计算物理第一次大作业

王台宜

学号：1500011419

2019 年 3 月 29 日

目录

Chapter 1

第一题

1.1 题目

（一）【10分】编写两个程序，实现任意十进制数与其规范化的二进制双精度浮点数编码间的相互转换，并列举足够算例说明你程序的正确性。

1.2 分析与思路

在十进制转化成二进制的程序里，首先要判断数是否可以规范化用双精度浮点数表示。所以先验证数的绝对值在 1.79E-308 和 2.23E308 之间，浮点数分为三个部分：符号位，指数部分和小数部分。先根据第一位判定十进制数的正负，当道二进制的符号位，然后将十进制数分为小数部分和整数部分，然后分三种情况，分情况利用 while 循环求出指数位和每一个小数位，最后将三个部分串联起来即可。

在二进制转化为十进制的程序里，要尽可能地减少误差，不能使用每次取一位小数位的方法相加，这样会造成舍入误差，因为大数和小数相加时会有很大误差，所以我将指数位和小数位分开算，先算出幂次，在算出小数位代表的数，最后用两者相乘得到准确的值。

1.3 算法描述和实现

1.3.1 十进制转化成二进制

首先进行输入数据的输入与检查上下限，并进行 0 的检查，如果输入的是 0，直接输出 0 的浮点数表示结果。接着讲输入的数字判断正负，然后取绝对值，并且分为整数部分 integer 和小数部分 decimal：

```
# coding=UTF-8
# 此程序用来将十进制数转化为双精度浮点数
for k in range(1,100):
    num = input("Enter the input decimal number: ")
```

```
    num_decimal = float(num)
    if num_decimal == 0:  # 如果数为0，则直接输出
        print(64 * '0')
        exit()
    while abs(num_decimal) > 2.23E308 or abs(num_decimal) < 1.79E-308:
        # 检验是否在可表示的范围内
        print("您输入的数不在双精度浮点数表示范围内，请重新输入：\n")
        num_decimal = input("Enter your input: ")
    if num_decimal < 0:  # 看正负
        pm = str(1)
    else:
        pm = str(0)
    num_decimal = abs(num_decimal)  # 取绝对值
    bit = 52
    integer = int(num_decimal)
    decimal = num_decimal - integer
```

接着分为三种情况：第一种是整数部分大于 1，那么把整数部分用 bin 函数转化为二进制表示，除掉第一个 1，储存在 numbinary1 中，并且根据整数部分的二进制位数确定小数部分的二进制位数，两者之和为 52，用 bit 表示，将小数部分 result 用 while 循环不断乘 2，如果每次得到的结果大于 1，那么下一位取 1，否则取 0，然后取小数部分进入下一次循环，直到将 52 位小数部分用完，则退出 while 循环，将正负号表示位和指数部分和整数部分连接起来输出即可；第二种情况是整数部分正好等于 1，这样就不用再考虑整数部分，指数部分也是确定的'01111111111'，因为整数部分的 1 正好被削掉，于是小数部分与上一中方法相同，取 52 位即可；第三种情况：整数部分为 0 需要不断对小数部分进行乘 2 操作，知道 result 大于 1，之后同第一种情况，得到 52 位小数部分，最后把三个部分串联输出即可。代码实现如下：

```
    if integer > 1:  # 整数部分大于1
        num_binary_1 = str(bin(integer)[3:])  # 整数部分
        bit = bit - len(num_binary_1)
        num_binary_2 = ""  # 小数部分
        i = 0
        while i < bit:
            result = int(decimal * 2)
            decimal = decimal * 2 - result
            num_binary_2 += str(result)
            i += 1
        index = str(bin(1023 + len(num_binary_1))[2:])  # 指数部分
        x = len(index)
        num_binary = pm + index + num_binary_1 + num_binary_2
    elif integer == 1:  # 整数部分等于1
        num_binary_2 = ""  # 小数部分
        i = 0
        while i < bit:
```

```
            result = int(decimal * 2)
            decimal = decimal * 2 - result
            num_binary_2 += str(result)
            i += 1
        index = "01111111111"  # 指数部分
        x = len(index)
        num_binary = pm + index + num_binary_2
    else:  # 整数部分等于0
        num_binary_2 = ""  # 小数部分
        result = 0
        t = 0
        while result == 0:
            result = int(decimal * 2)
            decimal = decimal * 2 - result
            t += 1
        i = 0
        while i < bit:
            result = int(decimal * 2)
            decimal = decimal * 2 - result
            num_binary_2 += str(result)
            i += 1
        index = str(bin(1023 - t))[2:]  # 指数部分
        x = len(index)
        index_str = (11 - x) * '0' + index
        num_binary = pm + index_str + num_binary_2
    print(num_binary)

    cob = input("Continue or Break: ")
    if cob == "C" or cob == "Continue" :
        continue
    elif cob == "B" or cob == "Break":
        break
    else:
        print ("Invalid command! ")
        break
```

1.3.2 二进制转化成十进制

首先输入 64 位数字，对长度进行检验，如果长度不对则需要重新输入，另外进行 0 检验，如果输入为 0，则直接输出 0. 然后对小数部分和指数部分分别进行处理，指数部分通过 int(num,2) 函数转化为十进制数，再减去 1023 得到指数大小，2 的指数次幂得到一个指数部分乘积数，小数部分将 12 到 52 位的二进制数通过 for 循环 x 不断乘以 2 的-x 次方，最后把这些加起来得到小数部分的乘积数，小数部分的乘积数加一再乘上指数部分的乘积数得到最后结果，最后一步判断正负号，符号位为 0 则取负号。

```python
# coding=UTF-8
# 此程序用来将二进制浮点数转化为十进制数

for k in range(1,100):
    indexnumber = 0
    decimalnumber = 0
    num = input("Enter the input binary number:") # 输入二进制数
    while len(num) != 64: # 检验是否是64位
        print("please enter again:\n")
        num = input("Enter the input binary number:")
    if num == 64 * '0':
        print(0)
        exit()
    index = int(num[1:12], 2) - 1023
    indexnumber = 2 ** index # 指数部分
    for x in range(1, 53):
        decimalnumber += int(num[11 + x]) * (2 ** (-x)) # 小数部分
    number = indexnumber * (1 + decimalnumber)
    if num[0] == '1':
        number = -number
    print(number)
    cob = input("Continue or Break: ")
    if cob == "C" or cob == "Continue":
        continue
    elif cob == "B" or cob == "Break":
        break
    else:
        print("Invalid command! ")
        break
```

1.4 结果

首先检验十进制数转化为二进制数的程序：

输入：0
输出：00000000000000000000000000000000000000000000000000000000000000000000000000000000

输入：103.78
输出：01000000010110011111000111101011100001010001111010111000010100 10

输入：3456789.123
输出：01000001010010100101111110001010100011111011111001110110110010 01

输入：1.2345678
输出：00111111111100111100000011001010001010100101101100011101010111 01

输入：0.98765
输出：00111111111011111001101011010100001011000011110010011110111011 01

输入：-0.98765
输出：10111111111011111001101011010100001011000011110010011110111011 01

反过来，用刚才得到的结果来输入二进制转化十进制程序中：

输入：00000000000000000000000000000000000000000000000000000000000000000000000000000000
输出：0

输入：01000000010110011111000111101011100001010001111010111000010100 10
输出：103.78

输入：01000001010010100101111110001010100011111011111001110110110010 01
输出：3456789.123

输入：00111111111100111100000011001010001010100101101100011101010111 01
输出：1.2345678

输入：00111111111011111001101011010100001011000011110010011110111011 01
输出：0.98765

输入：10111111111011111001101011010100001011000011110010011110111011 01
输出：-0.98765

转化回来的结果与原结果一样，没有误差，可以认为达到了目标

Chapter 2

第二题

2.1 题目

（二）【10分】高斯求积具有很高的精确性，请分别构造如下三种非标准权函数的高斯求积公式的求积系数和节点：

$$\int_0^1 \sqrt{x} f(x)dx \approx A_0 f(x_0) + A_1 f(x_1); \tag{1}$$

$$\int_0^1 \sqrt{x} f(x)dx \approx A_0 f(x_0) + A_1 f(x_1) + A_2 f(x_2); \tag{2}$$

$$\int_{-1}^1 (1+x^2) f(x)dx \approx A_0 f(x_0) + A_1 f(x_1). \tag{3}$$

2.2 分析与思路

此处先求解公式 1 和公式 3，再求解公式 2，由简到繁，定义为方程 1，2，3

利用 guass 求积公式的相关推论：n+1 次正交多项式的零点是 n+1 阶 gauss 公式的 gauss 点。利用正交多项式得出 guass 点之后，利用利用代数精度的概念，分别令 $f(x) = 1, x, ...x^n$，解方程组可以得到。

具体来说，

第一步：利用 $\omega(x)$ 与 x 幂次多项式带权正交得到的积分公式为 0，得到关于 $\omega(x)$ 系数的多元方程。

第二步：求解线性方程组后得到高斯点。

第三步：将求解出来的高斯点回带到原高斯求积公式得到关于求积系数的线性方程组。

第四步：求解即可得到最终结果。

以下是每个公式的计算过程：

2.2.1 公式 1

$$\int_0^1 \sqrt{x} f(x)\,\mathrm{d}x \approx A_0 f(x_0) + A_1 f(x_1)$$

设$\omega(x) = (x - x_0)(x - x_1) = x^2 + bx + c$，则由正交性知$\omega(x)$与1及$x$带权正交，即得到

$$\int_0^1 \sqrt{x}\omega(x)\,\mathrm{d}x = 0, \quad \int_0^1 \sqrt{x}x\omega(x)\,\mathrm{d}x = 0$$

于是得到

$$\frac{2}{7} + \frac{2}{5}b + \frac{2}{3}c = 0 \quad 及 \quad \frac{2}{9} + \frac{2}{7}b + \frac{2}{5}c = 0$$

由此解得$b = -\frac{10}{9}, c = \frac{5}{21}$，即

$$\omega(x) = x^2 - \frac{10}{9}x + \frac{5}{21}$$

令$\omega(x) = 0$可解得

$$x_0 = 0.289949, \quad x_1 = 0.821162$$

原高斯求积公式对$f(x) = 1,\ x$精确成立，将两种情况分别带入，可得

$$A_0 + A_1 = \int_0^1 \sqrt{x}\mathrm{d}x = \frac{2}{3}, \quad A_0x_0 + A_1x_1 = \int_0^1 \sqrt{x} \cdot x\mathrm{d}x = \frac{2}{5}$$

由此解出

$$A_0 = 0.277556, \quad A_1 = 0.389111$$

2.2.2 公式 3

$$\int_{-1}^{1}(1+x^2)f(x)\,\mathrm{d}x \approx A_0 f(x_0)+A_1 f(x_1)$$

设$\omega(x)=(x-x_0)(x-x_1)=x^2+bx+c$，则由正交性知$\omega(x)$与1及$x$带权正交，即得到

$$\int_{-1}^{1}(1+x^2)\omega(x)\,\mathrm{d}x=0,\quad \int_{-1}^{1}(1+x^2)x\omega(x)\,\mathrm{d}x=0$$

于是得到

$$\frac{16}{15}+\frac{8}{3}c=0 \quad 及 \quad \frac{16}{15}b=0$$

由此解得$b=0,c=-\frac{2}{5}$，即

$$\omega(x)=x^2-\frac{2}{5}$$

令$\omega(x)=0$可解得

$$x_0=-\frac{\sqrt{10}}{5},\quad x_1=\frac{\sqrt{10}}{5}$$

原高斯求积公式对$f(x)=1,\ x$精确成立，将两种情况分别带入，可得

$$A_0+A_1=\int_{-1}^{1}(1+x^2)\mathrm{d}x=\frac{8}{3},\quad A_0x_0+A_1x_1=\int_{-1}^{1}(1+x^2)\cdot x\mathrm{d}x=0$$

由此解出

$$A_0=A_1=\frac{4}{3}$$

2.2.3 公式 2

$$\int_0^1 \sqrt{x} f(x)\,\mathrm{d}x \approx A_0 f(x_0) + A_1 f(x_1) + A_2 f(x_2)$$

设$\omega(x) = (x-x_0)(x-x_1)(x-x_2) = x^3 + ax^2 + bx + c$，则由正交性知$\omega(x)$与1、$x$及$x^2$带权正交，即得到

$$\int_0^1 \sqrt{x}\omega(x)\,\mathrm{d}x = 0, \quad \int_0^1 \sqrt{x}x\omega(x)\,\mathrm{d}x = 0, \quad \int_0^1 \sqrt{x}x^2\omega(x)\,\mathrm{d}x = 0$$

于是得到

$$\frac{2}{9} + \frac{2a}{7} + \frac{2b}{5} + \frac{2c}{3} = 0, \quad \frac{2}{11} + \frac{2a}{9} + \frac{2b}{7} + \frac{2c}{5} = 0, \quad \frac{2}{13} + \frac{2a}{11} + \frac{2b}{9} + \frac{2c}{7} = 0$$

由此解得$a = -\frac{21}{13}, b = \frac{105}{143}, c = -\frac{35}{429}$，即

$$\omega(x) = x^3 - \frac{21}{13}x^2 + \frac{105}{143}x - \frac{35}{429}$$

令$\omega(x) = 0$可解得

$$x_0 = 0.164710, \quad x_1 = 0.549868, \quad x_2 = 0.900806$$

原高斯求积公式对$f(x) = 1,\ x,\ x^2$精确成立，将三种情况分别带入，可得

$$A_0 + A_1 + A_2 = \int_0^1 \sqrt{x}\mathrm{d}x = \frac{2}{3},$$

$$A_0x_0 + A_1x_1 + A_2x_2 = \int_0^1 \sqrt{x}\cdot x\mathrm{d}x = \frac{2}{5},$$

$$A_0x_0^2 + A_1x_1^2 + A_2x_2^2 = \int_0^1 \sqrt{x}\cdot x^2\mathrm{d}x = \frac{2}{7}$$

由此解出

$$A_0 = 0.125782, \quad A_1 = 0.307603, \quad A_2 = 0.233281$$

2.3 算法描述和实现

2.3.1 预先定义的函数

高斯消元法

设定为 GEM(A,b,x), 利用高斯消元法解方程 Ax=b

```
def GEM(A, b, x):  # 高斯消元法。Ax=b
    n = len(A)  # A的行数
    m = len(A[0])  # A的列数
    B = [[0 for i in range(m)] for j in range(n)]  # B为增广矩阵
    for i in range(n):
        for j in range(n):
            # print(B)
            B[i][j] = A[i][j]  # 为增广矩阵赋值
        B[i].append(b[i])

    for i in range(n - 1):
        # 找主元
        PivotColumn = []
        for iPivot in range(i, n):
            PivotColumn.append(abs(B[iPivot][i]))
        if not max(PivotColumn):
            print('矩阵奇异')
            return

        PivotIndex = PivotColumn.index(max(PivotColumn))
        temp1 = B[i]
        B[i] = B[i + PivotIndex]
        B[i + PivotIndex] = temp1
        # 消元
        for iothers in range(i + 1, n):
            Coeff = B[iothers][i] / B[i][i]
            B[iothers][i] = 0
            for j in range(i + 1, m + 1):
                B[iothers][j] = B[iothers][j] - Coeff * B[i][j]

    # 反代
    for i in range(m - 1, -1, -1):
        x[i] = B[i][m]
        for j in range(m - 1, i, -1):
            x[i] = x[i] - B[i][j] * x[j]
        x[i] = x[i] / B[i][i]
    return x
```

向量内积，$\sqrt{x}$ 与 x^n 在 [0,1] 区间的定积分，$(1+x^2)$ 与 x^n 在 [-1,1] 区间的定积分，解一元二次方程组，计算行列式，解二元一次方程组

```
def multiple(list1, list2, list_result): # 向量内积
    for i in range(len(list_result)):
        for x in range(i + 1):
            list[i] += list[x] * list[i - x]


def f1(n):   # 定义与根号x在0到1区间的积分结果
    return 1/(n + 3/2)


def f2(n):      # 定义与(1+x^2.py)在-1到1区间的积分结果
    return (1/(n + 1) + 1/(n + 3)) * (1 ** (n + 1) - (-1) ** (n + 1))


def equation_2(list, list_result):  # 解一元二次方程
    a = list[2]
    b = list[1]
    c = list[0]
    delta = b * b - 4 * a * c
    if delta < 0:
        print('No result.')
        exit()
    x1 = (-b - math.sqrt(delta)) / (2 * a)
    x2 = (-b + math.sqrt(delta)) / (2 * a)
    list_result[0] = x1
    list_result[1] = x2


def det(a11, a12, a21, a22):        # 行列式
    return a11 * a22 - a21 * a12


def equa(a11, a12, a21, a22, b1, b2, list_result):   # 解二元一次方程组
    s = det(a11, a12, a21, a22)
    x1 = det(b1, a12, b2, a22) / s
    x2 = det(a11, b1, a21, b2) / s
    list_result[0] = x1
    list_result[1] = x2
```

2.3.2 主函数

第一个 gauss 积分的主函数

```
# 1
a1 = -f1(1) / f1(0)
b1 = -f1(2)
a11 = f1(1)
a12 = f1(0)
b2 = - f1(3)
a21 = f1(2)
a22 = f1(1)
list_1 = [0, 0]
equa(a11, a12, a21, a22, b1, b2, list_1)
list_X = [0, 0]
list = [list_1[1], list_1[0], 1]
equation_2(list, list_X)
list_A = [0, 0]
equa(1, 1, list_X[0], list_X[1], f1(0), f1(1), list_A)
print('方程1: \n','[x1,x2] =',list_X,'\n','[A1,A2] =',list_A)
```

第三个 gauss 积分的主函数

```
# 2.py

b1 = -f2(2)
a11 = f2(1)
a12 = f2(0)
b2 = -f2(3)
a21 = f2(2)
a22 = f2(1)
list_2 = [0, 0]
equa(a11, a12, a21, a22, b1, b2, list_2)
list_X = [0, 0]
list = [list_2[1], list_2[0], 1]
equation_2(list, list_X)
list_A = [0, 0]
equa(1, 1, list_X[0], list_X[1], f2(0), f2(1), list_A)
print('方程2: \n','[x1,x2] =', list_X, '\n', '[A1,A2] =', list_A)
```

第二个 gauss 积分的主函数

```
# 3
a = a1
b = list_1[0]
c = list_1[1]
A1 = [f1(2), f1(1), f1(0)]
```

```
A2 = [f1(3), f1(2), f1(1)]
A3 = [f1(4), f1(3), f1(2)]
A = [A1, A2, A3]
b = [-f1(3), -f1(4), -f1(5)]
x = [0, 0, 0]
GEM(A, b, x)
d = x[0]
e = x[1]
f = x[2]
p = e - d ** 2 / 3
q = f - d ** 3 / 27 - d / 3 * p
r = math.sqrt(-(p/3)**3)
theta = 1/3 * math.acos(-q /2/r)
u1 = 2 * math.pow(r, 1/3) * math.cos(theta)
u2 = 2 * math.pow(r, 1/3) * math.cos(theta + 2*math.pi/3)
u3 = 2 * math.pow(r, 1/3) * math.cos(theta + 4*math.pi/3)
x = [u1 - d/3, u2 - d/3, u3 - d/3]
A_1 = [1, 1, 1]
A_2 = [x[0], x[1], x[2]]
A_3 = [x[0]**2, x[1]**2, x[2]**2]
A = [A_1, A_2, A_3]
b = [f1(0), f1(1), f1(2)]
x_new =[0, 0, 0]
GEM(A, b, x_new)
print('方程3: \n','[x1,x2,x3] =',x,'\n','[A1,A2,A3] =',x_new)
```

2.4 结果

```
方程1:
 [x1,x2] = [0.2899491979256908, 0.8211619131854218]
 [A1,A2] = [0.27755599823106236, 0.3891106684356041]
方程2:
 [x1,x2] = [-0.6324555320336759, 0.6324555320336759]
 [A1,A2] = [1.3333333333333333, 1.3333333333333333]
方程3:
 [x1,x2,x3] = [0.9008058292716279, 0.1647102868965395, 0.5498684992164373]
 [A1,A2,A3] = [0.2332816246559186, 0.12578267432883503, 0.307602367681913]
```

Chapter 3

第三题

3.1 题目

（三）【10分】考虑A和B两类原子核随时间的放射衰变问题，t时刻，其布居数分别为$N_A(t)$和$N_B(t)$。假定A类核衰变为B类核，B类核可以继续衰变，满足以下微分方程组：

$$\frac{dN_A}{dt} = -\frac{N_A}{\tau_A}, \tag{4}$$

$$\frac{dN_B}{dt} = \frac{N_A}{\tau_A} - \frac{N_B}{\tau_B}, \tag{5}$$

其中，τ_A和τ_B为衰变时间常数。设时间起始点为$t_i = 0$时$N_A(t_i) = N_B(t_i) = 1$。（1）请给出上述方程的解析解；（2）使用合适算法数值求解上述耦合微分方程；（3）固定$\tau_A = 1$秒，分别讨论$\tau_B = 0.1$秒, 1秒, 10秒三种情况下的短期和长期衰变行为。选取$\tau_B = 10$秒这种情况，讨论你的数值算法的误差，展示取不同时间步长$\Delta t = 0.2, 0.1, 0.05$时与解析结果的比较。

3.2 分析与思路

对于（1）问来说，我们很容易通过手动解微分方程算得解析解，这也是后续计算精确解绘制曲线的原理：

$$N_A(t) = N_{0A} \times e^{-\frac{t}{\tau_A}} \tag{3.1}$$

$$N_B(t) = \frac{\frac{1}{\tau_A} N_{0A}}{\frac{1}{\tau_B} - \frac{1}{\tau_A}} \times (e^{-\frac{t}{\tau_A}} - e^{-\frac{t}{\tau_B}}) + e^{-\frac{t}{\tau_B}} \times N_{0B} \tag{3.2}$$

其中，N_{0B} 和 N_{0A} 表示初始时刻的粒子个数，此题中定为 1，τ 表示衰变特征时间。当我们通过程序去“重复”衰变过程时，我们并不需要直接求解微分方程，而是直接利用 t 时间和 t+step 时间的粒子数关系，即题目中所给的微分关系，进行模拟。

无论是对于 A 还是 B 而言，在原子衰变的程序中，核心是设置一个原子个数，随着时间按照步长的推移而逐次递减，若是要表示不同步长情况下的不同，就在最外围设置一个 for 循环，对步长值进行循环，依次画出不同步长值的情况，然后对这几种情况添加不同的图例；若要对不同的衰变特征时间进行模拟，就在最外层对不同长度的衰变

进行循环。最后在跳出循环后再画一个精确结果，即标准情况下原子数的衰减情况，与上述几种步长进行对比。程序的思想很简单，需要下载 python 下的模块 matplotlib 来进行画图。

3.3 算法描述和实现

衰变过程的 step-by-step 算法，A 粒子

```
# coding=UTF-8
# 此程序用来求衰变规律
import math
import matplotlib.pyplot as plt


#Na
i=0
for step in [0.4, 0.2, 0.1, 0.05]:  # 有四种时间步长，选定Taub=10,
    Taua=1
    Na = 1
    Nb = 1
    t = 0
    a = [0]
    b = [1]  # 添加初始值
    c = [1]
    while t <= 5:  # 添加元素
        Na = Na-step*Na
        Nb = Nb + (Na - Nb/10) * step
        t = t + step
        a.append(t)
        b.append(Na)
        c.append(Nb)
    plt.figure(1)
    i += 1
    if i == 1:  # 对于四种不同的步长使用不同标签标出
        plt.plot(a, b, label='step=0.4, Na')
    if i == 2:
        plt.plot(a, b, label='step=0.2, Na')
    if i == 3:
        plt.plot(a, b, label='step=0.1, Na')
    if i == 4:
        plt.plot(a, b, label='step=0.05, Na')

plt.figure(1)
x = [0]
for i in range(1, 101):  # 求出准确结果
    x.append(i * 0.05)
```

```
f2 = [1]
for i in range(1, 101):
    f2.append(1 * math.exp(-x[i]))


plt.plot(x, f2, label='accurate Na')
plt.title('decay curve')
plt.xlabel('Time/s')
plt.ylabel('N')
plt.legend()
plt.show()
```

衰变过程的 step-by-step 算法，B 粒子不同步长

```
#Nb, 不同步长
i = 0

for step in [0.4, 0.2, 0.1, 0.05]:  # 有四种时间步长，选定Taub=10, Taua=1
    Na = 1
    Nb = 1
    t = 0
    a = [0]
    b = [1]  # 添加初始值
    c = [1]
    while t <= 5:  # 添加元素
        Na = Na - step * Na
        Nb = Nb + (Na - Nb / 10) * step
        t = t + step
        a.append(t)
        b.append(Na)
        c.append(Nb)
    plt.figure(2)
    i += 1
    if i == 1:  # 对于四种不同的步长使用不同标签标出
        plt.plot(a, c, label='step=0.4, Nb')
    if i == 2:
        plt.plot(a, c, label='step=0.2, Nb')
    if i == 3:
        plt.plot(a, c, label='step=0.1, Nb')
    if i == 4:
        plt.plot(a, c, label='step=0.05, Nb')

plt.figure(2)
x = [0]
for i in range(1, 101):  # 求出准确结果
    x.append(i * 0.05)
```

```
f3 = [1]
for i in range(1, 101):
    f3.append((-10 / 9) * (math.exp(-x[i]) - math.exp((-x[i]) / 10)) +
        math.exp((-x[i]) / 10))

plt.plot(x, f3, label='accurate Nb')
plt.title('decay curve')
plt.xlabel('Time/s')
plt.ylabel('N')
plt.legend()
plt.show()
```

衰变过程的 step-by-step 算法，B 粒子不同特征时间

```
#Nb，不同衰变特征时间
k=0

for taub in [0.1, 1, 10]: #对不同的衰变特征时间，Taua=1
    Na = 1
    Nb = 1
    t = 0
    a = [0]
    b = [1]  # 添加初始值
    c = [1]
    step = 0.05
    while t <= 5:  # 添加元素
        Na = Na - step * Na
        Nb = Nb + (Na - Nb / taub) * step
        t = t + step
        a.append(t)
        c.append(Nb)
    plt.figure(3)
    k += 1
    if k==1:
        plt.plot(a, c, label='taub=0.1, Nb')
    if k==2:
        plt.plot(a, c, label='taub= 1, Nb')
    if k==3:
        plt.plot(a, c, label='taub= 10, Nb')

plt.figure(3)
x = [0]
for i in range(1, 101):  # 求出准确结果
    x.append(i * 0.05)
f3 = [1]
f4 = [1]
```

```
f5 = [1]
for i in range(1, 101):
    f3.append((1/9) * (math.exp(-x[i])-math.exp((-x[i])*10)) + math.
        exp((-x[i])*10))
plt.plot(x, f3, label='accurate Nb, taub = 0.1')
for i in range(1, 101):
    f4.append((1 + x[i])* math.exp(-x[i]))
plt.plot(x, f4, label='accurate Nb , taub = 1')
for i in range(1, 101):
    f5.append((-10/ 9) * (math.exp(-x[i]) - math.exp((-x[i]) / 10)) +
        math.exp((-x[i]) / 10))
plt.plot(x, f5, label='accurate Nb , taub = 10')

plt.title('decay curve')
plt.xlabel('Time/s')
plt.ylabel('N')
plt.legend()
plt.show()
```

3.4 结果

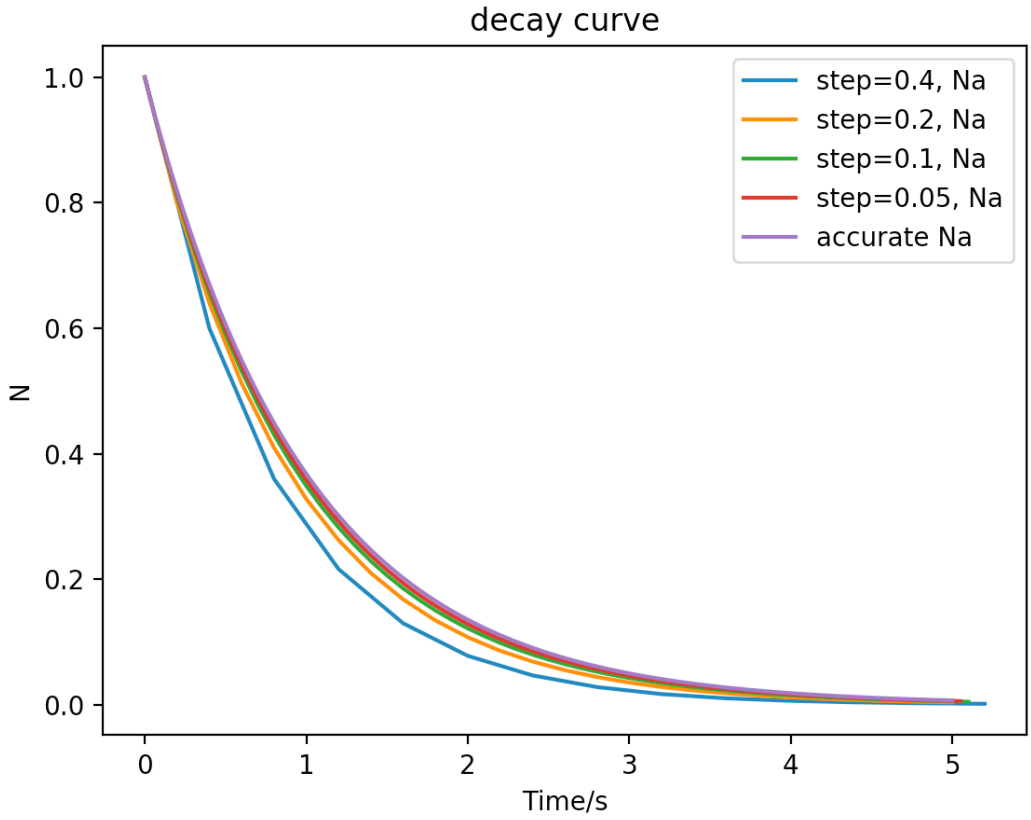

图 3.1: A 粒子

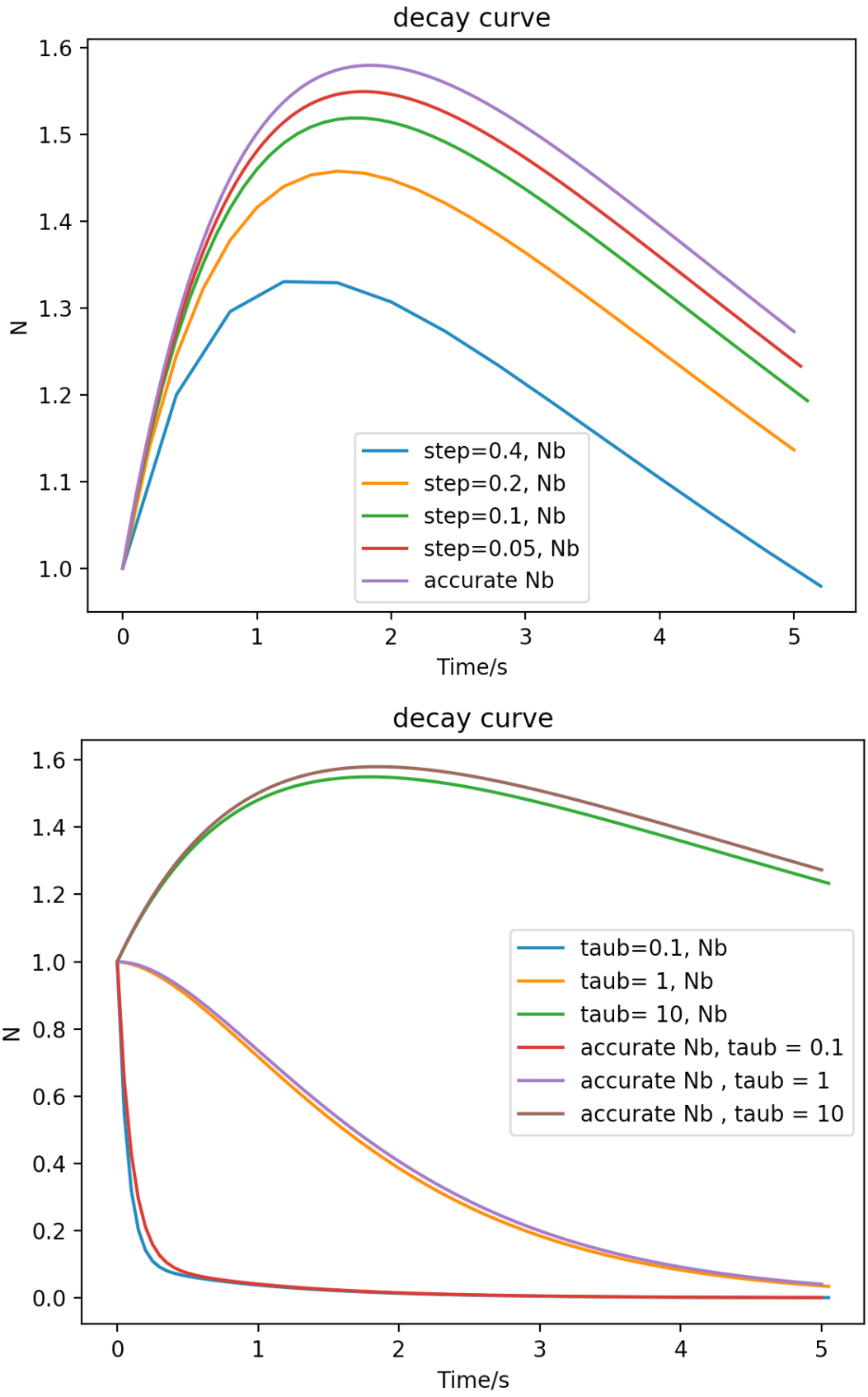

图 3.2: B 粒子，不同步长（上），不同衰变特征时间（下）

Chapter 4

第四题

4.1 题目

（四）【10分】在飞机设计中，已知机翼下轮廓上数据如下表，加工时，需要x每改变0.1米时的y 值，利用自然边界条件，试用三次样条插值估计任意一点y的值，与原始数据点一起在同一张图上绘制出飞机的轮廓曲线。

x	0	3	5	7	9	11	12	13	14	15
y	0	1.2	1.7	2.0	2.1	2.0	1.8	1.2	1.0	1.6

4.2 分析与思路

这一道题是根据课上所学知识写出三次样条插值曲线，算法课上已经讲的比较详细，主要是选择合适的算法以及实现即可。

三次样条插值是一种插值方法，在给定两点之间插入三次曲线，能够保证端点值的大小和不同段的端点处的一阶和二阶导数连续，要解出这些曲线还需要给出总区间的一些端点条件，本题的条件是自然边界条件，即端点处的二阶导数值为 0。

三次样条插值有两种表示方法，一种是根据一次导数表示，另一种是根据二次导数表示，此处采用了二次导数表示的方法，即在区间 $[x_j, x_{j+1}]$ 上的表达式为

$$S(x) = \frac{(x_{j+1}-x)^3}{6h_j}M_j + \frac{(x-x_j)^3}{6h_j}M_{j+1} + (y_j - \frac{M_j h_j^2}{6})\frac{x_{j+1}-x}{h_j} + (y_{j+1} - \frac{M_{j+1}h_j^2}{6})\frac{x-x_j}{h_j}$$

其中，$h_j = x_{j+1} - xj$ 要解出合适的m需要解方程方程内的数字为：

$$\mu_j = \frac{hj-1}{hj-1+h_j}$$

$$\lambda_j = \frac{hj}{hj-1+h_j}$$

$$d_j = \frac{6}{hj-1+hj}[-\frac{y_j - yj-1}{h_{j-1}} + \frac{y_{j+1}-y_j}{h_j}]$$

可以得到n-1个方程：

$$\begin{cases} \mu_1 M_0 + 2M_1 + \lambda_1 M_2 = d_1 \\ \mu_2 M_1 + 2M_2 + \lambda_2 M_3 = d_2 \\ \vdots \\ \mu_{n-1} M_{n-2} + 2M_{n-1} + \lambda_{n-1} M_n = d_{n-1} \end{cases}$$

由自然边界条件得$M_0 = 0, M_n = 0$,于是直接求解n-1 个方程：

$$\begin{bmatrix} 2 & \lambda_1 & & & \\ \mu_2 & 2 & \lambda_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \mu_{n-2} & 2 & \lambda_{n-2} \\ & & & \mu_{n-1} & 2 \end{bmatrix} \begin{bmatrix} M_1 \\ M_2 \\ \vdots \\ M_{n-2} \\ M_{n-1} \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-2} \\ d_{n-1} \end{bmatrix}$$

上述系数矩阵严格对角占优，方程可以用追赶法求解。
追赶法的原理是对系数矩阵进行克劳特分解$A = LU_0$，然后再进行求解，克劳特分解的原理是：

$$\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{bmatrix} = \begin{bmatrix} l_1 & & & & \\ \alpha_2 & l_2 & & & \\ & \ddots & \ddots & & \\ & & \alpha_{n-1} & l_{n-1} & \\ & & & \alpha_n & b_n \end{bmatrix} \begin{bmatrix} 1 & u_1 & & & \\ & 1 & u_2 & & \\ & & \ddots & \ddots & \\ & & & 1 & u_{n-1} \\ & & & & 1 \end{bmatrix}$$

解出M来就可以得到各段上的方程表达式，生成一些点根据表达式求出y值然后画图即可。

4.3 算法描述和实现

4.3.1 预先定义的函数

首先是定义分段的插值函数 S(x)，输入 x 值以及必要参数，返回 y 值。在这个函数里，首先需要判断输入的自变量 x 在哪个区间内。对于 x 所在的区间 $[x_j, x_{j+1}]$，则可以将 x 代入方程：

$$S(x) = \frac{(x_{j+1}-x)^3}{6h_j} M_j + \frac{(x-x_j)^3}{6h_j} M_{j+1} + (y_j - \frac{M_j h_j^2}{6}) \frac{x_{j+1}-x}{h_j} + (y_{j+1} - \frac{M_{j+1} h_j^2}{6}) \frac{x-x_j}{h_j}$$

从而得到相应的 y 值。程序如下：

```
def S(x, xlist, h, ylist, M):  # 定义三次样条函数，输入坐标，得到y值
    for i in range(len(xlist)-1):
        if x<=xlist[i+1] and x>=xlist[i]:
            j = i
```

```
    s = (xlist[j + 1] - x) ** 3 * M[j] / 6 / h[j] \
    + (x - xlist[j]) ** 3 * M[j + 1] / 6 / h[j]\
    + (ylist[j] - M[j] * h[j] ** 2 /6) *(xlist[j+1] - x) / h[j]\
    + (ylist[j + 1] - M[j + 1] * h[j] ** 2 / 6) *(x - xlist[j]) / h[j]
    return s
```

然后定义 chase 函数，进行追逐法解方程，输入值是 a,b,c,f 和 n, 能够解出 x 的 list。具体算法是 LU 分解法，先更新 b 和 m 的值，根据 b 和 m 的值算出 x 的值，b 和 f 是正向迭代更新，x 是反向迭代更新，具体公式如下：

$$b_i = b_i - m * c_{i-1}$$
$$f_i = f_i - m * f_{i-1}$$
$$x_{n-1} = \frac{f_{n-1}}{b_{n-1}}$$
$$x_i = (f_i - c_i * x - i + 1)/b_i$$

```
def chase(n, a, b, c, f):  # 定义追逐法解方程的函数
    for i in range(1, n):
        m = a[i]/b[i-1]
        b[i] = b[i] - m * c[i-1]
        f[i] = f[i] - m * f[i-1]
    x = [0] * n
    x[n-1] = f[n-1]/b[n-1]
    for i in range(n-2, -1, -1):
        x[i] = (f[i] - c[i] * x[i + 1])/b[i]
    return x
```

4.3.2 主函数

最后是主程序部分，把 x 和 y 的值分别赋给 x 和 y 的 list，然后根据公式求出 h 和 μ 和 λ 以及 d, 公式如下：

$$h_j = x_{j+1} - xj$$
$$\mu_j = \frac{hj - 1}{hj - 1 + h_j}$$
$$\lambda_j = \frac{hj}{hj - 1 + h_j}$$
$$d_j = \frac{6}{hj - 1 + hj}[-\frac{y_j - yj - 1}{h_{j-1}} + \frac{y_{j+1} - y_j}{h_j}]$$

由追逐法求出 M_1 和 M_{n-1} 后，加上 $M_0 = 0$ 和 $M_n = 0$，生成步长为 0.1 的 0 到 15 之间的等间距点，然后利用 S 函数求出对应的 y 值，将这些点利用绘图工具画出来即可。

```
x = [0, 3, 5, 7, 9, 11, 12, 13, 14, 15]  # 初始化后面要用到的变量
y = [0, 1.2, 1.7, 2.0, 2.1, 2.0, 1.8, 1.2, 1.0, 1.6]
h = [x[i+1] - x[i] for i in range(len(x)-1)]
miu = [(h[j])/(h[j] + h[j + 1]) for j in range(len(h) - 1)]
lam = [(h[j + 1])/(h[j] + h[j + 1]) for j in range(len(h) - 1)]
d = [6 / (h[j] + h[j + 1]) * ((y[j] - y[j + 1]) / h[j] + (y[j + 2] - y
    [j + 1]) / h[j + 1]) for j in range(len(h)-1)]
M = chase(len(h)-1, miu, [2]*(len(h)-1), lam, d)
M = [0] + M + [0]  # M的初末添加0
aa = [0.1 * i for i in range(151)]
bb = []
for i in range(len(aa)):
    bb.append(S(aa[i], x, h, y, M))
plt.plot(aa, bb)  # 绘图
plt.xlabel("x")
plt.ylabel("S(x)")
plt.title('Wing outline')
plt.show()
print(bb)
```

4.4 结果

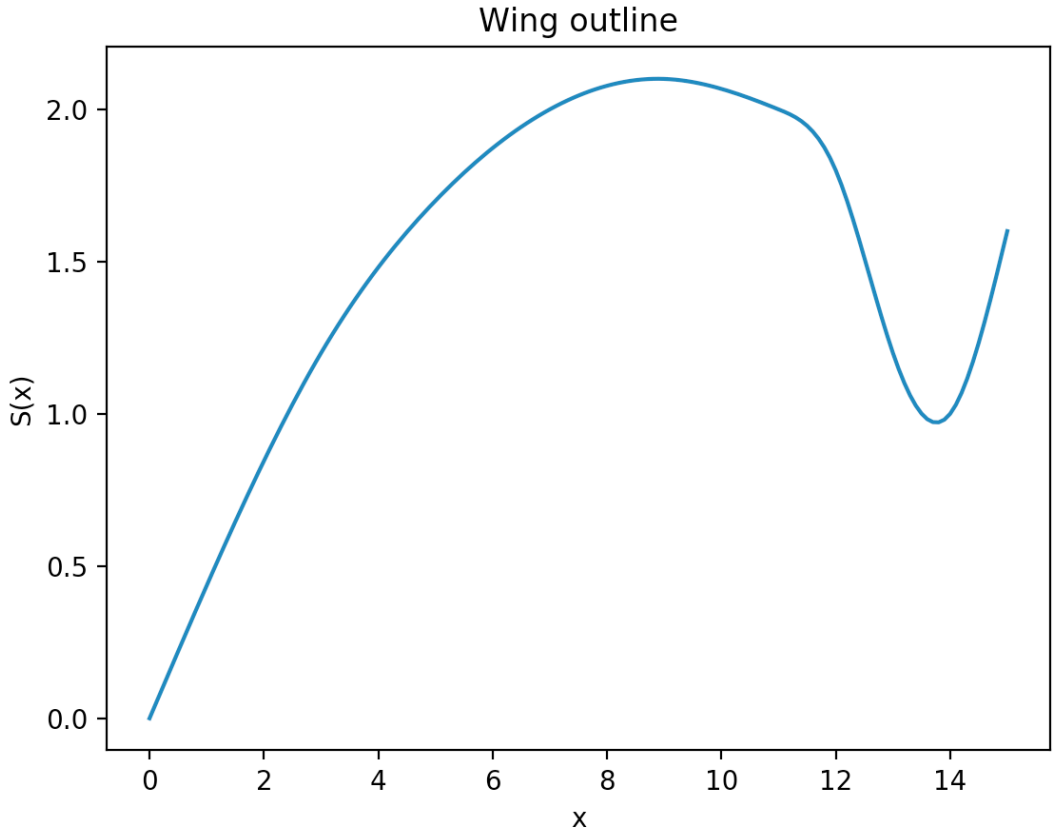

图 4.1: 机翼的轮廓，三次样条差值曲线

Chapter 5

第五题

5.1 题目

（五）【15分】课堂上讲过，带宽为$2m+1$的对称正定带状矩阵$\mathbf{A}$，可以分解为$\mathbf{A}=\mathbf{LDL}^T$，其中$\mathbf{L}$为下半带宽为$m$的带状阵。请根据课堂上讨论的一般对称正定带状系数矩阵方程组$\mathbf{Ax}=\mathbf{b}$的直接解法编写程序，注意计算时空复杂度要做到最小。给定对称阵带状矩阵$\mathbf{A}$，$m=2$，其非零元素: $a_{11}=a_{nn}=5$；$a_{ii}=6\ (i=2,...,n-1)$；$a_{i,i-1}=4\ (i=2,...,n)$；$a_{i,i-2}=1\ (i=3,...,n)$。右端向量$\mathbf{b}=(60,120,...,120,60)^T$。请取$n=100$和10000两个算例，展示你的结果。

5.2 分析与思路

这一题是通过分解一个对称正定带状矩阵，从而求解线性方程组。这个矩阵的参数有 N 和 m，N 是阶数，m 是半带宽。从一开始我们就需要定义这两个参数，然后开始定义其它函数和类，对带状矩阵我们可以设立一个叫做 band_matrix 类，这个类中包含 rank 阶数，band 半带宽，value 为按照讲义中的方式存储的 band*rank 阶矩阵 c，这样就可以尽可能地节省储存空间，因为这个空间和正常的矩阵空间可以通过变换得到，所以我也定义了函数 get_a 和 put_a 用于从 c 矩阵中取出 a 矩阵中的值，和写入 a 矩阵的值到 c 中，这样就可以更方便的从储存空间调用和写入，此外我还写了 3 个小的函数 list_sum, list_sumb, list_sumx，分别用来实现 $\Sigma\frac{l_{ik}\cdot l_{jk}}{l_{kk}}$, $\Sigma\frac{l_{ij}\cdot b_j}{l_{jj}}$, $\Sigma l_{ji}\cdot b_j$

在主要算法中，解方程的过程被分为三步，第一步是求出 L，第二步是求出变换后的 b，第三步是求解 x。在第一步中，我通过定义函数 Cholesky，对矩阵 A 进行逐行逐列的更新得到 L；第二部中，我通过定义 Choleskyb 对 b 也进行了类似的变换；第三步利用 solve 反向更新 x 向量，最后输出解。

5.3 算法描述和实现

5.3.1 矩阵类和预定义函数

a. 带状矩阵类和矩阵函数首先，设定带状矩阵类 band_matrix，因为带状矩阵大部分矩阵元为 0，因此为了节省空间，我们选择了 band*rank 阶的矩阵形式，考虑了初始化，和 assignment 幅值函数。其中，如上所述，函数 list_sum, list_sumb, list_sumx，分别用来实现 $\Sigma\frac{l_{ik}\cdot l_{jk}}{l_{kk}}$, $\Sigma\frac{l_{ij}\cdot b_j}{l_{jj}}$, $\Sigma l_{ji}\cdot b_j$

```
class band_matrix:  # 设立结构体
    def __init__(self, x, y):
        self.rank = x
        self.band = y
        self.value = [([0] * self.band) for n in range(self.rank)]

    def assignment(self, x, y, value):  # x,y都从0开始，赋值为value
        self.value[x][y] = value


def list_sum(i, j, start, end):  # 为后面的函数写一个a求和函数
    sum = 0
    for n in range(start, end+1):  # start 和 end都包括在内
        sum += get_a(i, n) * get_a(j, n) / get_a(n, n)
    return sum


def list_sumb(start, end, i):  # 为a和b写一个求和函数
    sum = 0
    for j in range(start, end+1):
        sum += get_a(i, j)*b[j-1]/get_a(j, j)
    return sum


def list_sumx(start, end, i):  # 为a和x写一个求和函数
    sum = 0
    for j in range(start, end + 1):
        sum += get_a(j, i) * x[j-1]
    return sum
```

因为实际的矩阵与存储这个对称带状矩阵的矩阵不同，在实际操作起来有一定麻烦，所以我写了两个函数便于读取和写入，分别是get_a 和put_a,能够从想象中的对称带状矩阵的i行j 列读取数据和写入数据，这这一考虑使得后边编写主函数时更直观。想象中的对称带状矩阵是这个样子的：

$$\begin{bmatrix} 5 & 4 & 1 & 0 & \cdots & 0 & 0 \\ 4 & 6 & 4 & 1 & \cdots & 0 & 0 \\ 1 & 4 & 6 & 4 & \cdots & 0 & 0 \\ 0 & 1 & 4 & 6 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 6 & 4 \\ 0 & 0 & 0 & 0 & \cdots & 4 & 5 \end{bmatrix}$$

在实际存储中的矩阵是这个样子的：

$$\begin{bmatrix} 0 & 0 & 5 \\ 0 & 4 & 6 \\ 1 & 4 & 6 \\ 1 & 4 & 6 \\ \vdots & \vdots & \vdots \\ 1 & 4 & 6 \\ 1 & 4 & 5 \end{bmatrix}$$

对比就可以发现大大节省了存储空间，理想数组中的（i，j）（非0）是实际存储矩阵中的（i-1，j-i+m），坐标对于理想数组是从1 开始，对于存储空间来说是从0 开始，通过这一关系，我们可以在实际空间和理想空间中来回切换，存取数据。

```
def get_a(i, j):  # 规定i《j,从带状数组的存储空间得到存储值
    if i <= m + 1:
        return A.value[i - 1][j-i+m]
    else:
        if j < i - m:
            return 0
        else:
            return A.value[i - 1][j - i + m]


def put_a(i, j, value):  # 此写的范围仅限于在c范围内，将得到的值写入a
    A.value[i - 1][j - i + m] = value
```

定义的 Cholesky 函数对 A 进行分解，$A = LDL^T$, 其中 L 为下半带宽为 m 的带状矩阵：

$$\begin{bmatrix} l_{11} & & & & & & \\ l_{21} & l_{22} & & & & & \\ \vdots & \ddots & \ddots & & & & \\ l_{m+1,1} & \cdots & l_{m+1,m} & l_{m+1,m+1} & & & \\ & \ddots & & \ddots & \ddots & & \\ & & l_{n,n-m} & \cdots & l_{n,n-1} & l_{n,n} \end{bmatrix}$$

当 i<j 或者 i-j>m 时，$l_{ij} = 0$, D 为对角阵，对角元素为 $d_{ii} = \frac{1}{l_{ii}}$，具体的导出 l_{ij}

的算法是：

$$l_{ij} = a_{ij} - \sum_{k=r}^{j-1} l_{ik} l_{jk} / l_{kk}$$

其中

$$r = \begin{cases} 1 & (i <= m+1) \\ i-m & (i > m+1) \end{cases} \tag{5.1}$$

a_{ij} 的值通过 get_a 函数取出，求和部分通过 list_sum 算出，最后得到的 l_{ij} 通过 put_a 写入，最终按行列更新 A 矩阵从而得到 L 矩阵。

```
def Cholesky():
    for i in range(2, A.rank+1):
        if i <= m + 1:# i在前面几行时
            for j in range(2, i + 1):
                r=1
                put_a(i, j, (get_a(i, j) - list_sum(i, j, r, j - 1)))
        else:
            for j in range(i-m, i+1):
                r = i - m  # 求和时在i-m开始求和
                put_a(i, j, (get_a(i, j) - list_sum(i, j, r, j - 1)))
                    # 将得到的a值写入
```

同理，在算法中，b 向量也要进行类似的分解，即 $b = LD\tilde{b}$，利用已有矩阵和向量得出计算公式：

$$\tilde{b_i} = b_i - \sum_{j-r}^{i-1} l_{ij} \tilde{b_j} / l_{jj}$$

r 仍然满足

$$r = \begin{cases} 1 & (i <= m+1) \\ i-m & (i > m+1) \end{cases} \tag{5.2}$$

具体实现的程序中，b_1 不用计算，直接从 b_2 到 b_n 进行循环，选取 r 值，r 值代表着求和的起始点。上式中的求和函数就通过此前定义的 list_sumb 实现，得到的值直接在 b 的存储空间更新。

```
def Choleskyb():  # 对b进行操作的函数
    for i in range(2, len(b)+1):
        if i <= m + 1:
            r = 1
        else:
            r = i - m
        b[i - 1] = b[i - 1] - list_sumb(r, i-1, i)
```

5.3.2 解带状矩阵主函数

由前面的分解 $A = LDL^T$ 和 $b = LD\tilde{b}$ 可以得到等价方程 $L^T x = \tilde{b}$，然后解这个方程即可。由于系数矩阵是上三角矩阵，所以该方程组的解很容易表示出来

$$x_i = (\tilde{b_i} - \sum_{j=i+1}^{t} l_j i x_j)/l_{ii}$$

求和中的 t 取下列值:

$$r = \begin{cases} n & (i > n-m-1) \\ i+m & (i <= n-m-1) \end{cases} \tag{5.3}$$

这个程序和前面最不一样的是他的解是倒序更新的，即从 x_{100} 开始逐个往前更新，对于 x_100，在 solve 函数的开始就把他更新完成，然后再对 x_99 到 x_1 倒序进行更新，进入一个 for 循环，在这个循环中对不同 case 进行 t 值的选取，然后运用公式对 x 逐个更新，储存在原列表中，list_sumx 函数即是用来计算公式中的求和函数。

最后，在主函数对各个预先定义好的函数进行调用。在初始化矩阵和向量后，依次使用 Cholesky ，Choleskyb，solve 函数，最后输出 x 的值即可。具体实现如下:

```
def solve():  # 对x进行操作的函数
    x[N-1] = b[N-1]/get_a(N, N)
    for i in range(N-1, 0, -1):
        if i > N-m-1:
            t = N
        else:
            t = i + m
        x[i-1] = (b[i - 1]-list_sumx(i+1, t, i))/get_a(i, i)


global A  # 定义A为全局变量
A = band_matrix(N, 3)
A.assignment(0, 2, 5)
A.assignment(N - 1, 2, 5)
for n in range(1, N - 1):
    A.assignment(n, 2, 6)
for n in range(1, N):
    A.assignment(n, 1, 4)
for n in range(2, N):
    A.assignment(n, 0, 1)  # 初始化A矩阵
global b
b = [60]
b += (N - 2) * [120]
b += [60]  # 初始化b
global x
x = [0] * A.rank
Cholesky()  # 依次进行三个函数的操作
```

```
Choleskyb()
solve()
print(x)
```

5.4 结果

N=100 时，得到了下面的结果：
(-475.24752474852374, 951.08910890299, -1368.118811869341, 1786.9306930535197, -2148.118811861469, 2512.2772276991323, -2819.9999999724528, 3131.8811880873754, -3388.5148514498433, 3650.4950494658, -3858.41584154119, 4072.871287081953, -4234.455445494031, 4403.762376183365, -4521.386138555898, 4647.920792017574, -4723.960395974342, 4810.099009832151, -4846.930692996951, 4895.049504874688, -4895.049504871312, 4907.524752392769, -4873.069306845004, 4852.277227633961, -4785.742574165588, 4734.05940584583, -4637.821782080636, 4557.623762275954, -4434.059405837728, 4327.7227721719, -4179.207920684409, 4049.108910781195, -3878.019801868201, 3726.5346533513684, -3535.2475246366375, 3364.7524751299484, 3155.64356423724, 2968.514851364451, -2743.960395917519, 2542.5742573023817, -2304.9504949249786, 2091.683168191247, -1843.3663365071284, 1620.5940592785619, -1363.9603959114884, 1134.0594058118488, -871.4851483855847, 636.831683038638, -370.69306917695013, 133.66336620646308, 133.6633664668809, -370.6930694371395, 636.8316832983705, -871.4851486446314, 1134.0594060699793, -1363.9603961684713, 1620.5940595341647, -1843.3663367611155, 2091.68316844338, -2304.9504951750164, 2542.5742575500826, -2743.9603961626376, 2968.51485160674, -3155.6435644764515, 3364.752475365832, -3535.247524868942, 3726.5346535798426, -3878.0198020925955, 4049.108911001259, -4179.207920899894, 4327.7227723825645, -4434.059406043336, 4557.623762476271, -4637.821782275437, 4734.059406034899, -4785.742574348725, 4852.277227810979, -4873.069307015729, 4907.524752557038, -4895.049505028966, 4895.049505025577, -4846.930693140932, 4810.099009969098, -4723.960396104141, 4647.92079214013, -4521.3861386711305, 4403.762376291207, -4234.455445594428, 4072.8712871748553, -3858.4158416265486, 3650.4950495435687, -3388.5148515199767, 3131.8811881498323, -2820.0000000271943, 2512.2772277461213, -2148.1188119006733, 1786.9306930849125, -1368.1188118929022, 951.089108918705, -475.24752475638365)

N=10000 的结果太长了，在此不予展示，可以在我的程序中随意设定 N 的值得到。可以看出两个 x 的解几乎都是左右对称的，这与前面的 A 和 b 都是对称的情况相符，是应该得到的结果。

5.5 误差分析

关于误差，理论上左右是完全对称的，但实际有所偏离，N=10000 和 N=100 时，相对偏差、绝对偏差如下：

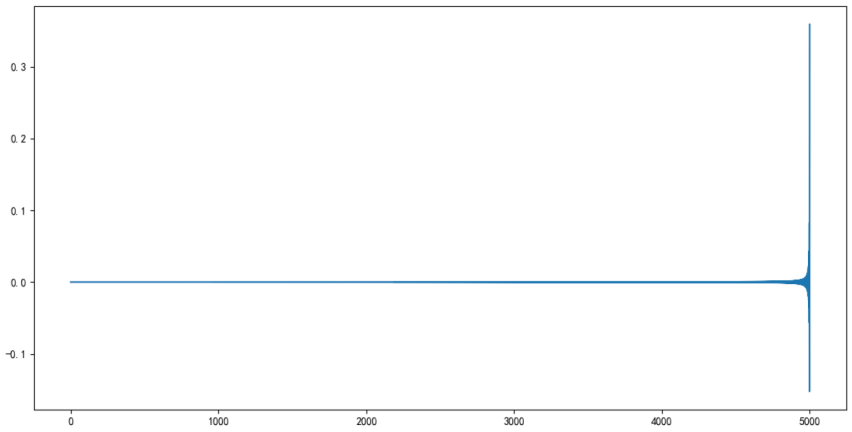

图 5.1: N=10000 相对偏差

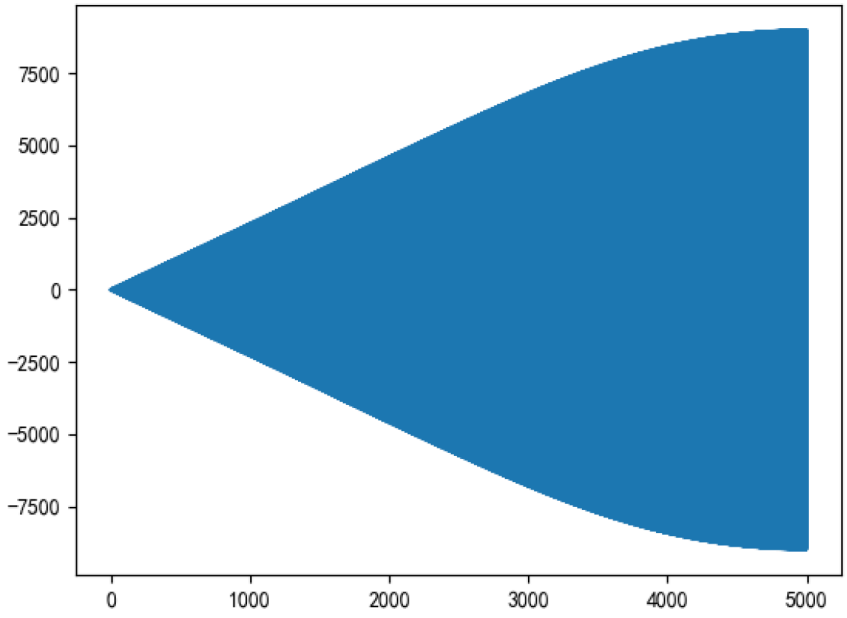

图 5.2: N=10000 绝对偏差

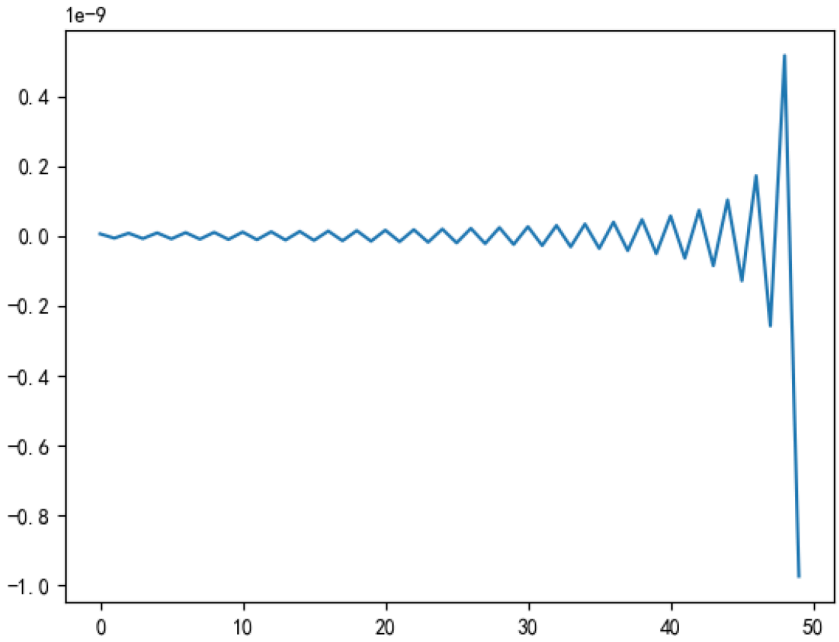

图 5.3: N=100 相对偏差

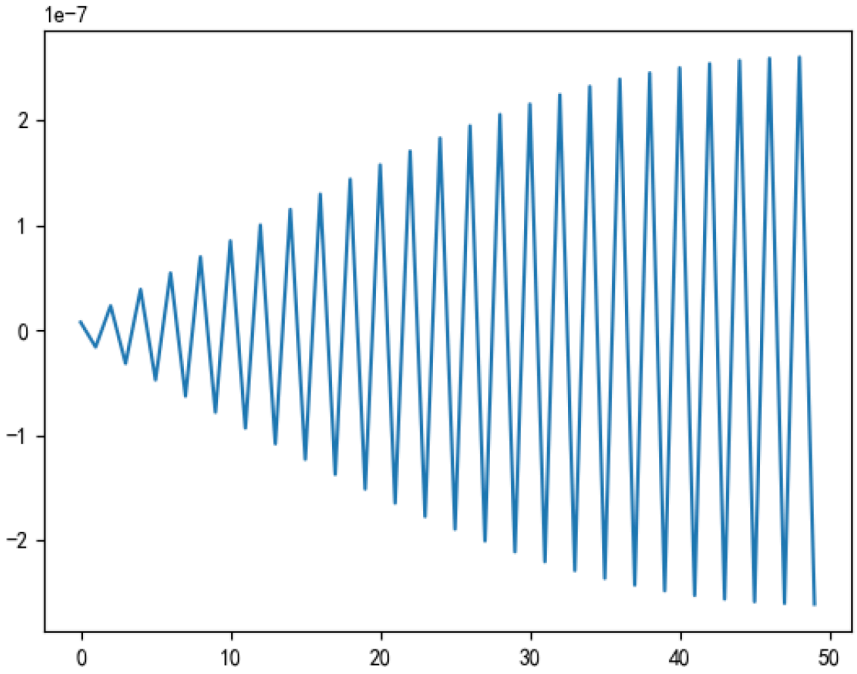

图 5.4: N=100 相对偏差

仔细分析会发现，问题是 N=10000 是解向量的中间部分是这样求得的：

```
x[i-1] = (b[i - 1]-list_sumx(i+1, t, i))/get_a(i, i)
```

这样解出来的解向量中部的数值很小，同时 list_sumx 函数在中部明显下降，如下二图所示，而 b 和 get_a 函数变化不明显。于是，绝对误差累积了起来，在中间解的数值变小，造成相对误差增大。

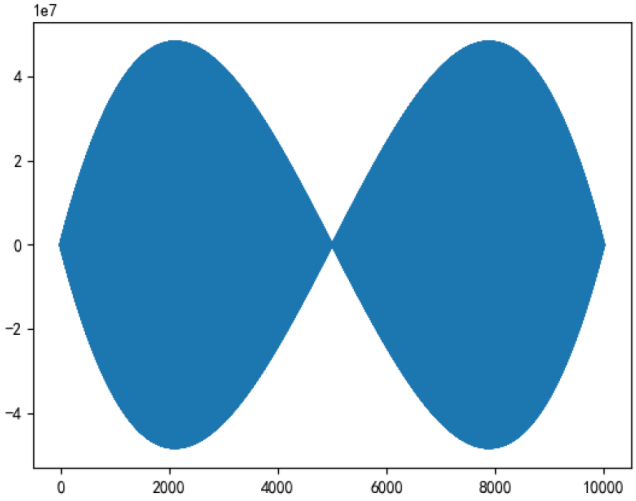

图 5.5: 解向量

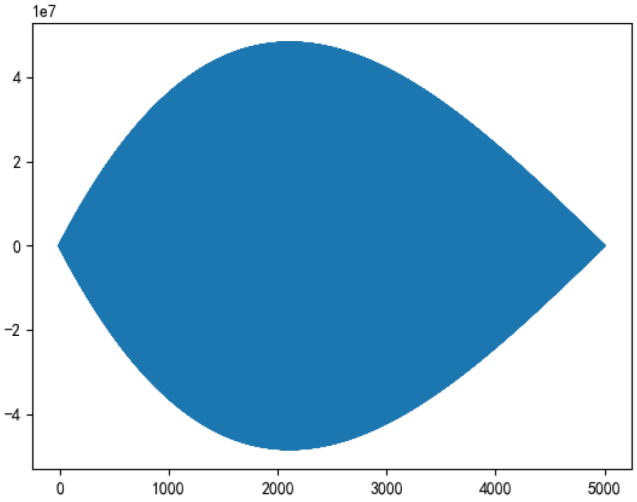

图 5.6: 求和函数

Chapter 6

第六题

6.1 题目

（六）【10分】设有稀疏矩阵**A**，即对角元均为3，两个次对角元均为−1；**除此三对角线之外但处于位置**$(i, n+1-i)$ $(i = 1, 2, ..., n)$**上的元素，值恒为**$\frac{1}{2}$；其余元素为0。例如，形式如下：

$$\mathbf{A} = \begin{bmatrix} 3 & -1 & 0 & \cdots & 0 & \frac{1}{2} \\ -1 & 3 & -1 & \cdots & \frac{1}{2} & 0 \\ 0 & -1 & 3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & \frac{1}{2} & 0 & \cdots & 3 & -1 \\ \frac{1}{2} & 0 & 0 & \cdots & -1 & 3 \end{bmatrix}. \tag{6}$$

对$\mathbf{b} = (2.5, 1.5, \cdots, 1.5, 1.0, 1.0, 1.5, \cdots, 1.5, 2.5)^T$，请用共轭梯度法求解线性方程组$\mathbf{Ax} = \mathbf{b}$，相邻两次的迭代误差要求小于$\varepsilon = 10^{-6}$。分别取$n = 100$和$n = 10000$两个算例，展示你的结果。

6.2 分析与思路

这一题是运用共轭梯度法解方程，方程的 A 和 b 均已给出，只是阶数不定，所以先设置一个变量 N 作为阶数可调，本题中的 N 是一个稀疏矩阵，不能用常规方法存储，于是便需要自定义一套存储方法：先定义一个类叫 matrix_value, 即稀疏矩阵中一个点的行和列和值，然后再定义一个类叫 sparse_matrix, 在这一类中储存整个稀疏矩阵整体的行数和列数，以及每个元素值，可以是无规则的，而且不进行行列重叠的检查，这是一个不太完善的点，不过如果稀疏矩阵已知的话，初始化的时候注意一点就好了，在这个类下还定义了一个操作叫 assignment, 可以对 list 进行添加元素的操作。

接着我定义了一系列对 list (也就是向量）进行操作的函数，包括 matrix_multiply, 可以对一个矩阵和向量进行乘法，定义了 list_minus 和 list_plus 可以对等长度的向量进行逐个相加和相减的操作，然后定义 list_distance, 可以用来计算两个向量之间的距离，定义了 list_multiply 用来计算两个向量的内积，上面定义的这些函数都有对输入维

度的检查，如果输入维度不合适的话就会报错。

函数预定义完之后是算法部分，先对 A 按照题目要求初始化，b 也按照题目要求初始化，x 向量也初始化到某个值。初始化的过程包括 N，可以更改 N 值定义而其他地方不变改变阶数，然后计算第一次的 r=b-Ax, p=r, k=0, k 记录迭代次数，下面是算法的主题 while 循环部分：当 k=0 或者相邻两次的迭代结果 x,x_old 的距离大于 1E-6 时进行迭代，迭代计算时先计算 α 的值，即 r 同转置的内积除以 p 转置与 Ap 的内积，然后令 x_old = x, 计算新的 $x = x + \alpha p$, r_old = r, 计算新的 $r = r - \alpha Ap$, 计算 $\beta = -r$ 和转置的内积除以 r_old 和其转置的内积，最后更新 p 值 $p = r + \beta p$，k=k+1，每迭代一次计算一次两相邻解的误差输出，可以看到误差确实是在减小的，要是计算 N=10000 的时候要很长时间也可以根据这个迭代误差来看进行到哪里了，最后迭代完成后输出得到的解。

6.3 算法描述和实现

6.3.1 预先定义的函数

首先是矩阵的定义与存储，下面是我的矩阵存储结构，因为本题是稀疏矩阵不能采用老式的存储思路，所以我定义总的矩阵是一个类，里面包含着行列以及各点值清单三个变量，各点值清单是以 list 的方式储存，这个类的每一个变量存放矩阵中一个点的行号和列号和值的大小。assignment 用来往 list 中添加点。

```
class matrix_value:  # 这一段定义了稀疏矩阵中的每一个值的结构
    def __init__(self, x, y, value):
        self.row = x
        self.column = y
        self.number = value


class sparse_matrix:  # 矩阵从1开始，这一段定义了矩阵的结构
    def __init__(self, x, y):
        self.row = x
        self.column = y
        self.value = []

    def assignment(self, x, y, value):  # 定义了对矩阵进行添加值的操作
        self.value.append(matrix_value(x, y, value))
```

然后定义矩阵、向量、常数之间运算的函数，包括加减乘除，和距离范数的定义。

```
def matrix_multiply(matrix, list):  # 定义了矩阵与向量相乘的函数
    if matrix.column != len(list):
        print("矩阵维度不匹配")
        exit()
    else:
```

```
        list_return = [0 for n in range(matrix.row)]
        for n in range(len(matrix.value)):
            list_return[matrix.value[n].row - 1] = list_return[matrix.
                value[n].row - 1] + matrix.value[n].number * list[
                matrix.value[n].column - 1]
    return list_return


def list_minus(list1, list2):  # 定义向量减法
    list_return = [0] * len(list2)
    if len(list1) != len(list2):
        print("列表长度不匹配")
        exit()
    else:
        for n in range(len(list1)):
            list_return[n] = list1[n] - list2[n]
    return list_return


def list_plus(list1, list2):  # 定义向量加法
    list_return = [0] * len(list2)
    if len(list1) != len(list2):
        print("列表长度不匹配")
        exit()
    else:
        for n in range(len(list1)):
            list_return[n] = list1[n] + list2[n]
    return list_return


def list_distance(list1, list2):  # 定义向量距离
    distance = 0
    if len(list1) != len(list2):
        print("列表长度不匹配")
        exit()
    else:
        for n in range(len(list1)):
            distance += (list1[n] - list2[n]) ** 2
        distance = distance ** 0.5
        return distance


def list_multiply(list1, list2):  # 定义向量乘法
    multiply = 0
    if len(list1) != len(list2):
        print("列表长度不匹配")
        exit()
```

```
    else:
53      for n in range(len(list1)): multiply += list1[n] * list2[n]
        return multiply
55
57  def list_nummultiply(num, list2): #定义常数向量乘法
        multiply = []
59      for n in range(len(list2)):
            multiply.append(num * list2[n])
61      return multiply
```

6.3.2 主函数

主函数部分，首先要对矩阵进行初始化，初始化的过程带着参数 N，以便于对两种情况都适用，并按照题目所给的方式初始化，注意交叉点的取值以主对角线的规律为准，由于预先定义的稀疏矩阵无法自检，所以初始化时一定不能有问题。然后对 b，x 也进行初始化，x 的初始化没有具体要求，为了方便在这里初始化为 0.

```
1   A = sparse_matrix(N, N)  # 对A进行初始化,此稀疏矩阵不能更新值
    for n in range(int(N / 2 - 1)):
3       A.assignment(n + 1, N - n - 1, 0.5)
    for n in range(int(N / 2 + 1), N):
5       A.assignment(n + 1, N - n - 1, 0.5)
    for n in range(N):
7       A.assignment(n + 1, n + 1, 3)
    for n in range(N - 1):
9       A.assignment(n + 1, n + 2, -1)
    for n in range(N - 1):
11      A.assignment(n + 2, n + 1, -1)

13  b = [2.5]  # 初始化b
    for n in range(int((N - 4) / 2)):
15      b.append(1.5)
    b.append(1)
17  b.append(1)
    for n in range(int((N - 4) / 2)):
19      b.append(1.5)
    b.append(2.5)
21
    x = [3 for n in range(N)]  # 初始化x
```

然后是具体的迭代过程，根据初始化的 A,b，x 也对 r,p 进行初始化，k 代表迭代次数，进入 while 循环后，运用共轭梯度法的原理求解：

搜索步长 $\alpha = \frac{r^T r}{p^T A p}$

更新解 $x = x + \alpha p$

保留残差向量 $r_{old} = r$
更新残差向量 $r = r - \alpha Ap$
计算搜索方向变量 $\beta = \frac{r^T r}{r_{old}^T r_o ld}$
计算新的搜索方向 $r = r + \beta p$

当 N=10000 时，要很长时间才能算出，所以在每个循环我加了一个输出迭代误差的质量，来确认误差在减少，最后输出结果即可。

```
r = list_minus(b, matrix_multiply(A, x))
p = r
k = 0
listplot=[]

while k == 0 or list_distance(x, x_old) >= 1E-6:
    alpha = list_multiply(r, r) / list_multiply(p, matrix_multiply(A,
        p))
    x_old = x
    x = list_plus(x, list_nummultiply(alpha, p))
    r_old = r
    r = list_minus(r, list_nummultiply(alpha, matrix_multiply(A, p)))
    beta = list_multiply(r, r) / list_multiply(r_old, r_old)
    p = list_plus(r, list_nummultiply(beta, p))
    #print("迭代误差为: ", math.log(list_distance(x, x_old), 10))
    listplot.append(math.log(list_distance(x, x_old), 10))
    k = k + 1

print(len(listplot))
print(listplot[0:30])
print("得到的解是: ",x)
```

6.4 结果

6.4.1 解的结果

N=100 时, 得到的解是:
(1.0000000066846837, 1.0000000045934885, 1.0000000078466043, 1.0000000068196864, 0.9999999985190978, 1.0000000117904193, 1.0000000032915703, 1.000000000123534, 0.9999999999766193, 0.9999999714144596, 0.9999999861485218, 0.9999999855535676, 1.0000000122740755, 1.000000039879814, 1.0000000295879996, 1.000000022613648, 0.9999999263621588, 0.999999882356668, 0.9999998777092098, 0.9999998975792438, 0.9999999296227713, 0.9999999299734375, 0.9999999999368451, 0.9999999999167769, 1.0000000016285473, 1.0000000016285473, 0.9999999926417528, 0.9999999846111833, 0.9999999182346436, 0.9999998863715065, 0.9999998827312742, 0.9999999379262625,

0.9999999917607247, 1.0000000291410325, 1.0000000524280221, 1.00000000792635, 0.9999999839230009, 0.9999999516923722, 0.9999999609634109, 0.9999999997552348, 1.0000000285384414, 1.0000000361705461, 1.0000000168543948, 0.999999976846537, 0.9999999691259728, 0.9999999787654281, 1.0000000090085548, 1.0000000332849148, 1.0000000157603544, 0.999999992332466, 0.9999999625039865, 0.9999999593294118, 0.9999999791318911, 1.0000000116427683, 1.0000000360386316, 1.0000000125373467, 0.9999999890908294, 0.9999999582407664, 0.999999959295192, 0.9999999903602995, 1.0000000083831986, 1.0000000176245445, 0.9999999928366323, 0.9999999571685215, 0.9999999614250423, 0.9999999738263899, 1.0000000463805099, 1.0000000853833968, 1.000000102104795, 1.0000001163967691, 1.000000066385395, 1.00000003937181, 1.0000000033302838, 1.0000000024844324, 1.0000000016285473, 1.0000000016285473, 1.0000000196021366, 1.0000000194014569, 1.0000001320078744, 1.0000001394087688, 1.0000000651446235, 1.0000000684365522, 1.0000000768213444, 1.0000000388649462, 1.00000002301945, 1.0000000102734476, 0.9999999700844595, 0.999999996176559, 0.9999999728743563, 0.9999999780427553, 0.999999976728035, 0.9999999815258124, 0.9999999815351444, 1.0000000007897716, 0.999999997481002, 1.0000000039677852, 1.0000000043234294, 0.9999999946403385, 1.0000000023686821, 0.9999999962314355)

对于 N=10000，得到的结果也都是在 1 左右，结果太长就不列出了。我们知道准确值都是 1。

6.4.2 误差下降分析

由于误差变化过于剧烈，y 轴取了以 10 为底的对数，可以看到，再从初始点经过第一次的下降后，开始平稳下降，可以看到梯度下降法的误差是单调下降的。

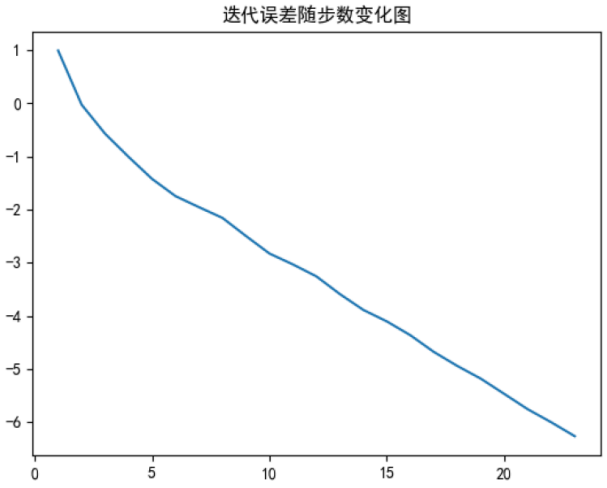

图 6.1: n=100, 误差随步数下降

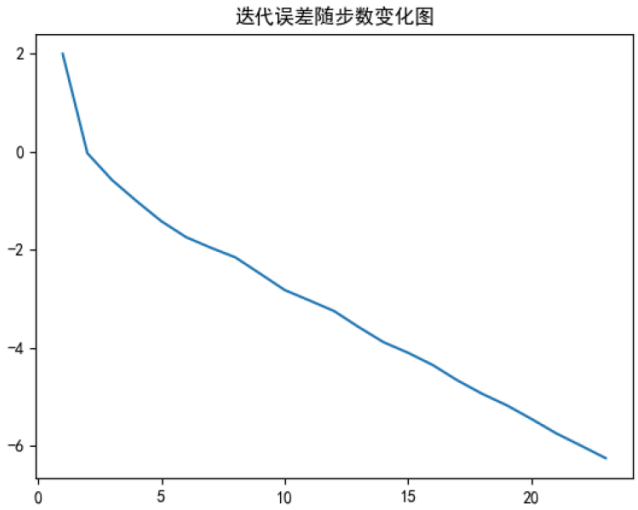

图 6.2: n=10000, 误差随步数下降

Chapter 7

第七题

7.1 题目

请解决以下问题：

（1）【10分】寻求可靠算法，编写程序数值求解广义拉盖尔多项式$L_n^\alpha(x)$。对$n=3,10,30$，分别列表给出$\alpha=2,20,40$和$x=10^{-3},10^0,10^2$各种组合时的数值结果。

（2）【10分】请自行查阅相关数学资料，设计算法，编写出高效而准确子函数程序Ylm，对任意给定L,M,θ,ϕ，返回精确的Ylm(L,M,θ,ϕ)值。作为测试，固定$\phi=\pi/5$，而$\theta=\pi/1000,3\pi/10,501\pi/1000$，列表给出$L=100,500,1000$，而$M$分别为$1,L/100,L/10,L-1$时，你的程序给出的数值结果。

（3）【5分】结合你的(1)、（2）程序，利用一定的作图软件，若以z为极轴，在x-y平面上展示电子的密度分布$\rho=|\Psi_{nlm}(r,\theta,\phi)|^2$，其中$n=2,l=1,m=\pm1,0$。

（4）【10分】试数值计算当$n=3,l=1,m=1$时的氢原子体系总能量$E=\langle\Psi|H|\Psi\rangle$，要求误差不大于$10^{-6}$。【$E_{311}$准确值为：$-\frac{1}{2n^2}=-\frac{1}{18}$】。**此小问提示：**球坐标系中氢原子的哈密顿算符为$H=-\frac{1}{2}\nabla^2-\frac{1}{r}$，而球坐标系中的拉普拉斯算符为：

7.2 分析与思路

7.2.1 第（1）问

其中$L_{n+l}^{2l+1}(\rho)$为广义拉盖尔多项式，而$\rho=\frac{2r}{n}$。广义拉盖尔多项式$L_n^\alpha(x)$满足以下递推关系式：

$$L_n^\alpha(x) = L_n^{(\alpha+1)}(x)-L_{n-1}^{(\alpha+1)}(x), \tag{8}$$

$$xL_n^{(\alpha+1)}(x) = (x-n)L_n^\alpha(x)+(n+\alpha)L_{n-1}^\alpha(x), \tag{9}$$

头几阶的具体表达式为：$L_0^\alpha(x)=1, L_1^\alpha(x)=-x+\alpha+1, L_2^\alpha(x)=\frac{1}{2}\left[x^2-2(\alpha+2)x+(\alpha+1)(\alpha+2)\right]$，$L_3^\alpha(x)=\frac{1}{6}\left[-x^3+3(\alpha+3)x^2-3(\alpha+2)(\alpha+3)x+(\alpha+1)(\alpha+2)(\alpha+3)\right]$.

直接利用题目中所给的递推关系，利用 L_{n-1}^{α} 迭代求解 L_n^{α}，即

$$L_t^{\alpha}(x) = \frac{((t+\alpha)\cdot L_{t-1}^{\alpha}(x) - x\cdot L_{t-1}^{\alpha+1}(x))}{t} \tag{7.1}$$

7.2.2 第（2）问

这一问是要求球谐函数的值,我把球谐函数分成三个部分,第一个部分是 $\sqrt{\frac{(2l+1)(l-m)!}{4\pi(l+m)!}}$ 第二部分是 $P_l^m(cos\theta)$，第三部分是 $e^{im\phi}$，第一部分直接用阶乘和开方把常数带入即可求得，第二部分是连带勒让德函数，是问题的主要求解点，第三部分是一个 e 指数部分，有可能是复数，所以要考虑复数的情况。在程序中，为了防止值上下溢出，特地在程序中编写了一套科学计数法来储存数的指数部分和浮点数部分。对于复数中间值，我选择将实部和虚部分开计算。球谐函数中连带勒让德由递推关系得到，由低阶向高阶计算，即先算出 P_m^m，再向上计算出 P_l^m，最后将三部分乘起来即可。递推关系如下:

$$P_{l+1}^{l+1}(x) = -(2l+1)\sqrt{1-x^2}P_l^l(x)$$
$$(l-m+1)P_{l+1}^m(x) = (2l+1)xP_l^m(x) - (l+m)P_{l-1}^m(x)$$

7.2.3 第（3）问

根据公式，直接将各部分相乘后，把数据导出到 MATLAB 中进行画图

$$\psi(r,\theta,\phi) = \sqrt{(\frac{2}{n})^3\frac{(n-l-1)!}{2n(n+l)!}}e^{-\rho/2}\rho^l L_{n-l-1}^{2l+1}(\rho)Y_l^m(\theta,\phi)$$

7.2.4 第（4）问

这个题看起来比较复杂其实没有很复杂，需要进行一系列的数学化简就会容易很多。所求能量

$$E = <\psi|H|\psi> = \int\int\int \psi^*(H\psi)r^2 dr sin\theta d\theta d\phi$$

我将ψ分成三个部分，第一部分是常数C，第二部分是和r有关的函数f（r），第三部分是球谐函数Y_l^m，第一部分在化简时可以提出来，对二三部分进行化简，将H的表达式带入，可以得到

$$E = C^2\int\int\int (f(r)Y)^*(\frac{f(r)}{2}\mathcal{L}Y - \frac{Y}{2}\mathcal{L}f(r) - \frac{f(r)Y}{r})r^2 dr sin\theta d\theta d\phi$$

而我们又根据球谐函数的性质知道

$$L^2Y_l^m(\theta,\phi) = l(l+1)Y_l^m(\theta,\phi)$$

而且我们根据球谐函数的归一性知道：

$$\int\int Y^*Y sin\theta d\theta d\phi = 1$$

就可以进一步得到不含球谐函数的积分式：

$$E = C^2 \int f(r)(\frac{l(l+1)f(r)}{2} - rf(r) - \frac{1}{2}\frac{\partial}{\partial r}(r^2\frac{\partial}{\partial r})f(r))dr$$

根据这个式子可以进一步计算本题已经指定了l，n，m就比较容易计算，可以得到：

$$f(r) = e^{-\frac{r}{3}}(\frac{2r}{3})(4 - \frac{2r}{3})$$

在不同的r处使用有限差分求导，再根据分段牛顿-科斯特算法算出积分即可，最后把常数项乘上去即得到积分。

7.3 算法描述和实现

7.3.1 第（1）问

简单地通过迭代递推公式就可以得到广义拉盖尔多项式，但可能由于步长问题，在 N 和 α 较大时可能会与标准值产生一些偏差。

```
def Lnax(n,alpha,x):
    t = n
    if n == 0:
        return 1
    else:
        while t > 0:
            a = (t+alpha)*Lnax(t-1,alpha,x) - x * Lnax(t-1,alpha+1,x)
            a = a / t
            t -= 1
            #print(t)
            return a
```

7.3.2 第（2）问

预定义科学计数法函数，将科学计数法浮点（小数）部分和指数部分，即 $a \times 10^b$ 中 a 和 b 分别储存成 [a,b]，并定义 scinum_adjust 函数将科学计数法转化成标准的我所需要的科学计数法形式。同时，我也定义了科学计数法记录的数值间进行的运算函数，加减乘除乘方开方阶乘运算，即

scinum_multiply 的作用是将两个科学计数法相乘，方法是指数部分相加，小数部分相乘并且最后做出调整。

scium_divide 的作用是将两个科学计数法相除，方法是指数部分相减，小数部分相除并且最后做出调整。

scinum_plus 的作用是将两个科学计数法相加，就是先找出指数项大的那一项，然后把指数小的那一项化成大指数下的小数与大指数项相加即可。scinumminus 类似。

scinum_sqrt 的作用是对一个科学计数法开方，当指数部分是偶数时，指数部分除二，小数部分开方即可。当指数部分是奇数时，指数减一，小数部分乘 10 再开方即可。

scinum_factorial 的作用是求一个整数的阶乘，输出是科学计数法的形式，如果是 0 直接输出 1，否则一直进行乘法和科学计数法转化的复合运算。

```
def scinum(a):  # a是小数项，b是指数项
    b = 0
    if abs(a) >= 10:
        while abs(a) >= 10:
            a = a/10
            b += 1
    elif abs(a) < 1:
        while abs(a) < 1:
            a = a*10
            b -= 1
    return [a, b]


def scinum_adjust(a):  # 用以稍作调整为规范科学计数法
    if abs(a[0]) >= 10:
        while abs(a[0]) >= 10:
            a[0] = a[0] / 10
            a[1] += 1
    elif abs(a[0]) < 1:
        while abs(a[0]) < 1:
            a[0] = a[0] * 10
            a[1] -= 1
    return a

def scinummultiply(a, b):  # 科学计数法的乘法
    c = [0] * 2
    c[0] = a[0] * b[0]
    c[1] = a[1] + b[1]
    return scinum_adjust(c)


def scinumdivide(a, b):  # 科学计数法的除法
    c = [0] * 2
    c[0] = a[0] / b[0]
    c[1] = a[1] - b[1]
    return scinum_adjust(c)


def scinumplus(a, b):  # 科学计数法的加法
```

```
    c = [0] * 2
    if a[1] > b[1]:
        c[1] = b[1]
        c[0] = a[0] + b[0] / (10 ** (a[1]-b[1]))
    else:
        c[1] = a[1]
        c[0] = b[0] + a[0] / (10 ** (b[1] - a[1]))
    return scinum_adjust(c)


def scinumminus(a, b):  # 科学计数法的减法
    c = [0] * 2
    if a[1] > b[1]:
        c[1] = a[1]
        c[0] = a[0] - b[0] / (10 ** (a[1]-b[1]))
    else:
        c[1] = b[1]
        c[0] = a[0] / 10 ** (b[1] - a[1]) - b[0]
    return scinum_adjust(c)


def scinumsqrt(a):  #科学计数法的开方
    if (a[1] % 2) == 0:
        a[1] = a[1]/2
        a[0] = math.sqrt(a[0])
    else:
        a[1] = (a[1] - 1)/2
        a[0] = math.sqrt((a[0]*10))
    return a


def scinumfactorial(t):  # 输入是整数，输出科学计数法的乘方
    a = [1, 0]
    if t==0:
        return a
    else:
        while t != 1:
            a = scinummultiply(a, scinum(t))
            t -= 1
        return a
```

接着定义连带勒让德函数和勒让德多项式，根据 ll 阶的勒让德函数递推公式定义 pll 函数，和 $P_0^0 = 1$ 可以递归得到。即根据

$$P_{l+1}^{l+1}(x) = -(2l+1)\sqrt{1-x^2}P_l^l(x) \tag{7.2}$$

对于勒让德函数，输入 l，m，x 就可以得到科学计数法的 P_l^m，具体计算方法是先

判断 m 正负，如果 m 为负数，则计算 $(-1)^m \frac{(l-m)!}{(l+m)!}$，如果 l==m，则计算 P_l^l，否则，先计算 P_m^m，然后计算 P_{m+1}^m，再逐次根据递推公式增加角标，直到达到所需的值为止。

```
def pll(l, xx): # 定义ll阶勒让德函数，返回值是科学计数法,输入值是浮点数
    if l == 0:
        return [1, 0]
    else:
        a = scinummultiply(scinum(-(2*l-1)),
                           scinummultiply(scinumsqrt(scinum(1 - xx **
                               2)), pll(l-1, xx)))
        return a


def legendre(l, m, x): # 定义勒让德函数
    if m < 0: # 如果m为负值
        m = -m
        a = scinummultiply(scinum(math.pow(-1, m)),
                           scinummultiply(scinumdivide(scinumfactorial(l
                               - m), scinumfactorial(l + m)), legendre(l,
                               m, x)))
        return a
    elif l == m: # 如果l=m返回pll值
        a = pll(l, x)
        return a
    else: # 否则利用递推公式
        t = m + 1
        mm = pll(m, x)
        mm2 = scinummultiply(scinum(x * (2 * m + 1)), pll(m, x))
        while l > t:
            t += 1
            mm_exchange = mm2
            mm2 = scinumdivide(scinumminus(scinummultiply(scinum((2 *
                t - 1) * x), mm2),scinummultiply(scinum(m + t - 1), mm)
                ),scinum(t - m))
            mm = mm_exchange
        return mm2
```

最后主函数执行三个部分即可，第一部分储存在 Ya 中，第二部分储存在 Yc 中，第三部分分别实部和虚部储存在 Yb_real 和 Yb_imag 中，最后把三个部分乘起来即可得到最后球谐函数的结果。对球谐函数而言，同理程序也将实部虚部分开储存，以 a+bi 的形式输出。实现如下：

```
for theta in [math.pi /1000, 3*math.pi/10 , 501* math.pi/1000 ]:
    for L in [100, 500 ,1000]:
        for M in [1, L/100, L/10, L-1 ]:
```

```
        phi = math.pi / 5
        x = math.cos(theta)
        Ya = scinumsqrt(scinumdivide(scinummultiply(scinum(2 * L +
            1), scinumfactorial(L - M)),scinummultiply(scinum(4 *
            math.pi), scinumfactorial(L + M))))# 计算第二部分

        Yb_real = scinum(math.cos(M * phi)) # 计算第三部分的实部
            和虚部
        Yb_imag = scinum(math.sin(M * phi))
        Yc = legendre(L, M, x) # 计算第一部分
        SH_real = scinummultiply(Ya, scinummultiply(Yb_real, Yc))
            # 计算球谐函数的实部和虚部
        SH_imag = scinummultiply(Ya, scinummultiply(Yb_imag, Yc))
        print('L=', str(L),'M=', str(M), 'theta=', str(theta), '
            phi=', str(phi),'球谐函数值为',SH_real[0],'* 10 ^',
              SH_real[1],'+',SH_imag[0],'* 10 ^',SH_imag[1],'i\n')
```

7.3.3 第（3）问

这一部分具体的计算方式非常简单，在题目所给的 n 和 l 的条件下，除了球谐函数，包括 R(r) 的部分，都可以通过简单的计算提前得到，在程序中把它定义为 cof1，分别与球谐函数的实部与虚部相乘，对空间的角度 θ, ϕ, r 按照要求进行积分，将得到的空间分布函数 ρ（程序中记为 Rou）输出到 MATLAB 进行画图。此处因为对称性的存在，做 xy 平面图时，θ 角固定为 $\pi/2$，而 ϕ 在 $(0, 2\pi)$ 以 $\pi/20$ 为间隔取了 400 个角度；同理，做 xz 平面图时，ϕ 角固定为 $\pi/5$，而 θ 在 M=0 时从 $(0, \pi)$ 以 $\pi/40$ 为间隔取了 400 个角度，在 M=1 时从 $(\pi/2, 3\pi/2)$ 以 $\pi/40$ 为间隔取了 400 个角度，最后做图。

7.3.4 第（4）问

首先利用程序定义 $f(r)$ 函数：

$$f(r) = e^{\frac{r}{3}}(\frac{2r}{3})(4 - \frac{2r}{3}) \tag{7.3}$$

接着定义基础微分函数。differential1 是两点插值微分，differential2 是三点插值微分，differential3 是五点插值微分，之所以要定义两点插值和三点插值是因为在边界处要用到，大多数地方用五点插值微分。两点插值微分中分为 Low 和 High 两种，用来计算下边界和上边界，微分的步长为 h1。

然后定义二阶微分函数：$\frac{\partial}{\partial r}(r^2 \frac{\partial}{\partial r})(f(r))$，很容易通过此前定义的一阶微分再取微分得到。此处 diferential1_two 是边界点的二阶微分，diferential3_two 是内部点的二阶微分。h2 是二阶微分的步长。上述过程实现如下：

```
def f(r):
    return (math.exp(-r / 3) * (2 * r / 3) * (4 - 2 * r / 3))
```

```
4
def differential1_low(h1, r):  # 定义了左端一阶导数
6     return (f(r + h1) - f(r)) / h1
8
def differential1_high(h1, r):  # 定义了右端一阶导数
10    return (f(r) - f(r - h1)) / h1
12
def differential2(h1, r):  # 定义了三点一阶导数
14    return (f(r + h1) - f(r - h1)) / (2 * h1)
16
def differential3(h1, r):  # 定义了五点一阶导数
18    return (f(r - 2 * h1) - 8 * f(r - h1) + 8 * f(r + h1) - f(r + 2 *
          h1)) / (12 * h1)
20
def differential1_two_low(h2, r):  # 定义了左端二阶导数
22    return ((r + h2) ** 2 * differential2(h2 / 3, r + h2) - (r) ** 2 *
           differential1_low(h2 / 3, r)) / h2
24
def differential1_two_high(h2, r):  # 定义了右端二阶导数
26    return ((r) ** 2 * differential1_high(h2 / 3, r) - (r - h2) ** 2 *
           differential2(h2 / 3, r - h2)) / h2
28
def differential3_two(h2, r):  # 定义了中间二阶导数
30    return (8 * (r + h2) ** 2 * differential3(h2 / 3, r + h2) - 8 * (r
          - h2) ** 2 * differential3(h2 / 3, r - h2) +
              (r - 2 * h2) ** 2 * differential3(h2 / 3, r - 2 * h2) - (r
                  + 2 * h2) ** 2 * differential3(h2 / 3, r + 2 * h2)) /
                 (12 * h2)
```

然后定义被积分函数，G1 是边界点，分为 Low 和 High，分别为上边界点和下边界点，G3 是中间点的函数，输入变量是 r 和 h，h 即为二阶微分的步长，G 的计算公式如下：

$$G = f(r)(\frac{l(l+1)f(r)}{2} - rf(r) - \frac{1}{2}\frac{\partial}{\partial r}(r^2\frac{\partial}{\partial r})(f(r))) \tag{7.4}$$

接着利用牛顿-科斯特公式进行积分，即

$$C_n = \frac{h}{90}[7f(a)+32\sum_{k=0}^{n-1} f(x_{k+\frac{1}{4}})+12\sum_{k=0}^{n-1} f(x_{k+\frac{1}{2}})+32\sum_{k=0}^{n-1} f(x_{k+\frac{3}{4}})+14\sum_{k=0}^{n-1} f(x_k)+7f(b)]$$

最后根据给定的 l,n,m 值计算出常数部分然后与积分值相乘就可以得到最后的结果。

```
def G1_low(r, h2):  # 定义了左端积分函数值
    G = f(r) * ((1 - r) * f(r) - 0.5 * differential1_two_low(h2, r))
    return G


def G1_high(r, h2):  # 定义了右端积分函数值
    G = f(r) * ((1 - r) * f(r) - 0.5 * differential1_two_high(h2, r))
    return G


def G3(r, h2):  # 定义了中间积分函数值
    G = f(r) * ((1 - r) * f(r) - 0.5 * differential3_two(h2, r))
    return G


def integral(h3):  # 复化牛顿–科斯特公式
    A = (7 * G1_high(60, h3 / 4) + 7 * G1_low(h3, h3 / 4))  # A是求和
        的左右端点值
    B=0
    C=0
    D=0
    E=0
    for i in range(int(60 / h3) ):
        B = B+ 32 * G3((i + 0.25) * h3, h3 / 4)
        C = C+12 * G3((i + 0.5) * h3, h3 / 4)
        D = D +32 * G3((i + 0.75) * h3, h3 / 4)


    for i in range(1, int(60 / h3) - 1):
        E = E+ 14 * G3((i) * h3, h3 / 4)

    intsum = h3 / 90 * (A + B + C + D + E)  # 对积分值进行加和
    return intsum


n = 3
l = 1
m = 1
h = 0.01  # 设置步长
C = (2 / n) ** 3 * math.factorial(n - l - 1) / (2 * n * math.factorial
    (n + l))
E = C * integral(h)  # 得到能量
print('l=1,m=1,n=3时的能量为: ', E)
```

7.4 结果

7.4.1 第（1）问

```
n= 3 alpha= 2 x= 0.001 Lnax= 9.990002499833334
n= 3 alpha= 2 x= 1 Lnax= 2.3333333333333335
n= 3 alpha= 2 x= 100 Lnax= -142656.66666666666
n= 3 alpha= 20 x= 0.001 Lnax= 1770.7470114998332
n= 3 alpha= 20 x= 1 Lnax= 1529.3333333333333
n= 3 alpha= 20 x= 100 Lnax= -75195.66666666667
n= 3 alpha= 40 x= 0.001 Lnax= 12340.097021499832
n= 3 alpha= 40 x= 1 Lnax= 11459.333333333334
n= 3 alpha= 40 x= 100 Lnax= -29625.666666666668
n= 10 alpha= 2 x= 0.001 Lnax= 65.7802473680385
n= 10 alpha= 2 x= 1 Lnax= -5.95454668209877
n= 10 alpha= 2 x= 100 Lnax= 7117724862237.076
n= 10 alpha= 20 x= 0.001 Lnax= 30030710.776123226
n= 10 alpha= 20 x= 1 Lnax= 18348617.842650738
n= 10 alpha= 20 x= 100 Lnax= 566337213392.4255
n= 10 alpha= 40 x= 0.001 Lnax= 10269773004.722675
n= 10 alpha= 40 x= 1 Lnax= 8019281167.169131
n= 10 alpha= 40 x= 100 Lnax= 7028595559.772855
n= 30 alpha= 2 x= 0.001 Lnax= 491.0579464750633
n= 30 alpha= 2 x= 1 Lnax= 4.9141635942154664
n= 30 alpha= 2 x= 100 Lnax= -1.7601471920423125e+20
n= 30 alpha= 20 x= 0.001 Lnax= 47061929154802.664
n= 30 alpha= 20 x= 1 Lnax= 10329978992206.137
n= 30 alpha= 20 x= 100 Lnax= 4.805973795332116e+16
n= 30 alpha= 40 x= 0.001 Lnax= 5.530725569029713e+19
n= 30 alpha= 40 x= 1 Lnax= 2.6212505365081723e+19
n= 30 alpha= 40 x= 100 Lnax= 290655108971274.3
```

7.4.2 第（2）问

```
L= 100 M= 1 theta= 0.0031415926535897933 phi= 0.6283185307179586 球谐函数值为 -5.044392891064218 * 10 ^ -1.0 + -3.6649659633260683 * 10 ^ -1.0 i
L= 100 M= 1.0 theta= 0.0031415926535897933 phi= 0.6283185307179586 球谐函数值为 -5.044392891064218 * 10 ^ -1.0 + -3.6649659633260683 * 10 ^ -1.0 i
L= 100 M= 10.0 theta= 0.0031415926535897933 phi= 0.6283185307179586 球谐函数值为 1.039774918421693 * 10 ^ -14.0 + -2.546714051357653 * 10 ^ -30.0 i
L= 100 M= 99 theta= 0.0031415926535897933 phi= 0.6283185307179586 球谐函数值为 -1.7933760569019725 * 10 ^ -247.0 + 1.3029639740458612 * 10 ^ -247.0 i
L= 500 M= 1 theta= 0.0031415926535897933 phi= 0.6283185307179586 球谐函数值为 -4.094036464409247 * 10 ^ 0.0 + -2.974491602598024 * 10 ^ 0.0 i
L= 500 M= 5.0 theta= 0.0031415926535897933 phi= 0.6283185307179586 球谐函数值为 2.0134614652412184 * 10 ^ -2.0 + -2.4657791386141983 * 10 ^ -18.0 i
L= 500 M= 50.0 theta= 0.0031415926535897933 phi= 0.6283185307179586 球谐函数值为 1.5931810880211537 * 10 ^ -69.0 + -1.9510841199072035 * 10 ^ -84.0 i
L= 500 M= 499 theta= 0.0031415926535897933 phi= 0.6283185307179586 球谐函数值为 -4.334243906770455 * 10 ^ -1248.0 + 3.149012525017036 * 10 ^ -1248.0 i
L= 1000 M= 1 theta= 0.0031415926535897933 phi= 0.6283185307179586 球谐函数值为 -2.899262267621155 * 10 ^ 0.0 + -2.106437337268029 * 10 ^ 0.0 i
L= 1000 M= 10.0 theta= 0.0031415926535897933 phi= 0.6283185307179586 球谐函数值为 2.5474704325863753 * 10 ^ -4.0 + -6.239503022378856 * 10 ^ -20.0 i
L= 1000 M= 100.0 theta= 0.0031415926535897933 phi= 0.6283185307179586 球谐函数值为 4.803967962428549 * 10 ^ -138.0 + -1.1766327976789113 * 10 ^ -152.0 i
L= 1000 M= 999 theta= 0.0031415926535897933 phi= 0.6283185307179586 球谐函数值为 -2.7364256252968664 * 10 ^ -2499.0 + 1.9881295915017159 * 10 ^ -2499.0 i
L= 100 M= 1 theta= 0.9424777960769379 phi= 0.6283185307179586 球谐函数值为 8.773713914333182 * 10 ^ -2.0 + 6.3744762873154395 * 10 ^ -2.0 i
L= 100 M= 1.0 theta= 0.9424777960769379 phi= 0.6283185307179586 球谐函数值为 8.773713914333182 * 10 ^ -2.0 + 6.3744762873154395 * 10 ^ -2.0 i
L= 100 M= 10.0 theta= 0.9424777960769379 phi= 0.6283185307179586 球谐函数值为 -3.5484056194773492 * 10 ^ -1.0 + 8.691087167938836 * 10 ^ -17.0 i
L= 100 M= 99 theta= 0.9424777960769379 phi= 0.6283185307179586 球谐函数值为 -4.930969739298411 * 10 ^ -9.0 + 3.582559219907847 * 10 ^ -9.0 i
L= 500 M= 1 theta= 0.9424777960769379 phi= 0.6283185307179586 球谐函数值为 8.832497657762032 * 10 ^ -2.0 + 6.417185176871856 * 10 ^ -2.0 i
L= 500 M= 5.0 theta= 0.9424777960769379 phi= 0.6283185307179586 球谐函数值为 -1.0329904182807348 * 10 ^ -1.0 + 1.2650484092973873 * 10 ^ -17.0 i
L= 500 M= 50.0 theta= 0.9424777960769379 phi= 0.6283185307179586 球谐函数值为 -2.3162506176285267 * 10 ^ -2.0 + 2.836588904901855 * 10 ^ -17.0 i
L= 500 M= 499 theta= 0.9424777960769379 phi= 0.6283185307179586 球谐函数值为 -2.5093938480962343 * 10 ^ -45.0 + 1.8231813501570535 * 10 ^ -45.0 i
L= 1000 M= 1 theta= 0.9424777960769379 phi= 0.6283185307179586 球谐函数值为 8.83989862185553 * 10 ^ -2.0 + 6.422562292034023 * 10 ^ -2.0 i
L= 1000 M= 10.0 theta= 0.9424777960769379 phi= 0.6283185307179586 球谐函数值为 -3.403239675121182 * 10 ^ -1.0 + 8.335533149736868 * 10 ^ -17.0 i
L= 1000 M= 100.0 theta= 0.9424777960769379 phi= 0.6283185307179586 球谐函数值为 -3.4914283177149725 * 10 ^ -1.0 + 8.551533027484139 * 10 ^ -16.0 i
L= 1000 M= 999 theta= 0.9424777960769379 phi= 0.6283185307179586 球谐函数值为 -4.0186693063359993 * 10 ^ -91.0 + 2.9197341570427255 * 10 ^ -91.0 i
L= 100 M= 1 theta= 1.5739379194484864 phi= 0.6283185307179586 球谐函数值为 -7.996089588471793 * 10 ^ -2.0 + -5.809499143765643 * 10 ^ -2.0 i
L= 100 M= 1.0 theta= 1.5739379194484864 phi= 0.6283185307179586 球谐函数值为 -7.996089588471793 * 10 ^ -2.0 + -5.809499143765643 * 10 ^ -2.0 i
```

```
L= 100 M= 1.0 theta= 1.5739379194484864 phi= 0.6283185307179586 球谐函数值为 -7.996089588471793 * 10 ^ -2.0 + -5.809499143765643 * 10 ^ -2.0 i
L= 100 M= 10.0 theta= 1.5739379194484864 phi= 0.6283185307179586 球谐函数值为 -3.0348271782153597 * 10 ^ -1.0 + 7.433182779533669 * 10 ^ -17.0 i
L= 100 M= 99 theta= 1.5739379194484864 phi= 0.6283185307179586 球谐函数值为 3.4107023733074286 * 10 ^ -2.0 + -2.4780203245766943 * 10 ^ -2.0 i
L= 500 M= 1 theta= 1.5739379194484864 phi= 0.6283185307179586 球谐函数值为 -2.575186188080206 * 10 ^ -1.0 + -1.8709822831722818 * 10 ^ -1.0 i
L= 500 M= 5.0 theta= 1.5739379194484864 phi= 0.6283185307179586 球谐函数值为 3.1831817975841195 * 10 ^ -1.0 + -3.8982733995155106 * 10 ^ -17.0 i
L= 500 M= 50.0 theta= 1.5739379194484864 phi= 0.6283185307179586 球谐函数值为 -2.0085386986391547 * 10 ^ -3.0 + 2.459750488252031 * 10 ^ -18.0 i
L= 500 M= 499 theta= 1.5739379194484864 phi= 0.6283185307179586 球谐函数值为 1.1364889468913306 * 10 ^ -1.0 + -8.257075525246297 * 10 ^ -2.0 i
L= 1000 M= 1 theta= 1.5739379194484864 phi= 0.6283185307179586 球谐函数值为 4.042061735966816 * 10 ^ -4.0 + 2.9367297520030675 * 10 ^ -4.0 i
L= 1000 M= 10.0 theta= 1.5739379194484864 phi= 0.6283185307179586 球谐函数值为 3.183182837879508 * 10 ^ -1.0 + -7.796549347019856 * 10 ^ -17.0 i
L= 1000 M= 100.0 theta= 1.5739379194484864 phi= 0.6283185307179586 球谐函数值为 -3.1907861260872075 * 10 ^ -1.0 + 7.815172032152964 * 10 ^ -16.0 i
L= 1000 M= 999 theta= 1.5739379194484864 phi= 0.6283185307179586 球谐函数值为 1.9062716263727473 * 10 ^ -1.0 + -1.3849874064896626 * 10 ^ -1.0 i
```

7.4.3 第（3）问

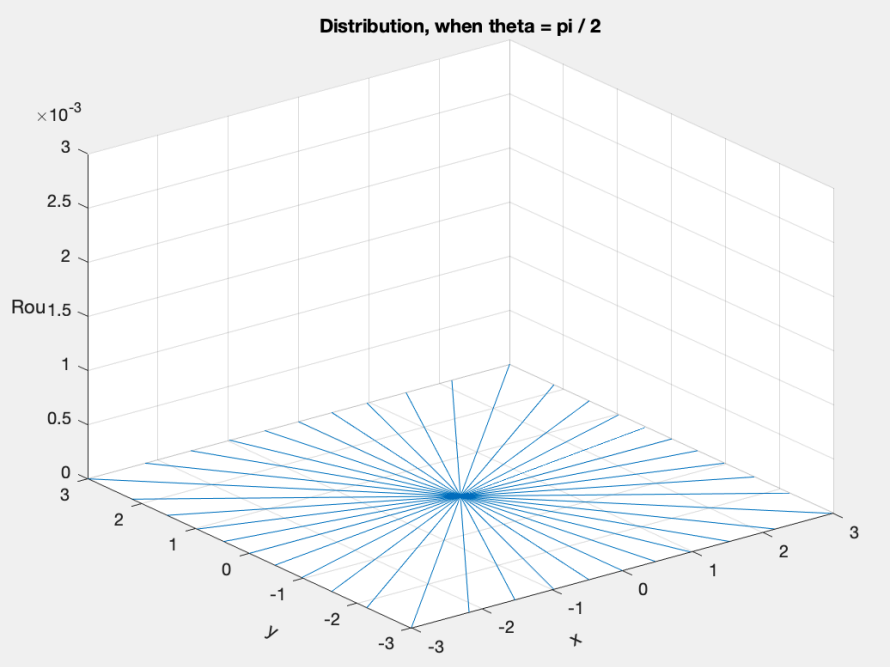

图 7.1: M=0,xy 平面图

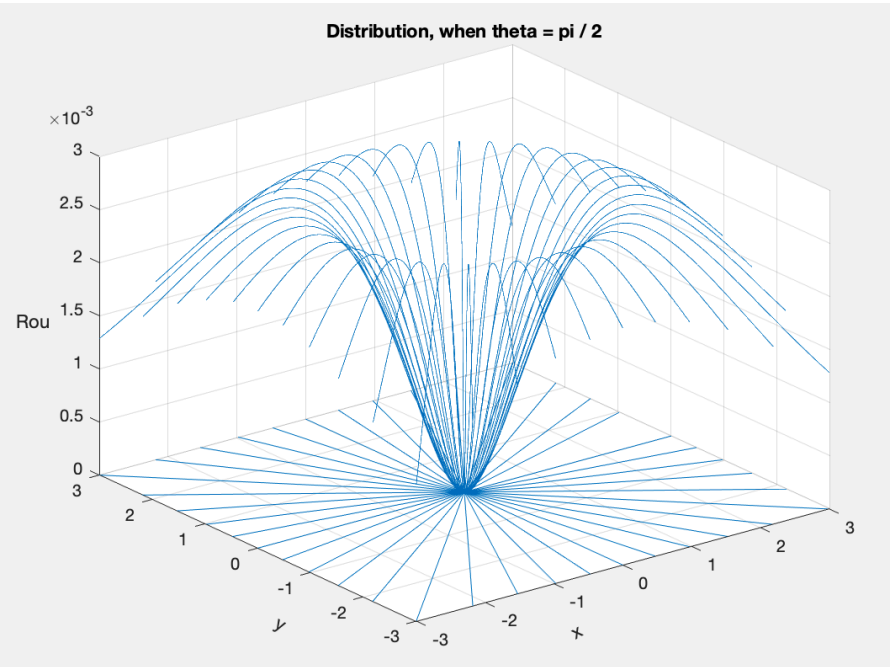

图 7.2: M=-1,xy 平面图

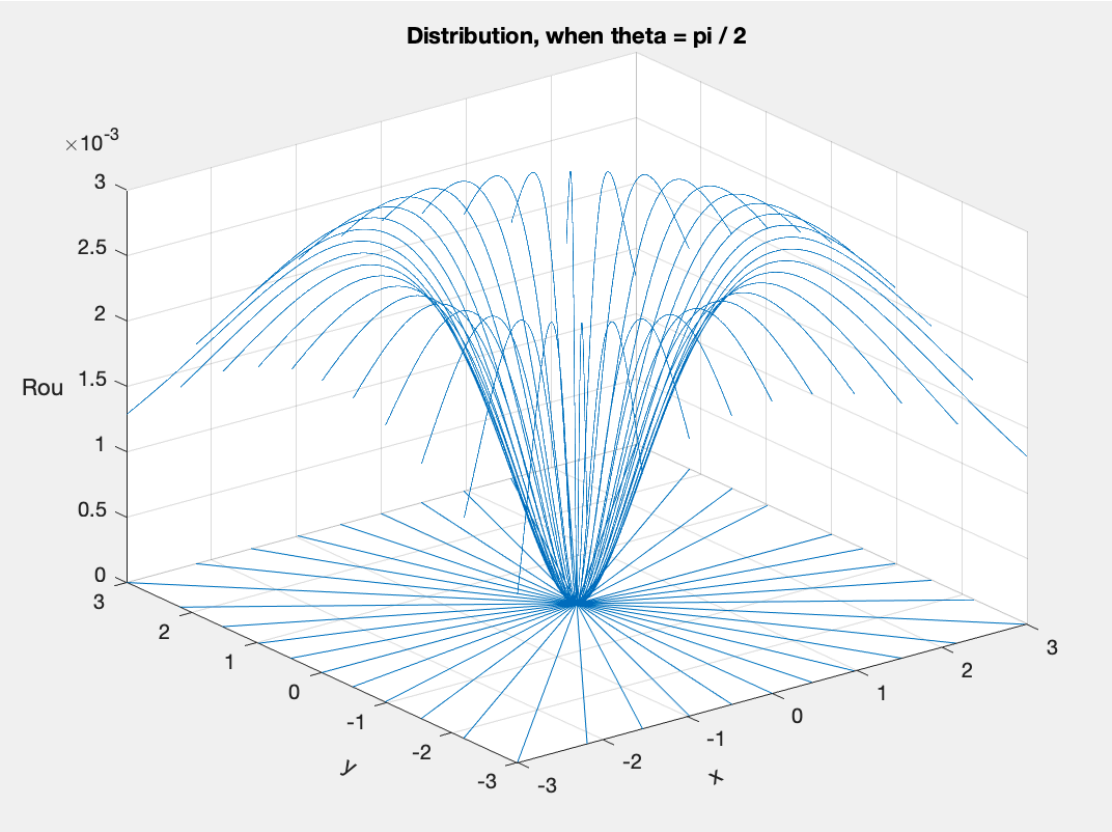

图 7.3: M=1,xy 平面图

很容易看到，这一波函数分布完全符合 211 和 210 氢原子波函数的预期，在 M=0 时，xy 平面应该什么都没有。

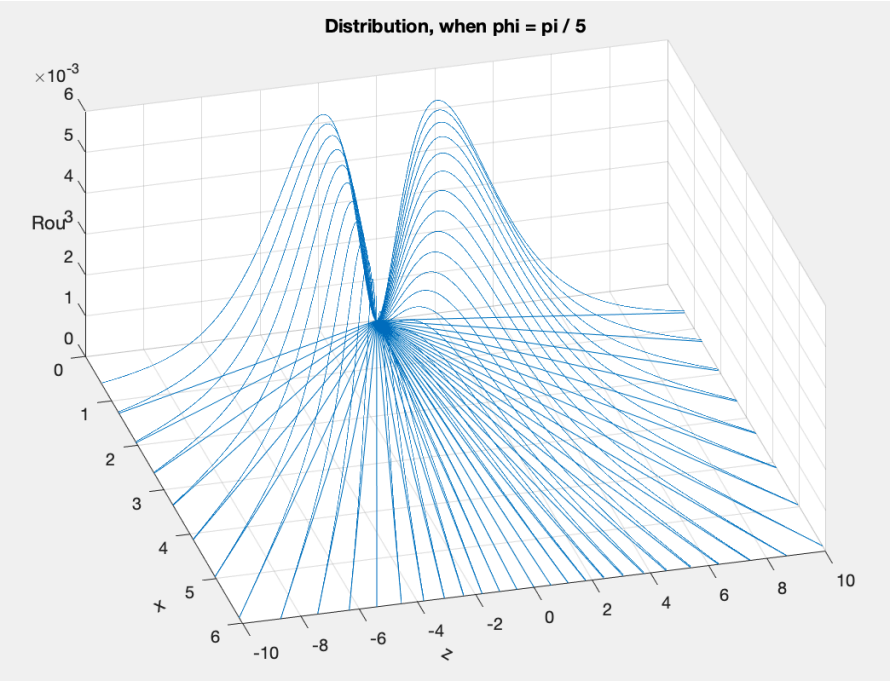

图 7.4: M=0,xz 平面图

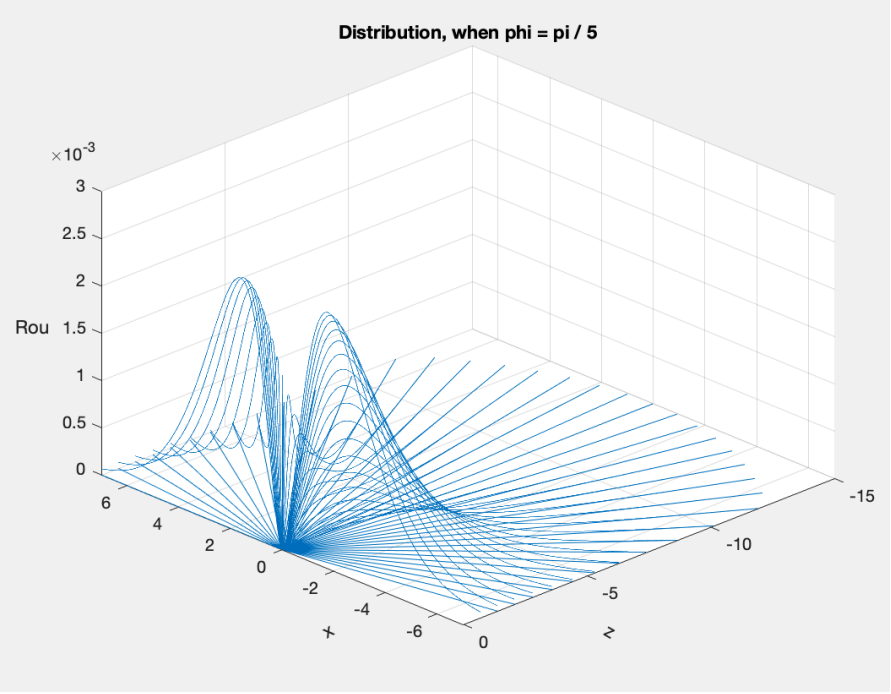

图 7.5: M=1,xz 平面图

M=1 和-1 分布相同，可以看到 M=0 和 M=1 只是 xz 轴交换，在空间分布图形上是完全一样的

7.4.4 第（4）问

l=1,m=1,n=3 时的能量为：-0.05555555568708451，得到的结果精度是 1E-9 满足要求。

我们可以大概看一下不同步长的情况：

步长为 0.1 时，E 大约为-0.05555565932443771

步长为 0.01 时，E 大约为-0.05555555568708451

步长为 0.001 时，E 大约为-0.05555555555099363

一般情况下，步长越小误差越小，但考虑时间和精度两个因素，0.01 作为步长比较合适