幅，画图可以看到波包的反射：

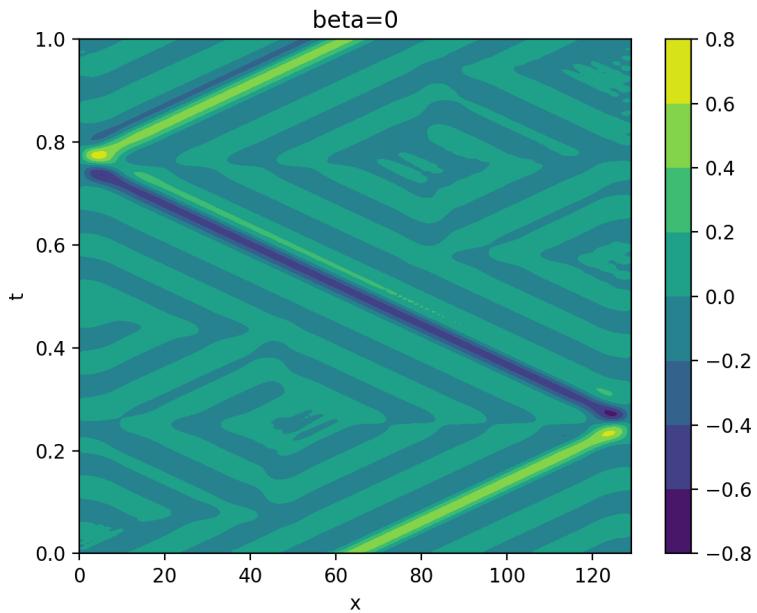


图 3.10: beta=0, t=0-1

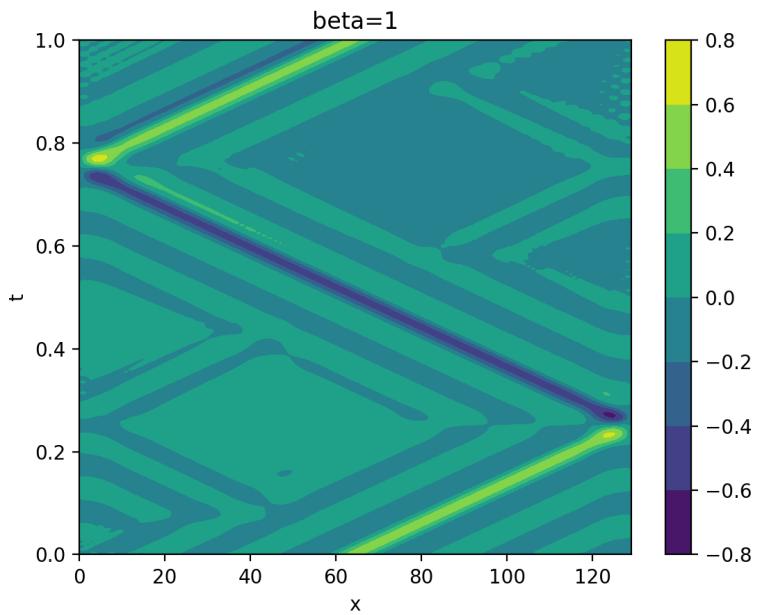


图 3.11: $\text{beta}=1$, $t=0-1$

同理, 取 t_{max} 为 10 时可以看观察 $t=9-10$ 的运动, 可以看到 $\beta = 1$ 的情况并没有发生扩散, 而 $\beta = 0$ 的情况已经发生扩散, 出现了多个峰值:

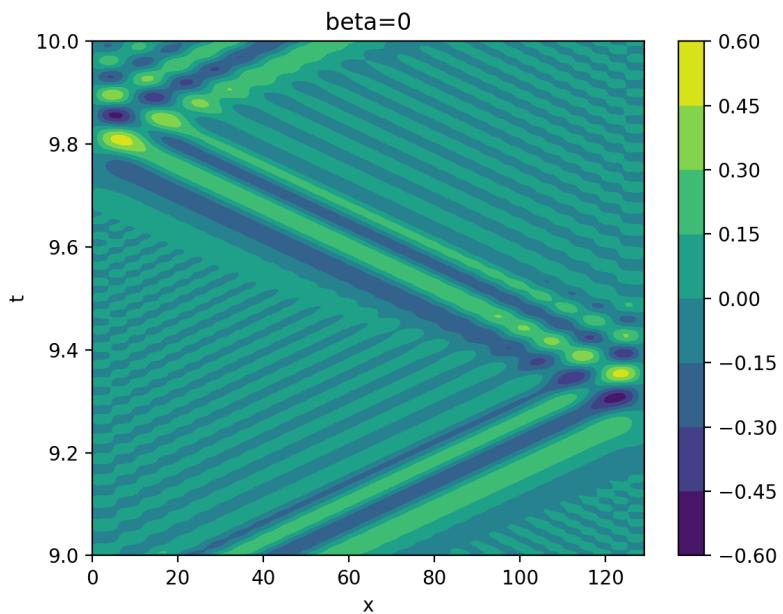


图 3.12: $\text{beta}=0$, $t=9-10$

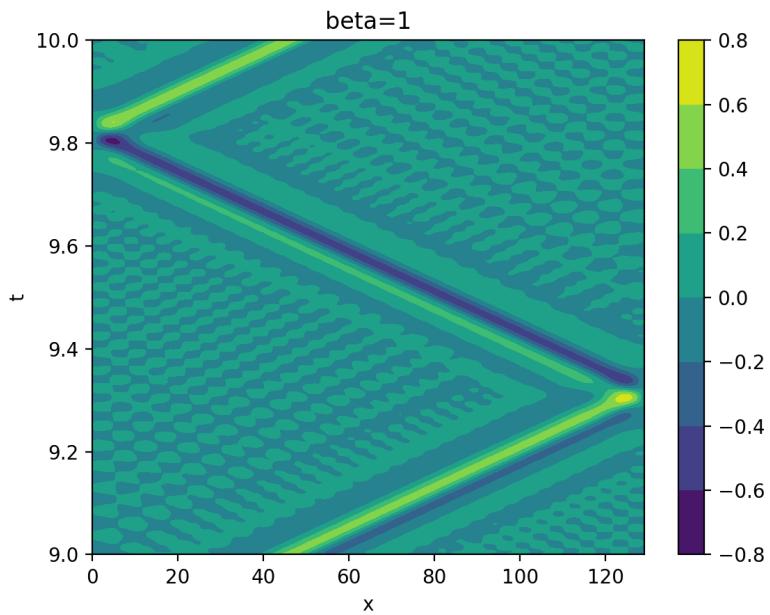


图 3.13: $\beta=1$, $t=9-10$

可以在前两幅图中看到波包的运动和反射, $\beta = 1$ 或 $\beta = 0$ 的区别不是很大, 都在初始条件下在左右两壁间传播, 在碰到壁时发生反射反向传播, 但是在后面两幅图中可以看到 $\beta = 1$ 或 $\beta = 0$ 有一定的区别, $\beta = 0$ 时波形已经不再稳定, 出现了多个峰值, 宽度也加宽了, 说明发生了扩散, 而 $\beta = 1$ 则没有什么变化, 仍然维持着原来的波形, 发生这种情况的原因是 $\beta - FPU$ 模型是 KdV 方程特有的孤子解, 初始条件在 $\beta = 1$ 时可以稳定存在, 但是在 $\beta = 0$ 时不是解, 所以会随时间变化, 也就是我们看到的波形扩散。