“计算物理学”第 1 次大作业
(补第 7 题)

李聪乔 1500011425

2019 年 6 月 18 日

目录

1 双精度浮点数转化

题目：编写两个程序，实现任意十进制数与其规范化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.

1.1 解题思路

本题只需实现十进制数至国际通用 IEEE 754 标准双精度浮点数编码进行转化即可. 首先熟悉一下转化规则：第一步应写出十进制数在二进制下的科学计数法表示，记为

$$a = a_0 \times 2^m.$$

其中 a_0 为二进制的实数，满足 $(1)_2 \le a_0 < (10)_2$. 对于转化后的 64 位浮点数，分为三部分分别表示数值的正负、a_0 以及 m，规则如下：

(i) 第 1 位标记十进制数的正、负. 其中 0 为正，1 为负.

(ii) 第 2–12 位（共 11 位数）所示二进制数的十进制表示等于 $1023 + m$. 显然该二进制数表示的十进制的范围是 $0 \sim 2047$，因此幂指数 m 的范围是 $-1023 \le m \le 1024$.

(iii) 第 13–64 位（共 52 位数）为 a_0 从第一位起的二进制字符串.

通过以上算法，很容易给出程序.

1.2 算法提纲

首先是十进制数转双精度浮点数编码. 程序请见 `01_convert_ieee/01_1_dec2ieee.py`，算法提纲如下：

(i) 首先讨论 $a = 0$ 的特殊情况，直接输出 64 位 0.

(ii) 判断 a 的正负.

(iii) 求 $\ln |a|$ 向下取整可知幂指数 m.

(iv) 幂指数 m_i 从 m 向下遍历至 $m - 51$，分别检查当前的十进制数是否大于 2^{m_i}，如大于则减去 2^{m_i}，并知道 a_0 的相应位上为 1；否则为 0.

(v) 将 a 的正负、幂指数 m 以及 a_0 按上述规则结合可得 64 为浮点数.

其次是从双精度浮点数到十进制数的转化. 程序请见 `01_convert_ieee/01_2_ieee2dec.py`，算法更为简单：

(i) 判断实数 a 正负，可知二进制编码第 1 位.

(ii) 通过编码 2–12 位二进制数求出 m.

(iii) 通过编码 13–64 位，计算相应位数的十进制数 $d_i \times 2^{m_i}$，并加和.

(iv) 最后考虑整体正负号即得到 a.

1.3 结果与讨论

取以下例子进行验证. 首先是十进制数转双精度浮点数编码:

```
请输入任意十进制小数: 987.75
双精度浮点数结果为:  0100000010001111011011110000000000000000000000000000000000000000

请输入任意十进制小数: -1234.5678
双精度浮点数结果为:  1100000010011001101001010010001010110110101011100111110101010110

请输入任意十进制小数: 0
双精度浮点数结果为:  0000000000000000000000000000000000000000000000000000000000000000
```

其次是双精度浮点数编码转十进制数, 用上面的输出作为输入测试结果:

```
请输入任意 64 位双精度浮点编码:
    0100000010001111011011110000000000000000000000000000000000000000
转化为十进制数的结果为:  987.75

请输入任意 64 位双精度浮点编码:
    1100000010011001101001010010001010110110101011100111110101010110
转化为十进制数的结果为:  -1234.5677999999998

请输入任意 64 位双精度浮点编码:
    0000000000000000000000000000000000000000000000000000000000000000
转化为十进制数的结果为:  0
```

可以验证, 十进制转双精度浮点数结果是正确的. 然而, 在双精度浮点数转十进制时, 发现对于 $a = -1234.5678$ 的浮点数编码, 转回后却得到 $a' = -1234.5677999999998$, 此处产生微小误差. 容易分析出这是由双精度浮点数的精度限制的. 一个验证方法是将 a' 再次作为输入, 计算出双精度浮点数, 可得:

```
请输入任意十进制小数: -1234.5677999999998
双精度浮点数结果为:  1100000010011001101001010010001010110110101011100111110101010110
```

发现与 $a = -1234.5678$ 的转化结果完全相同, 说明双精度浮点数已无法区分这两个极其接近的数.

1.4 源代码

首先是十进制转双精度浮点编码, 源文件: `01_convert_ieee/01_1_dec2ieee.py`:

```
# -*- coding: utf-8 -*-
import numpy as np

def dec2ieee(a_dec):
    a_dec = float(a_dec)
```

```
    # 讨论特殊a=0情况
    if a_dec == 0:
        return '0'*64

    # 确定正负，幂次
    a = a_dec if a_dec>0 else -a_dec
    n = np.int(np.log2(a))
    a_bin = [0 for i in range(64)]
    a_bin[0] = 0 if a_dec>0 else 1

    # 确定13-64位，即二进制下的科学计数法
    digit = 12
    for i in range(n,n-52,-1):
        if a >= 2**i:
            a -= 2**i
            a_bin[digit] = 1
        digit = digit+1

    # 确定2-12位，即幂次
    exp = 1023 + n
    if exp > 2047 or exp < 0:
        print("出界了您呐!")
    for i in range(1,12):
        if exp >= 2**(11-i):
            exp -= 2**(11-i)
            a_bin[i] = 1

    # 转化为字符串输出
    a_bin_str = ''
    for i in range(64):
        a_bin_str += '%s'%a_bin[i]

    return a_bin_str

a_dec = input('请输入任意十进制小数: ')
print('双精度浮点数结果为: ', dec2ieee(a_dec))
```

其次是双精度浮点编码转十进制，源文件：`01_convert_ieee/01_1_ieee2dec.py`：

```
# -*- coding: utf-8 -*-
import numpy as np

def ieee2dec(a_bin_str):
    # 错误提示
    if len(a_bin_str) != 64:
        print('输入的不是 64 位啊亲! ')
```

```
        return 'error'
    a_bin = [0 for i in range(64)]
    for i in range(64):
        a_bin[i] = int(a_bin_str[i])
        if a_bin[i]!= 0 and a_bin[i]!= 1:
            print('只能输入 0 或 1 啊亲! ')
            return 'error'

    # 确定幂次
    exp = -1023
    for i in range(1,12):
        if a_bin[i] == 1:
            exp += 2**(11-i)

    # 将每一个小数位转化为十进制相加
    a_dec = 0
    for i in range(12,64):
        if a_bin[i] == 1:
            a_dec += 2**(12+exp-i)

    # 考察正负, 得到结果
    minusQ = -1 if a_bin[0]==1 else 1
    a_dec *= minusQ
    return a_dec

a_bin_str = input('请输入任意 64 位双精度浮点编码: ')
print('转化为十进制数的结果为: ', ieee2dec(a_bin_str))
```

2 非标准权函数的高斯求积

题目：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和求积系数和节点：

$$\int_0^1 \sqrt{x} f(x)\,\mathrm{d}x \approx A_0 f(x_0) + A_1 f(x_1);$$
$$\int_0^1 \sqrt{x} f(x)\,\mathrm{d}x \approx A_0 f(x_0) + A_1 f(x_1) + A_2 f(x_2);$$
$$\int_{-1}^1 (1+x^2) f(x)\,\mathrm{d}x \approx A_0 f(x_0) + A_1 f(x_1).$$

2.1 解题思路

本题根据高斯求积的一般理论首先求出 n 个节点 x_i，即：带权高斯积分公式的 n 个节点 x_i 恰好是同样的权函数下 n 次正交多项式的 n 个根. 本例使用格拉姆-施密特正交化求解 n 次正交多项式，再利用牛顿法求根，即为节点 x_i. 由此，利用幂多项式可列出线性方程组，通过列主元 LU 分解与高斯消元可最终求解方程的系数 A_i.

2.2 算法提纲

首先，作者编写了矩阵“类”方便统一管理矩阵运算，命名为 `class matrix`，包含了矩阵的诸多常规操作，详细介绍请见附录 A.

程序的整体算法思路如下：

(i) 根据指定小问的权函数 $\rho(x)$ 与节点数 n，手动计算出 $2n-1$ 阶及以内的幂多项式的指定上下限的积分值，即

$$b_i = \int_a^b \rho(x) x^i\,\mathrm{d}x\,, \quad \text{其中} x_i = 0,\, 1,\, \cdots,\, 2n-1.$$

存之为向量 $\mathbf{b}$.

(ii) 利用格拉姆-施密特正交化构造 n 次正交多项式，其具体表达式为

$$\phi(x) = 1, \quad \phi_k(x) = x^{k-1} - \sum_{j=1}^{k-1} \frac{\langle x^{k-1},\, \phi_j(x)\rangle}{\langle \phi_j(x),\, \phi_j(x)\rangle}\, \phi_j(x),\ (k = 2, 3, \cdots).$$

容易利用递归的思路建立算法，计算出 n 次正交多项式 $\phi_n(x)$. 详情请见代码注释.

(iii) 使用牛顿法求 $\phi_n(x) = 0$ 的根. 显然，正交多项式 n 个根都位于 $[0,\, 1]$ 区间内. 单次寻根的伪代码是

```
x1, x2 初始化
while (|func(x1)| > 0.00000001 and |x1-x2| > 0.00000001) # 还不够接近零点，继续找
    s = x2 - func(x2) / (func(x2)-func(x1)) * (x2-x1)
```

```
        x1 = x2
        x2 = s
    end while
```

以上算法运行一次即可找到一个零点. 由于题目中至多只要处理 $n = 3$ 的多项式，且考虑到 x_1, x_2 初始化对寻根的位置影响重大，故我们只用牛顿法找最左右两个根，对于 $n = 3$ 的情形我们利用韦达定理直接计算中间的根. 由此，可得到多项式的根列表，也即题目要求的节点 $x_i, i = 0, \cdots, n-1$.

(iv) 求解 $A_0 f(x_0^i) + \cdots + A_n f(x_n^i) = b^i$ $(i = 0, \cdots, n-1)$ 这个 n 元线性方程组，利用主元 LU 分解法及上、下三角方程组解法可以最终求得系数 A_i. 方法及伪代码见讲义第二章内容.

(v) 输出节点列表 x_i 与系数列表 A_i.

事后，本例还设计了验证高斯求积是否正确的方案. 我们使用 `scipy.integrate.quad` 函数来获得数值解，用于比较高斯求积的结果. 本例只对第 (2) 问进行了验证，验证结果请见下节.

2.3 结果与讨论

运行程序 `02_guass_int/02_guass_int.py`，得到如下输出结果:

```
权函数:  sqrt(x) ;  N = 2
节点坐标 x_i:  [0.28994919792568774, 0.8211619131854476]
对应系数 A_i:  [0.27755599823107985, 0.3891106684355868]



权函数:  sqrt(x) ;  N = 3
节点坐标 x_i:  [0.16471028689646192, 0.5498684992165273, 0.9008058292716292]
对应系数 A_i:  [0.12578267432883045, 0.30760236768197463, 0.23328162465586158]



权函数:  1+x^2 ;  N = 2
节点坐标 x_i:  [-0.632455532033676, 0.632455532033676]
对应系数 A_i:  [1.3333333333333333, 1.3333333333333333]



对 权函数:  sqrt(x) ;  N = 3 的验证
我的函数: sin(x)^3
--> scipy积分值:    0.15683748981758447
    Gauss求积结果:  0.15679357794774212
    相对误差:   -0.027998324822348233 %
```

验证结果时我们使用的测试函数为 $f(x) = \sin^3(x)$，可见高斯求积与真实积分在该情形下值较为接近，精确度较高. 读者可通过改变 `myfunc(x)` 函数的形式自主设计 $f(x)$，检查计算结果的正确性.

2.4 源代码

关于本例中调用的简易矩阵库，请参见 `mymatrixlib/matrix.py` 文件，源代码请见附录 A.
本例的源文件请见： `02_guass_int/02_guass_int.py`

```
# -*- coding: utf-8 -*-
import sys
sys.path.append('..') # 为导入上级目录的mymatrixlib库
from mymatrixlib.matrix import matrix
from mymatrixlib.matrix import matrix_multiply

import numpy as np  # 只用numpy里的sprt, sin, cos简单函数啊喵
def specify_wfunc(wfunc='sqrt(x)', node=2):
    # 通过给定的权函数 (x) 与 node 的值，直接给出 x^p (x) dx 的值作为求解的 input
    node += 1
    if wfunc == 'sqrt(x)':
        b = [2/(2*i +1) for i in range(2*node-2)]
    elif wfunc == '1+x^2':
        b = [2/(2*i + 1) + 2/(2*i + 3) for i in range(2*node-2)]
    return b

def poly_eval(L, x):
    # 求多项式值
    y = 0
    for i in range(len(L)):
        y = y + L[i] * x ** i
    return y
def ortho_poly(node, b):
    # 利用格拉姆-施密特正交化求解多项式值
    ortho_coef = [0 for i in range(node-1)]
    ortho_coef.append(1)
    for i in range(node-1):
        for j in range(i+1,node):
            a = 0 # 这是要计算的分母<x^i, _j(x)>
            c = 0 # 这是要计算的分子<_j(x), _j(x)>
            for l in range(j):
                a = a + b[l+node-1] * ortho_poly(j, b)[l]
            for l in range(j):
                for m in range(l+1):
                    c = c + b[l] * ortho_poly(j, b)[m] * ortho_poly(j, b)[l-m]
            for l in range(j, 2*j-1):
                for m in range(l+1-j, j):
                    c = c + b[l] * ortho_poly(j, b)[m] * ortho_poly(j, b)[l-m]
            ortho_coef[i] = ortho_coef[i] - a / c * ortho_poly(j, b)[i] #
    正交多项式的各项系数
    return ortho_coef
```

```
def newton(L):
    # 牛顿法求根，区间[0,1]，由于只需解决题目只对根个数<=3有效
    nzeros = len(L) - 1
    Lx = []
    if nzeros >= 1 : # 找第一个根
        x1 = -1
        x2 = -0.85
        while abs(poly_eval(L, x2)) > 0.00000001 or abs(x2-x1) > 0.00000001: #
    是否足够接近零点
            s = x2 - poly_eval(L, x2) / (poly_eval(L, x2) - poly_eval(L, x1)) * (x2-x1)
            x1 = x2
            x2 = s
        Lx.append(x2)
    if nzeros >= 2: # 找第二个根
        x1 = 1
        x2 = 0.85
        while abs(poly_eval(L, x2)) > 0.00000001 or abs(x2-x1) > 0.00000001: #
    是否足够接近零点
            s = x2 - poly_eval(L, x2) / (poly_eval(L, x2) - poly_eval(L, x1)) * (x2-x1)
            x1 = x2
            x2 = s
        Lx.append(x2)
    if nzeros == 3: # 第三个根用韦达定理找啦，更多的就不再本例里实现了. .
        Lx.insert(1, -L[nzeros-1] / L[nzeros]-Lx[0]-Lx[-1])

    return Lx

def func_max(L, func): # 返回最大值点
    maxvalue = func(L[0])
    s = 0
    for i in range(len(L)):
        if func(L[i]) > maxvalue:
            maxvalue = func(L[i])
            s = i
    maxvalue = L[s]
    return maxvalue

def up_solve(A,b): # 解上三角方程组
    x = []
    for i in range(A.nrow):
        x.append(0)
    for i in range(A.nrow-1, -1, -1):
        x[i] = b[i]
        for j in range(A.nrow-1, i, -1):
```

```
            x[i] = x[i] - A.row[i][j] * x[j]
        x[i] = x[i] / A.row[i][i]
    return x

def low_solve(A,b): # 解下三角方程组
    x = []
    for i in range(A.nrow):
        x.append(0)
    for i in range(A.nrow):
        x[i] = b[i]
        for j in range(i):
            x[i] = x[i] - A.row[i][j] * x[j]
        x[i] = x[i] / A.row[i][i]
    return x

def LU_decomp(M, B): # 线性方程组的列选主元LU分解解法
    A = M.copy()
    p = range(A.nrow)
    for k in range(A.nrow-1):
        # 列主元
        L = A.column[k][k:]
        s = k + L.index(func_max(L, abs))
        if s != k:
            exc = p[s]
            p[s] = p[k]
            p[k] = exc
            exc = A.row[s][:]
            A.row[s] = A.row[k][:]
            A.row[k] = exc[:]
        # LU分解
        for i in range(k+1, A.nrow):
            A.row[i][k] = A.row[i][k] / A.row[k][k]
            for j in range(k+1, A.nrow):
                A.row[i][j] = A.row[i][j] - A.row[i][k] * A.row[k][j]
    b0 = []
    for i in range(A.nrow):
        b0.append(B[p[i]])
    A1 = matrix.copy(A)
    for i in range(A1.nrow):
        A1.row[i][i] = 1
    # 解方程
    b1 = low_solve(A1, b0)
    x = up_solve(A, b1)
    return x
```

```
def cal_gauss_int(wfunc, node=2):
    node += 1
    if wfunc == 'sqrt(x)':
        b = [2/(2*i + 3) for i in range(2*node-2)]
    elif wfunc == '1+x^2':
        b = [8/3]
        for i in range(node-1):
            b.append(0)
            b.append(2/(2*i + 3) + 2/(2*i + 5))

    opoly = ortho_poly(node, b) # 求正交多项式
    Lx = newton(opoly) # 正交多项式求根, 得到节点表Lx
    MA = [[Lx[j]**i for j in range(len(Lx))] for i in range(len(Lx))]  # 利用节点表制成矩阵
    bA = b[:len(Lx)]
    LA = LU_decomp (matrix(MA), bA) # 解出求积系数表
    print('权函数: ', wfunc, ';  N =', node-1)
    print('节点坐标 x_i: ', Lx)
    print('对应系数 A_i: ', LA)
    print('\n')
    return Lx, LA

Lx_1, LA_1 = cal_gauss_int(wfunc='sqrt(x)', node=2)
Lx_2, LA_2 = cal_gauss_int(wfunc='sqrt(x)', node=3)
Lx_3, LA_3 = cal_gauss_int(wfunc='1+x^2', node=2)

from scipy import integrate # 使用scipy只是为了检验下答案是否正确
def myfunc(x):
    return np.sin(x)**3
def myfunc_w(x):
    return np.sqrt(x) * myfunc(x)
print('对 权函数:  sqrt(x) ;  N = 3 的验证')
print('我的函数: sin(x)^3 ')
true_value  = integrate.quad(myfunc_w, 0, 1)[0]
gauss_value = sum([LA_2[i] * myfunc(Lx_2[i]) for i in range(len(Lx_2))])
print('--> scipy积分值:   ', true_value)
print('    Gauss求积结果: ', gauss_value)
print('    相对误差:  ', (gauss_value - true_value)/true_value*100, "%")
```

3 核衰变问题：常微分方程初值问题的有限差分法

题目：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 $N_{\mathrm{A}}(t)$ 和 $N_{\mathrm{B}}(t)$. 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

$$\begin{cases} \dfrac{\mathrm{d}N_{\mathrm{A}}}{\mathrm{d}t} = -\dfrac{N_{\mathrm{A}}}{\tau_{\mathrm{A}}}, \\ \dfrac{\mathrm{d}N_{\mathrm{B}}}{\mathrm{d}t} = \dfrac{N_{\mathrm{A}}}{\tau_{\mathrm{A}}} - \dfrac{N_{\mathrm{B}}}{\tau_{\mathrm{B}}}, \end{cases}$$

其中，τ_{A} 和 τ_{B} 为衰变时间常数. 设时间初始点位 $t_{\mathrm{i}} = 0$ 时 $N_{\mathrm{A}}(t_{\mathrm{i}}) = N_{\mathrm{B}}(t_{\mathrm{i}}) = 1$.

(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 $\tau_{\mathrm{A}} = 1$ 秒，分别讨论 $\tau_{\mathrm{B}} = 0.1$ 秒，1 秒，10 秒三种情况下的短期和长期衰变行为. 选取 $\tau_{\mathrm{B}} = 10$ 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.

3.1 解析计算

首先对上述微分方程组进行解析求解. 对给定初值 $N_{\mathrm{A}}(0) = N_{\mathrm{B}}(0) = 1$，容易给出结果为（详细过程省略，因不是考察重点）

$$\begin{cases} N_{\mathrm{A}}(t) = \mathrm{e}^{-t/\tau_{\mathrm{A}}}, \\ N_{\mathrm{B}}(t) = \mathrm{e}^{-t/\tau_{\mathrm{B}}} + \dfrac{\tau_{\mathrm{B}}}{\tau_{\mathrm{A}} - \tau_{\mathrm{B}}}\left(\mathrm{e}^{-t/\tau_{\mathrm{A}}} - \mathrm{e}^{-t/\tau_{\mathrm{B}}}\right). \end{cases}$$

3.2 解题思路

本例最简洁的数值求解做法是利用有限差分方法求解该常微分方程的初值问题. 一个普适的常微分方程给出函数向量 $\mathbf{y}$ 关于自变量 x 的方程，这里 $\mathbf{y} = (y_1, y_2, \cdots, y_n)$. 方程表述为：

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}x} = f(x, \mathbf{y}), \quad \text{初始条件 } \mathbf{y}(0) = \mathbf{y}_0. \tag{1}$$

对于本例，考虑 $\mathbf{y}(t) = (N_{\mathrm{A}}(t), N_{\mathrm{B}}(t))$，则容易发现题目中给出的微分方程组正是 (1) 式表达的形式.

对具有普适性的 (1) 式，数值求解 $\mathbf{y}(x)$ 的方法是：引入 $[0, x]$ 上的有限个离散点 $0 = x_0 < x_1 < x_2 < \cdots < x_N = x$，相邻点为等间距定义步长为 $h = (x - 0)/N$，则可给出数值微商公式

$$\left.\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}x}\right|_{x=x_i} = \frac{\mathbf{y}(x_{i+1}) - \mathbf{y}(x_i)}{h}.$$

由此可利用 $\mathbf{y}(x_i)$ 推出 $\mathbf{y}(x_{i+1}) = \mathbf{y}(x_i) + f(x, \mathbf{y}(x_i))\, h$，利用递归法可最终求得 $\mathbf{y}(x)$.

3.3 算法提纲

如上分析，有限差分法求常微分方程的伪代码为：

```
y 的初始值
for t 遍历以tstep为间隔的时间序列:
    y = y + func(x, y) * tstep  # y随时间演化的更新
    y 存入 y序列中
end for
```

按照题目要求，分别另初始参数 $\tau_{\mathrm{B}} = 0.1, 1, 10$ 秒（不妨设步长 $\Delta t = 0.01$ 秒）与 $\Delta t = 0.2, 0.1, 0.05$ 秒进行求解，比较结果时采用 `matplotlib.pyplot.plot` 函数作图.

3.4 结果与讨论

运行程序 `03_decay/03_decay.py`，得到如下输出结果：

```
tauB=0.1 计算完毕
tauB=1.0 计算完毕
tauB=10.0 计算完毕
tstep=0.20 计算完毕
tstep=0.10 计算完毕
tstep=0.05 计算完毕
```

说明对以上六种参数值的微分方程数值计算已经完毕. 同时, 程序将输出两个图, 如图 1, 2 所示.

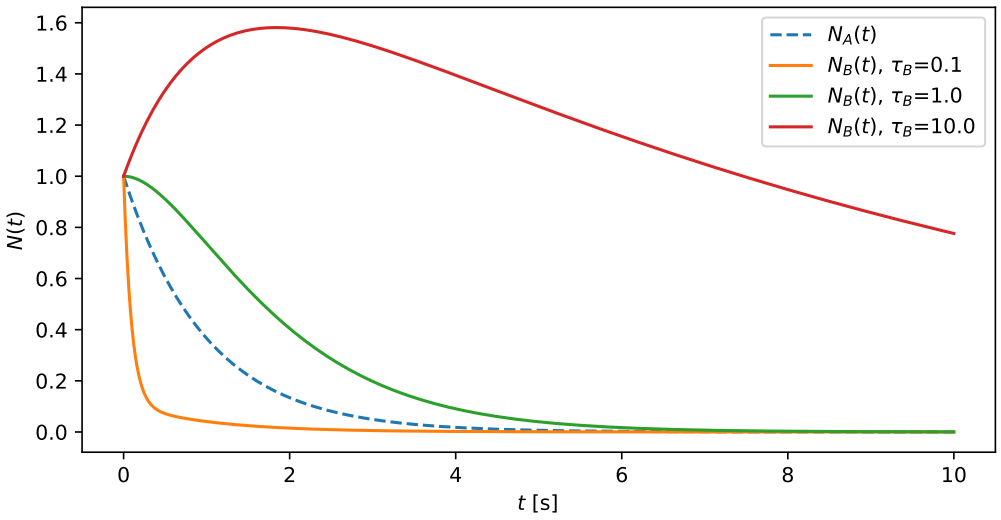

图 1: 改变 $\tau_{\mathrm{B}} = 0.1, 1, 10$ 秒所得的求解结果，此时 $\tau_{\mathrm{A}} = 1$ 秒，Δt 设为 0.01 秒.

对于 τ_{B} 对 B 粒子衰变特征的影响，可分三方面总结：

(i) 对于 τ_{B} 较小的短衰变周期情况，虽然 A 粒子会生成 B 粒子使之数目增加，但其影响极其有限，B 粒子仍会很快地衰减为 0；

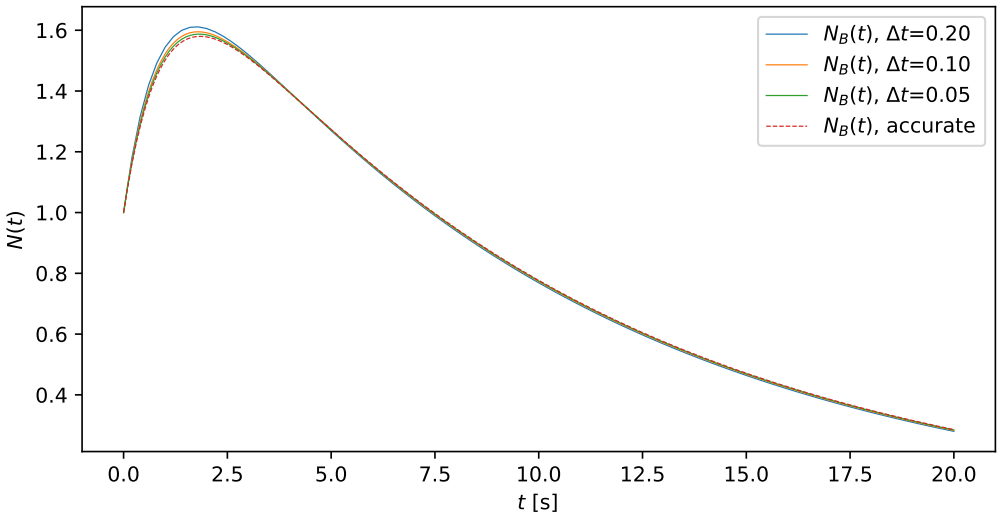

图 2: 改变 $\Delta t = 0.2, 0.1, 0.05$ 秒所得的求解结果，此时 $\tau_{\mathrm{A}} = 1$ 秒，$\tau_{\mathrm{B}} = 10$ 秒. 同时也根据解析式计算了精确结果，如图中虚线所示.

(ii) 对于 $\tau_{\mathrm{B}} = \tau_{\mathrm{A}}$ 的情形，由于 B 粒子会一直获得补充，其实际衰变衰变速率为慢于 A；在初始时刻，A 衰变为 B 的速率和 B 自身衰变速率相等，故 B 的减小率瞬时为 0，这一点也可从图 1 看出；

(iii) 对于 τ_{B} 很大的情形，A 会更快地补充给 B，使得 B 粒子数目先增大；而后 A 粒子数目减小到一定程度，不能给 B 足够的补充时，B 粒子自身的衰变开始占优，粒子数开始减小. 故 $N_{\mathrm{B}}(t)$ 曲线呈现先增大后衰减的形式.

对改变不同的步长 Δt 的图 2 中，$\Delta t = 0.2$ 相应的蓝色曲线在图片放大时能看到清晰锯齿，说明 $N_{\mathrm{B}}(t)$ 求解结果显然并不够精确；而当 Δt 减小时，数值曲线越发接近解析解所给出的精确曲线（红色虚线），说明 $N_{\mathrm{B}}(t)$ 计算结果越发精确.

3.5 源代码

本例的源文件请见：`03_decay/03_d0ecay.py`

```
# -*- coding: utf-8 -*-
def func(y, x):
    # 指定 f(x,y)形式，当然本例中f与x (也即时间t)无关
    return [-y[0]/tauA, y[0]/tauA - y[1]/tauB]

def next_step(func, y, x, tstep):
    # y的下一步演化值
    return [y[i] + func(y, x)[i]*tstep for i in range(2)]
```

```
def calculator(tauA, tauB, tstep, T): # 封装求解过程
    y = [1., 1.] # 初始函数值(NA, NB)
    Lt = [i*tstep for i in range(int(T/tstep) + 1)] # 等步长的时间演化向量
    Ly = []
    Ly.append(y)
    for t in range(len(Lt)-1):
        y = next_step(func, y, t, tstep) # y在下一时间的演化结果
        Ly.append(y)
    return Lt, Ly

# 下面开始就具体参数值计算微分方程数值解
Ly_dict = {}
Lt_dict = {}
tauA = 1.
tauB = 1.
for tauB in [0.1, 1., 10.]: # 改变3种不同的tauB
    Lt, Ly_dict['tauB=%.1f'%tauB] = calculator(tauA=1., tauB=tauB, tstep=0.01, T=10.)
    print('tauB=%.1f 计算完毕'%tauB)
for tstep in [0.2, 0.1, 0.05]: # 改变3种不同的步长tstep
    Lt_dict['tstep=%.2f'%tstep], Ly_dict['tstep=%.2f'%tstep] = calculator(tauA=1.,
     tauB=10., tstep=tstep, T=20.)
    print('tstep=%.2f 计算完毕'%tstep)




########## 计算完啦, 下面来画图验证一下 ##########
from matplotlib import pyplot as plt # 只是为了画图啦
import numpy as np # 为了用常数e
plt.figure(figsize=(8,4))
plt.plot(Lt, [Ly_dict['tauB=0.1'][i][0] for i in range(len(Lt))], label=r'$N_A (t)$',
    linestyle='--')
for tauB in [0.1, 1., 10.]:
    plt.plot(Lt, [Ly_dict['tauB=%.1f'%tauB][i][1] for i in range(len(Lt))], label=r'$N_B
    (t)$, $\tau_B$=%.1f'%tauB)
plt.xlabel(r'$t$ [s]')
plt.ylabel(r'$N(t)$')
plt.legend(loc=1)
plt.savefig('03_tauB.pdf')
plt.show()

plt.figure(figsize=(8,4))
for tstep in [0.2, 0.1, 0.05]: # 改变3种不同的步长tstep
    plt.plot(Lt_dict['tstep=%.2f'%tstep], [Ly_dict['tstep=%.2f'%tstep][i][1] for i in
    range(len(Lt_dict['tstep=%.2f'%tstep]))],
```

```
            label=r'$N_B (t)$, $\Delta t$=%.2f'%tstep, linewidth=0.6)
def func_realB(t):
    return np.exp(-t/10)+10./(1-10)*(np.exp(-t/1) - np.exp(-t/10))
plt.plot(np.linspace(0.,20.,1000), func_realB(np.linspace(0.,20.,1000)), label=r'$N_B
    (t)$, accurate', linewidth=0.6, linestyle='--')
plt.xlabel(r'$t$ [s]')
plt.ylabel(r'$N(t)$')
plt.legend()
plt.savefig('03_tstep.pdf')
plt.show()
```

4 样条插值函数

题目：在飞机设计中，已知机翼下轮廓上数据如下，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 y 的值，并画出轮廓曲线.

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

4.1 解题思路

本例使用三次样条插值方法求解. 三次样条插值是指: 在相邻节点使用不同的三次多项式, 使节点处的二阶导数连续. “样条”源于早起工程绘图时使用一根铁条弯曲地穿过各个节点，木条的整体性即给出节点处二阶导数连续的条件. 实际求解时，还需增加两个边界条件方程，使个三次多项式系数可以唯一确定. 本例使用自然边界条件，即令两端节点的二阶导数为 0.

一般地，对 $n+1$ 个节点（n 段插值）的情形可化简为如下严格对角占优的三对角矩阵方程，可以利用“追赶法”快速求解. 注意在自然边界条件下取 $f_0''=f_n''=0$.

$$\begin{bmatrix} 2 & \lambda_1 & & & \\ \mu_2 & 2 & \lambda_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \mu_{n-2} & 2 & \lambda_{n-2} \\ & & & \mu_{n-1} & 2 \end{bmatrix} \begin{bmatrix} M_1 \\ M_2 \\ \vdots \\ M_{n-2} \\ M_{n-1} \end{bmatrix} = \begin{bmatrix} d_1-\mu_1 f_0'' \\ d_2 \\ \vdots \\ d_{n-2} \\ d_{n-1}-\lambda_{n-1}f_n'' \end{bmatrix}.$$

其中令 $h_i=x_{i+1}-x_i$，有

$$\mu_j=\frac{h_{j-1}}{h_{j-1}+h_j},\quad \lambda_j=\frac{h_j}{h_{j-1}+h_j},$$
$$d_j=\frac{6}{h_{j-1}+h_j}\left(-\frac{y_j-y_{j-1}}{h_{j-1}}+\frac{y_{j+1}-y_j}{h_j}\right).$$

利用“追赶法”求解上述矩阵方程可得到向量 $\mathbf{M}=[M_1,M_2,\cdots,M_{n-1}]^{\mathrm{T}}$. 再根据下式即可计算出各段的三次多项式

$$S_j(x)=\frac{(x_j-x)^3}{6h_{j-1}}M_{j-1}+\frac{(x-x_{j-1})^3}{6h_{j-1}}M_j+\left(y_{j-1}-\frac{M_{j-1}h_{j-1}^2}{6}\right)\frac{x_j-x}{h_{j-1}}+\left(y_j-\frac{M_jh_{j-1}^2}{6}\right)\frac{x-x_{j-1}}{h_{j-1}}.$$

4.2 算法提纲

按照上述思路，算法可分为三部分:

(i) 根据给出的 x_i, y_i $(i=0,\cdots,n+1)$ 利用上式转化为三对角方程组的形式.

(ii) 利用“追赶法”求解该方程，算法封装在 `thomas(a, b, c, f)` 函数中.

(iii) 计算每段的插值三次多项式 $S_j(x)$，根据采样向量计算相应的插值函数值，绘图.

此后，再利用 Python 的 `scipy` 包中自带的三次样条插值函数进行求解，将 Python 的结果与自己编写的结果做一对比.

4.3 结果与讨论

运行程序 `04_spline/04_spline.py`，得到如下输出结果，此为求得的插值函数在以 0.1 为间距的 x 采样下的插值结果：

```
x =  0.0         y = 0.0000000000
x =  0.1         y = 0.0440732390
x =  0.2         y = 0.0881192928
x =  0.3         y = 0.1321109764
x =  0.4         y = 0.1760211045
x =  0.5         y = 0.2198224922
x =  0.6         y = 0.2634879542
x =  0.7         y = 0.3069903053
x =  0.8         y = 0.3503023606
x =  0.9         y = 0.3933969348
(......中间部分较长，在此省略......)
x = 14.0         y = 1.0000000000
x = 14.1         y = 1.0195268693
x = 14.2         y = 1.0472782250
x = 14.3         y = 1.0835426457
x = 14.4         y = 1.1286087097
x = 14.5         y = 1.1827649957
x = 14.6         y = 1.2463000819
x = 14.7         y = 1.3195025470
x = 14.8         y = 1.4026609694
x = 14.9         y = 1.4960639276
```

同时输出求解得到的插值函数图，其中包含原数据、自主编写的三次样条插值结果与 Python 内置函数 `scipy.interpolate.interp1d` 所得插值结果，请见图 3. 从图中看，自主编写的三次样条插值曲线能够穿过各个节点，曲线整体平滑，且与 Python 内置函数所得结果符合的很好，说明本算例得到结果有效. 此外，发现在两侧节点处略微偏离 Python 内置函数所得曲线，猜测与选取的边界条件不同有关.

4.4 源代码

本例的源文件请见：`04_spline/04_spline.py`

```python
# -*- coding: utf-8 -*-


def thomas(a, b, c, f):
    # Thomas算法解三对角矩阵方程
    # a: -1对角线向量补首位，b: 主对角线向量，c: +1对角线向量补末位，f: 结果向量
    m = [a[i+1] / b[i] for i in range(len(f)-1)]
    # 追
    for i in range(len(f)-1):
        b[i+1] = b[i+1] - m[i]*c[i]
```

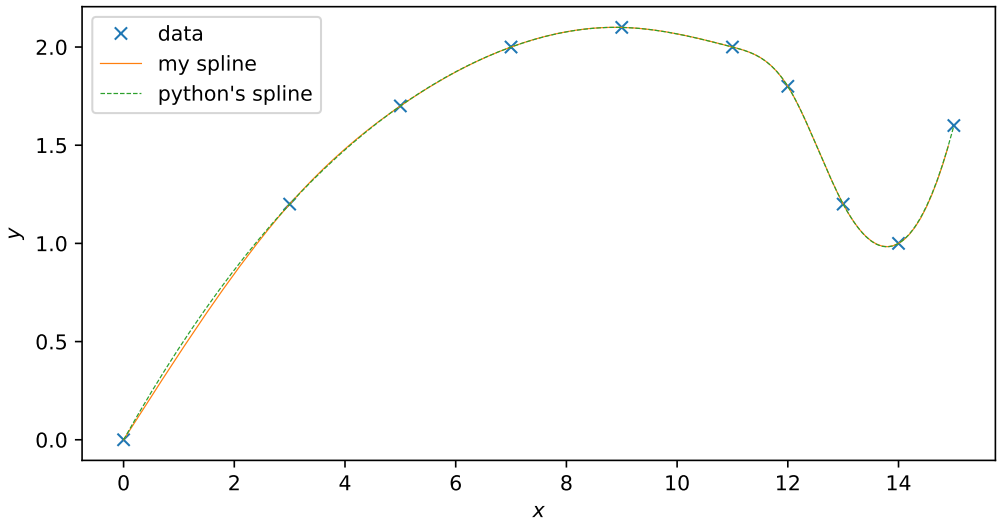

图 3: 自主编写的自然条件下的三次样条插值函数曲线，与 Python 内置函数所得插值曲线对比图.

```
        f[i+1] = f[i+1] - m[i]*f[i]
    x = [f[-1]/b[-1]]
    # 赶
    for i in range(len(f)-1, -1, -1):
        x.insert(0, (f[i] - c[i]*x[0]) / b[i])
    return x

def spline(x, y, Lx): # 计算插值函数在采样Lx上的值Ly
    # 计算N-2阶三对角矩阵
    h = [x[i+1] - x[i] for i in range(len(x)-1)]
    sh = [h[i+1] + h[i] for i in range(len(x)-2)]
    mu = [h[i]/sh[i] for i in range(len(x)-2)]
    lam = [1 - mu[i] for i in range(len(x)-2)]
    d = [6 * (y[i]/h[i]/sh[i] + y[i+2]/h[i+1]/sh[i] - y[i+1]/h[i]/h[i+1]) for i in
    range(len(x)-2)]
    b = [2 for i in range(len(x)-2)]

    # 追赶法解三对角方程组
    M = thomas(mu, b, lam, d)
    M.insert(0, 0)
    M.append(0)

    # 以指定步长遍历
    Ly = []
```

```
    hsqure6 = [h[i]**2 / 6 for i in range(len(h))]
    for i in range(len(Lx)):
        ii = 0
        for j in range(1, len(x)): # 找到相应分段
            if Lx[i] <= x[j]:
                ii = j - 1
                break
        h1 = (x[ii+1] - Lx[i]) / h[ii]
        h2 = (Lx[i] - x[ii]) / h[ii]
        # 计算插值函数值
        yy = M[ii] * h1**3 * hsqure6[ii] + M[ii+1] * h2**3 * hsqure6[ii] + (y[ii] - M[ii]
    * hsqure6[ii]) * h1 + (y[ii+1] - M[ii+1] * hsqure6[ii]) * h2
        Ly.append(yy)
    return Ly

# 题中所给参数
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]
Lx = [i*0.1 for i in range(int(15/0.1))]
Ly = spline(x, y, Lx)
# 逐点打印
print('样条插值函数所得结果: ')
for i in range(len(Lx)):
    print('x = %4.1f         y = %2.10f'%(Lx[i], Ly[i]))



########## 计算完啦, 下面来画图验证一下 ##########
from matplotlib import pyplot as plt # 只是为了画图啦
from scipy.interpolate import interp1d # 用python自带spline来检验是否正确
import numpy as np # 只是用来画图啦
plt.figure(figsize=(8,4))
plt.plot(x, y, 'x', label='data')
# 我的spline
plt.plot(Lx, Ly, label='my spline', linewidth=0.6)
# python自带spline
plt.plot(np.linspace(0,15,500), interp1d(x, y, kind='cubic')(np.linspace(0,15,500)),
    '--', label='python\'s spline', linewidth=0.6)
plt.xlabel(r'$x$'); plt.ylabel(r'$y$')
plt.legend()
plt.savefig('04_spline_result.pdf')
plt.show()
```

5 正定带状系数矩阵方程组解法

题目：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 $\mathbf{A}=\mathbf{LDL}^{\mathrm{T}}$，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 $a_{11}=a_{nn}=5$; $a_{ii}=6$ $(i=2,\cdots,n-1)$; $a_{i,i-1}=4$ $(i=2,\cdots,n)$; $a_{i,i-2}=1$ $(i=3,\cdots,n)$. 右端向量 $\mathbf{b}=(60,\underbrace{120,\cdots,120}_{n-2},60)^{\mathrm{T}}$. 请取 $n=100$ 和 10000 两个算例，展示你的结果.

5.1 解题思路

本算例先给出一般性的带宽为 $2m+1$ 的 n 阶带状正定矩阵 $\mathbf{A}$ 的求解过程. 首先，可以论证 $\mathbf{A}$ 可唯一且稳定地分解为 $\mathbf{A}=\mathbf{LDL}^{\mathrm{T}}$，其中 $\mathbf{L}$ 为下半带宽为 m 的带状阵，形式为

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

$\mathbf{D}$ 为对角矩阵，对角元素

$$d_{ii}=\frac{1}{l_{ii}},\quad i=1,\cdots,n.$$

由上式即可利用矩阵乘法导出 $\mathbf{L}$ 矩阵元的计算公式：

$$l_{ij}=a_{ij}-\sum_{k=\max\{1,\,i-m\}}^{j-1}\frac{l_{ik}l_{jk}}{l_{kk}},\quad i=1,\cdots,n.$$

同时，我们对列向量 $\mathbf{b}$ 也作分解：$\mathbf{b}=\mathbf{LD}\tilde{\mathbf{b}}$，直接利用矩阵和向量乘法，可得：

$$\tilde{b}_i=b_i-\sum_{j=\max\{1,\,i-m\}}^{i-1}\frac{l_{ij}\tilde{b}_j}{l_{jj}},\quad i=1,\cdots,n.$$

因此只需解等价方程组：$\mathbf{L}^{\mathrm{T}}\mathbf{x}=\tilde{\mathbf{b}}$. 利用上三角方程组的回代算法，可得

$$x_i=\left(\tilde{b}_i-\sum_{j=i+1}^{\min\{n,\,i+m\}}l_{ji}x_j\right)\Big/l_{ii},\quad i=1,\cdots,n,$$

即为原方程的解.

为使算法的空间复杂度减小，只需存储矩阵 $\mathbf{A}$ 的非零元即可，我们采用讲义第 2 章的记法，

即规定矩形数 $\mathbf{C}$，使 $\mathbf{C}$ 的每列是 $\mathbf{A}$ 的每条对角线，且左上角添 0：

$$\mathbf{C}=\begin{bmatrix} 0 & \cdots & 0 & a_{11} \\ 0 & \cdots & a_{21} & a_{22} \\ \vdots & \iddots & \vdots & \vdots \\ 0 & \cdots & a_{m-1,m-2} & a_{m-1,m-1} \\ a_{m,1} & \cdots & a_{m,m-1} & a_{m,m} \\ \vdots & & \vdots & \vdots \\ a_{n,1} & \cdots & a_{n,n-1} & a_{n,n} \end{bmatrix}$$

则有

$$c_{i,j-i+m+1}=a_{ij},\quad a_{ij}\text{ 为矩阵 }\mathbf{A}\text{ 的带上的元素.}$$

由此可将用于储存矩阵 $\mathbf{A}$ 的 $\mathcal{O}(n^2)$ 的空间复杂度降为 $\mathcal{O}(nm)$.

5.2 算法提纲

本例中的程序的数组角标规则采用数学规则，即从 1 开始编号. 于是可将上面的计算流程的每一步直接转化为伪代码：

(i) 初始化矩形数 $\mathbf{C}$ 及列向量 $\mathbf{b}$.

(ii) 进行 $\mathbf{A}=\mathbf{LDL}^{\mathrm{T}}$ 分解，计算 $\mathbf{L}$. 为节省储存空间可直接在 $\mathbf{C}$ 上进行操作得 $\mathbf{L}$ 的矩形数.

```
for j = 1, 2, ··· , n
    for i = j, j + 1, ··· , min{j + m, n}
        for k = max{1, i − m}, ··· , j − 1
            C_ij = C_ij − C_ik C_jk / C_kk
        end for
    end for
end for
```

上面伪代码里的 C_{ij} 实际是 $c_{i,\,j-i+m+1}$，它对应矩阵元 $\mathbf{A}(i,j)$. 这里出于方便阅读的考量进行了简化. 操作的 $\mathbf{C}$ 即为表示 $\mathbf{L}$ 的矩形数.

该算法的空间复杂度即为 $\mathbf{C}$ 的储存空间 $\mathcal{O}(nm)$；时间复杂度为分循环进行讨论：第一层循环为 $\mathcal{O}(n)$，第二、三层共 $\mathcal{O}(0+1+2+\cdots+m)\sim\mathcal{O}\left(\frac{m^2}{2}\right)$，每一次覆盖 C_{ij} 时需进行两次乘法与一次减法操作. 忽略常数系数可知总时间复杂度为 $\mathcal{O}(nm^2)$.

(iii) 分解 $\mathbf{b}$ 为 $\mathbf{b}=\mathbf{LD}\tilde{\mathbf{b}}$. 直接在向量 $\mathbf{b}$ 上操作即可转化为 $\tilde{\mathbf{b}}$, 伪代码为

```
for i = 1, 2, ··· , n
        for j = max{1, i − m}, ··· , i − 1
                b_i = b_i − C_ij b_j / C_jj
        end for
end for
```

这一步算法没有产生新的储存空间, 只不断覆盖 $\mathbf{C}$ 与 $\mathbf{b}$ 的数值, 故空间复杂度仍为 $\mathcal{O}(nm)$; 通过循环的方式亦可分析出时间复杂度为 $\mathcal{O}(nm)$.

(iv) 解等价方程组 $\mathbf{L}^{\mathrm{T}}\mathbf{x} = \tilde{\mathbf{b}}$. 需首先另 $\mathbf{x}$ 初始化为 $\mathbf{b}$, 随后在 $\mathbf{x}$ 上进行减法操作, 伪代码为

```
x = b    # 复制 b 至 x
for i = n, n − 1, ··· , 1    # 倒序遍历
        for j = i + 1, ··· , min{i + m, n}
                x_i = x_i − C_ji x_j
        end for
        x_i = x_i / C_ii    # 即最后除以 l_ii
end for
```

输出的向量 $\mathbf{x}$ 即为原方程的解向量. 这一步算法空间复杂度仍为 $\mathcal{O}(nm)$, 时间复杂度为 $\mathcal{O}(nm)$.

在计算完毕后, 本程序进一步对解进行了验证. 通过计算向量 $\mathbf{Ax} - \mathbf{b}$ 的模长是否足够小来验证解的正确性. 不过要注意的是, 为了能够在程序最后仍保留初始的 $\mathbf{A}$ 与 $\mathbf{b}$, 必须在刚开始初始化后进行一次拷贝. 如此一来, 若想验证解的正确性则需要额外的储存空间, 但单纯地求解 $\mathbf{x}$ 是不需要的.

我们还针对解向量 $\mathbf{x}$ 绘制了 $x_i - i$ 的折线以实现其可视化, 请见下节的讨论部分.

5.3 结果与讨论

运行程序 `05_band_mat/05_band_mat.py`, 首先需要输入矩阵阶数 n 的值, 输入 100 后, 得到如下输出结果:

```
输入矩阵的阶数 n = 100
x =  [-475.24752475503067, 951.0891089159978, -1368.118811888838, 1786.9306930794878,
    -2148.1188118938844, 2512.277227737964, -2820.0000000176624, 3131.8811881389156,
    -3388.5148515076594, 3650.4950495298317, -3858.415841611368, 4072.871287158201,
    -4234.4554455762645, 4403.762376271489, -4521.386138649807, 4647.920792117154,
    -4723.960396079466, 4810.099009942682, -4846.930693112743, 4895.049504995587,
    -4895.049504997151, 4907.5247525233735, -4873.069306980188, 4852.277227773534,
    -4785.742574309349, 4734.059405993573, -4637.821782232146, 4557.623762431012,
    -4434.059405996112, 4327.722772333385, -4179.20792084877, 4049.108910948206,
    -3878.019802037632, 3726.5346535229896, -3535.247524810223, 3364.752475305277,
    -3155.6435644140975, 2968.5148515426285, -2743.9603960968157, 2542.5742574826036,
```

```
    -2304.950495105938, 2091.6831683727637, -1843.3663366890237, 1620.5940594606611,
    -1363.9603960936192, 1134.0594059938417, -871.4851485672715, 636.8316832198525,
    -370.6930693575281, 133.6633663862419, 133.6633662880622, -370.6930692594405,
    636.831683121949, -871.4851484696438, 1134.0594058965805, -1363.9603959968154,
    1620.594059364405, -1843.3663365934062, 2091.683168277877, -2304.950495011874,
    2542.574257389453, -2743.9603960046697, 2968.5148514515777, -3155.643564324232,
    3364.7524752166873, -3535.2475247229945, 3726.534653437207, -3878.0198019533764,
    4049.1089108655556, -4179.207920767799, 4327.7227722541575, -4434.059405918685,
    4557.623762355435, -4637.821782158466, 4734.0594059218365, -4785.742574239611,
    4852.277227705848, -4873.069306914614, 4907.524752459974, -4895.049504935993,
    4895.049504936737, -4846.930693056272, 4810.099009888664, -4723.960396027978,
    4647.920792068281, -4521.386138603634, 4403.762376228097, -4234.45544553573,
    4072.8712871205917, -3858.415841576742, 3650.4950494982395, -3388.514851479141,
    3131.881188113503, -2819.999999995383, 2512.2772277188383, -2148.1188118779264,
    1786.9306930667078, -1368.118811879245, 951.0891089095986, -475.24752475182987]

计算完毕，下面验证结果是否正确
‖ A x - b ‖ = 2.372450164273537e-11
```

可以最终的残差非常小，几乎在 Python 的 double 精度的量级，故本算法正确.

再输入矩阵的阶数 $n = 10000$，得到输出结果为（只截取部分输出，全部输出已拷贝至文档 `05_band_mat/05_n10000_output.txt`）

```
输入矩阵的阶数 n = 10000
x =  [-49966.20121995007, 99932.40844023213, -149838.62766117818, 199744.86488312023,
    -249591.1261063903, 299437.41733132035, -349223.7445582423, 399010.11378748826,
    -448736.5310193901, 498463.00225428, -548129.5334924901, 597796.1307343517,
    -647402.7999801964, 697009.5472303557, -746556.3784851609, 796103.2997449436,
    -845590.3170100362, 895077.436280771, -944504.6635574809, 993932.0048404984,
    -1043299.4661301561, 1092667.0534267852, -1141974.7727307181, 1191282.6300422868,
    -1240530.6313618242,
(......中间部分较长，在此省略......)

计算完毕，下面验证结果是否正确
‖ A x - b ‖ = 2.3601986144478167e-06
```

亦可验证算法正确.

下面讨论本算法时空复杂度. 结合上一节分步的讨论，可知空间复杂度仅为储存的 $\mathbf{C}$, $\mathbf{b}$ 与 $\mathbf{x}$ 的空间，为 $\mathcal{O}(nm)$. 时间复杂度主要贡献是第一步 $\mathbf{A} = \mathbf{LDL}^{\mathrm{T}}$ 的分解过程，总体时间复杂度为 $\mathcal{O}(nm^2)$.

在实际问题中，一般带宽 $2m+1$ 与矩阵的形式有关，是固定的常量；而阶数 n 会因为系统自由度的增大而增大。因此我们更多要处理的是 m 固定，n 很大的矩阵方程组求解问题. 由以上分析知，其时空复杂度都仅为 $\mathcal{O}(n)$，可以看出这一算法的优越性.

另外，在最后计算 $\mathbf{Ax} - \mathbf{b}$ 来验证结果是否正确时，我们使用的矩阵乘法是针对“带状”矩阵的特征单独设计的，时间复杂度也仅为 $\mathcal{O}(nm)$；若选用一般的矩阵乘法计算方法，其复杂度将

为 $\mathcal{O}(n^2)$，计算速度将会显著变慢.

最后，随程序可同时得到 $n = 100$ 与 $n = 10000$ 两种情形下的 $x_i - i$ 的折线图，如图 4 所示.

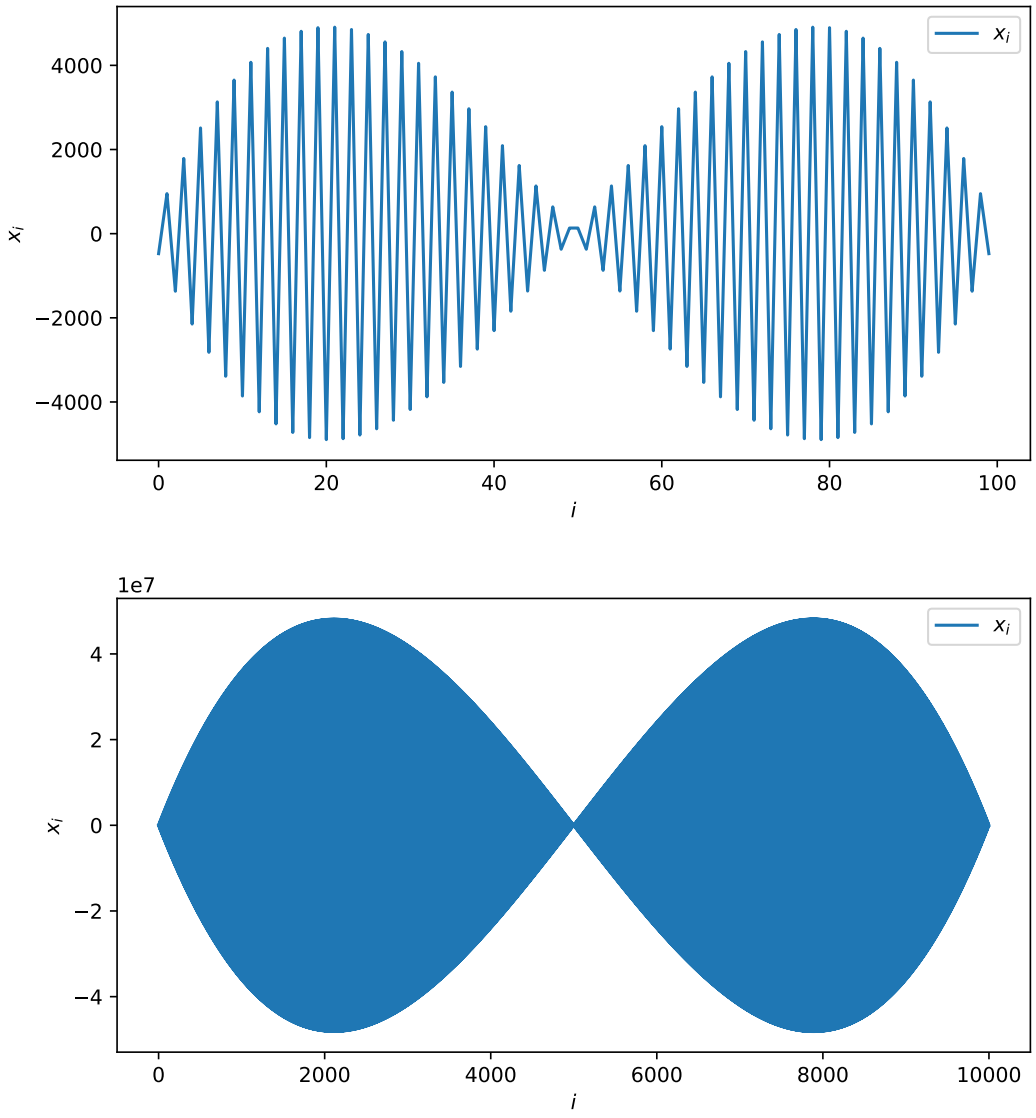

图 4: $n = 100$ (上) 与 $n = 10000$ (下) 两种情形下矩阵方程解 $\mathbf{x}$ 的 $x_i - i$ 的折线图

5.4 源代码

本例的源文件请见：`05_band_mat/05_band_mat.py`.

```
# -*- coding: utf-8 -*-

def convert_diag(C, op='sym'): # 将矩形数 C 转化为带状矩阵, 可选
    上对角、下对角、对称矩阵三个选项
    A = [[0 for i in range(n)] for i in range(n)]
    for j in range(n):
        for i in range(j, min(j+m+1, n)):
            if op=='low' or op=='sym':
                A[i][j] = C[i+1][j-i+m+1]
            if op=='up' or op=='sym':
                A[j][i] = C[i+1][j-i+m+1]
    return matrix(A)

def initialize(): # 初始化矩形数 C 和向量 b
    C = [[0 for i in range(m+2)] for i in range(n+1)]
    for i in range(1, n+1):
        C[i][m+1] = 6.
    for i in range(2, n+1):
        C[i][m] = 4.
    for i in range(3, n+1):
        C[i][m-1] = 1.
    C[1][m+1] = 5.
    C[n][m+1] = 5.

    b = [120. for i in range(n+1)]
    b[1] = 60; b[n] = 60
    return C, b

n = int(input('输入矩阵的阶数 n = ')) # 矩阵阶数
m = 2  # 带宽 2m+1
C, b = initialize()
# 拷贝一份原始的C和b (注意!! 实际算法中不需要这一步哒, 只是为了最后能检验下结果这里才这样做)
b0 = [bi for bi in b]
C0 = [[Cij for Cij in Ci] for Ci in C]

for j in range(1, n+1): # j=1,..,n
    for i in range(j, min(j+m+1, n+1)): # i=j,j+1,..,j+m or n
        for k in range(max(1, i-m), j): # k=r,..,j-1
            C[i][j-i+m+1] -= C[i][k-i+m+1] * C[j][k-j+m+1] / C[k][m+1] # 也就是 a_ij =
    a_ij - Σ_r^(j-1) l_ik l_jk/ l_kk

for i in range(1, n+1):  # i=1,..,n
    for j in range(max(1, i-m), i): # j=r,..,i-1
        b[i] -= C[i][j-i+m+1] * b[j] / C[j][m+1] # 也就是 b'_i = b_i - Σ_r^(i-1) l_ij
    b'_j/ l_jj
```

```
x = [bi for bi in b]  # 初始时 x=b'
for i in range(n, 0, -1): # i=n,n-1,..,1
    for j in range(i+1, min(i+m+1, n+1)): # j=i+1,..,i+m or n
        x[i] -= C[j][i-j+m+1] * x[j]
    x[i] /= C[i][m+1]
print('x = ', x[1:])


########## 计算完啦, 下面来画图验证一下 ##########
from matplotlib import pyplot as plt # 只是为了画图啦
plt.figure(figsize=(8,4))
plt.plot(range(n), x[1:], label=r'$x_i$')
plt.xlabel(r'$i$'); plt.ylabel(r'$x_i$')
plt.legend()
plt.savefig('06_n%d_xi.pdf'%n)
plt.show()

########## 最后验证一下结果的正确性 ##########
import numpy as np # 只用来计算向量的模长
print('\n计算完毕, 下面验证结果是否正确')
print('‖ A x - b ‖ = ', np.linalg.norm([sum([(C0[i][j-i+m+1] if i>j else
    C0[j][i-j+m+1])*x[j] for j in range(max(1,i-m), min(i+m+1,n+1))])-b0[i] for i in
    range(1, n+1)]))
```

6 共轭梯度法求解线性方程组

题目：设有稀疏矩阵 $\mathbf{A}$，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 $(i, n+1-i),\ (i=1,2,\cdots,n)$ 上的元素，值均为 $\frac{1}{2}$；其余元素均为 0. 例如，形式如下：

$$A=\begin{bmatrix} 3 & -1 & & \cdots & & \frac{1}{2} \\ -1 & 3 & -1 & \cdots & \frac{1}{2} & \\ & -1 & 3 & \cdots & & \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ & \frac{1}{2} & & \cdots & 3 & -1 \\ \frac{1}{2} & & & \cdots & -1 & 3 \end{bmatrix}$$

对 $b=(2.5, \overbrace{1.5, \cdots, 1.5}^{\frac{n}{2}-1}, 1.0, 1.0, \overbrace{1.5, \cdots, 1.5}^{\frac{n}{2}-1}, 2.5)^{\mathrm{T}}$，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.

6.1 解题思路

本题采用共轭梯度的算法求解方程组 $\mathbf{Ax}=\mathbf{b}$，它由最速下降法改进而来，以克服某些情况下收敛太慢的问题. 首先可以论证，该方程组求解问题可转化为变分求极值问题，也即等价为求解

$$\varphi(\mathbf{x})=\frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{Ax}-\mathbf{b}^{\mathrm{T}}\mathbf{x}$$

的最小值对应的 $\mathbf{x}$ 向量. 普适的迭代法要对迭代中每一步 $\mathbf{x}_k$ 查找搜索方向 $\mathbf{p}_k$，从而下一步向量有 $\mathbf{x}_{k+1}=\mathbf{x}_k+\alpha_k\mathbf{p}_k$. 定义每一步残差 $\mathbf{r_k}=\mathbf{b}-\mathbf{Ax}_k$, 可一般性地证明，若 $\mathbf{p}_k$ 已被确定，则使得 $\varphi(\mathbf{x}_{k+1}(\alpha_k))$ 取极小值的最佳的 α_k 为

$$\alpha_k=\frac{\mathbf{r}_k^{\mathrm{T}}\mathbf{p}_k}{\mathbf{r}_k^{\mathrm{T}}\mathbf{Ap}_k}.$$

因此不同方法间的差异主要在于 $\mathbf{p}_k$ 的选取. 优化的共轭梯度法取

$$\mathbf{p}_k=\mathbf{r}_k+\frac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{r}_{k-1}^{\mathrm{T}}\mathbf{r}_{k-1}}\mathbf{p}_{k-1}.$$

按照上述方法进行迭代，最终利用残差判据 $\|\mathbf{r}_k\|\le\varepsilon$ 作为迭代的判停标准，即可求得向量 $\mathbf{x}$.

另外，容易猜得，该方程组具有精确解 $\mathbf{x}_{acc}=(\overbrace{1,1,\cdots,1}^{n})$，故可由此验证数值结果的正确性.

6.2 算法提纲

算法的伪代码为

```
A, b 初始化
x 给定初始值
```

```
k = 0 # 迭代步数
r = b - Ax # 初始残差
while ||r|| > 0.000001:
      alpha 步长计算
      x = x + alpha p    # 更新解
      r1 = r             # 保留上一个残差向量
      r = r - alpha A p  # 计算新的残差向量
      p 搜索方向计算
      k += 1
end while
输出 x
```

程序请见 `06_conj_grad/06_conj_grad.py`. 另外需要说明，本算例给定的 $\mathbf{x}$ 向量初始值为向量初值为 $\mathbf{x}_{acc} = (5, 5, \cdots, 5)$，具有一定任意性.

6.3 结果与讨论

运行程序 `06_conj_grad/06_conj_grad.py`，首先需要输入矩阵阶数 n 的值，输入 100 后，得到如下输出结果:

```
输入矩阵的阶数 n = 100
第 16 步后得到结果:
x = [1.0000000000760108, 1.000000000177213, 1.0000000003369973, 1.000000000607658,
    1.0000000010766026, 1.000000001891998, 1.0000000033022738, 1.000000005711341,
    1.0000000097376847, 1.000000016219364, 1.000000025978268, 1.0000000388539012,
    1.0000000509760907, 1.0000000490264376, 1.0000000045947939, 0.9999999021954707,
    0.9999999852012714, 0.9999999852012714, 0.9999999852012714, 0.9999999852012714,
    0.9999999852012714, 0.9999999852012714, 0.9999999852012714, 0.9999999852012714,
    0.9999999852012714, 0.9999999852012714, 0.9999999852012714, 0.9999999852012714,
    0.9999999852012714, 0.9999999852012714, 0.9999999852012714, 0.9999999852012714,
    0.9999999852012714, 0.9999999852012714, 1.0000000267041718, 1.0000000377588607,
    1.000000032125072, 1.0000000226995625, 1.000000014622393, 1.0000000089444556,
    1.0000000053070015, 1.00000000309166, 1.000000001781819, 1.0000000010215917,
    1.000000000586389, 1.0000000003412817, 1.0000000002080935, 1.000000000143642,
    1.0000000001268565, 1.000000000152279, 1.000000000152279, 1.0000000001268563,
    1.000000000143642, 1.0000000002080935, 1.0000000003412817, 1.0000000005863887,
    1.0000000010215917, 1.000000001781819, 1.00000000309166, 1.0000000053070015,
    1.0000000089444556, 1.000000014622393, 1.0000000226995625, 1.000000032125072,
    1.0000000377588607, 1.0000000267041718, 0.9999999852012714, 0.9999999852012714,
    0.9999999852012714, 0.9999999852012714, 0.9999999852012714, 0.9999999852012714,
    0.9999999852012714, 0.9999999852012714, 0.9999999852012714, 0.9999999852012714,
    0.9999999852012714, 0.9999999852012714, 0.9999999852012714, 0.9999999852012714,
    0.9999999852012714, 0.9999999852012714, 0.9999999852012714, 0.9999999852012714,
    0.9999999021954707, 1.0000000045947939, 1.0000000490264376, 1.0000000509760907,
    1.0000000388539012, 1.000000025978268, 1.000000016219364, 1.0000000097376847,
    1.000000005711341, 1.0000000033022738, 1.000000001891998, 1.0000000010766026,
```

```
    1.000000000607658, 1.0000000003369973, 1.000000000177213, 1.0000000000760108]
```

可见在迭代过程的第 16 步时达到判停要求，此时输出结果 $\mathbf{x}$ 非常接近于理论值 $\mathbf{x}_{acc} = (1, 1, \cdots, 1)$，误差在 10^{-6} 以内，满足要求.

再次运行，输入矩阵的阶数 $n = 10000$，得到输出结果为（只截取部分输出，全部输出已拷贝至文档 `06_conj_grad/06_n10000_output.txt`）

```
输入矩阵的阶数 n = 10000
第 16 步后得到结果:
x = [1.000000000094299, 1.0000000002198601, 1.000000000418142, 1.0000000007541152,
    1.0000000013365071, 1.0000000023499793, 1.0000000041052013, 1.0000000071102677,
    1.0000000121520292, 1.0000000203234936, 1.0000000327852134, 1.0000000496942942,
    1.0000000670878846, 1.000000070168442, 1.0000000261694475, 0.9999999141577904,
    0.9999999999157262, 0.9999999999157262, 0.9999999999157262, 0.9999999999157262,
    0.9999999999157262, 0.9999999999157262, 0.9999999999157262, 0.9999999999157262,
    0.9999999999157262, 0.9999999999157262, 0.9999999999157262, 0.9999999999157262,
    0.9999999999157262, 0.9999999999157262, 0.9999999999157262, 0.9999999999157262,
    0.9999999999157262, 0.9999999999157262, 0.9999999999157262, 0.9999999999157262,
    0.9999999999157262, 0.9999999999157262, 0.9999999999157262, 0.9999999999157262,
    0.9999999999157262, 0.9999999999157262, 0.9999999999157262, 0.9999999999157262,
    0.9999999999157262, 0.9999999999157262, 0.9999999999157262, 0.9999999999157262,
    0.9999999999157262, 0.9999999999157262, 0.9999999999157262, 0.9999999999157262,
    0.9999999999157262, 0.9999999999157262, 0.9999999999157262, 0.9999999999157262,
    0.9999999999157262, 0.9999999999157262, 0.9999999999157262, 0.9999999999157262,
(......后续部分很长，在此省略......)
```

可以验证该结果亦满足误差要求.

6.4 源代码

该程序调用了自主编写的简易矩阵库，请参见 `mymatrixlib/matrix.py` 文件，源代码请见附录 A. 本例的源文件请见： `06_conj_grad/06_conj_grad.py`.

```
# -*- coding: utf-8 -*-
import sys
sys.path.append('..') # 为导入上级目录的mymatrixlib库
from mymatrixlib.matrix import matrix

def initialize():
    # 初始化矩阵A
    A = [[0 for i in range(n)] for i in range(n)]
    for i in range(n):
        A[i][n-1-i] = 0.5
    for i in range(n-1):
        A[i][i] = 3.
        A[i][i+1] = -1.
```

```
        A[i+1][i] = -1.
    A[n-1][n-1] = 3.

    # 初始化向量b
    b = [[1.5] for i in range(n)]
    b[0][0] = 2.5
    b[n-1][0] = 2.5
    b[int(n/2)-1][0] = 1.0
    b[int(n/2)][0] = 1.0
    return matrix(A), matrix(b)

def conj_grad(A, b, x):
    r = b - A*x
    p = r
    k = 0
    while r.mod() > 0.000001:
        # 优化的共轭梯度法
        alpha = r.mod2() / (p.T()*A*p).row[0][0]
        x = x + p.mul(alpha)
        r1 = r
        r = r - (A*p).mul(alpha)
        beta = r.mod2() / r1.mod2()
        p = r + p.mul(beta)
        k += 1
    return x, k

n = int(input('输入矩阵的阶数 n = ')) # 矩阵阶数
A, b = initialize()
x0 = matrix([[5] for i in range(n)]) # 给定x的初始值，其实具有任意性
x, k = conj_grad(A, b, x0)
print('第 %d 步后得到结果:\nx ='%k, x.column[0])
```

7 拉盖尔多项式、球谐函数与氢原子波函数

注：本稿相较于 6.18 日中午发送的版本，多加了本小题的第 (3) 问，同时对第 (2) 问中超出 double 精度的球谐函数补充了计算结果。目前该版本完成了全部小题。

7.1 解题思路

求解拉盖尔多项式与球谐函数主要用到递归的思想，需要构造合理的途径来算出指定的 $L_n^\alpha(x)$ 与 $Y_{lm}(\theta,\phi)$，具体的递归方法请见下述. 最后一问需要求解波函数的导数与积分. 求解导数我们使用有限差分方法，而积分则选取较简单的矩形面积求和.

7.2 算法提纲

第 (1) 问

根据题目所给拉盖尔多项式的递推公式，我们可以组合成由的 $L_{n-1}^\alpha(x)$, $L_{n-1}^{\alpha+1}(x)$ 推至 $L_n^\alpha(x)$, $L_n^{\alpha+1}(x)$ 的公式：

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

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

由此，可根据 $L_1^\alpha=-x+\alpha+1$ 计算出 $L_1^\alpha(x)$, $L_1^{\alpha+1}(x)$，随后递归推至 $L_n^\alpha(x)$, $L_n^{\alpha+1}(x)$. 代码结构很简单，详细请见源代码一节.

第 (2) 问

为求解球谐函数，我们需要先求连带勒让德函数. 根据连带勒让德函数的二阶递推公式

$$P_{l+1}^m(x)=\frac{x(2l+1)}{l-m+1}P_l^m(x)-\frac{l+m}{l-m+1}P_{l-1}^m(x),\quad (l>m)$$

$$P_{m+1}^m(x)=x(2m+1)P_m^m(x)$$

以及 $P_m^m(\cos\theta)=(2m-1)!!(-\sin\theta)^m$，可以递归地求出 $P_l^m(\cos\theta)$. 由此可计算出球谐函数

$$Y_{lm}(\theta,\phi)=\sqrt{\frac{(2l+1)}{4\pi}\frac{(l-m)!}{(l+m)!}}P_l^m(\cos\theta)\,\mathrm{e}^{\mathrm{i}m\phi}$$

需要注意，本题中对于 $\cos\theta=\pi/1000$，而 n 很大的情形，计算结果会超出 double 精度的范围，因此需要自主编写更高精度的浮点数计算方法. 实际程序中我们编写了“超精度”浮点数类 `strongdouble` 可分开存放浮点数科学计数法表示中的乘因子和指数，如此可实现范围为 $10^{-9223372036854775808}\sim 10^{9223372036854775808}$ 的小数的四则运算.

第 (3) 问

在 $n=2$, $l=1$, $m=\pm1$, 0 情形下，波函数为

$$\Psi_{31m}(r,\theta,\phi)=\frac{1}{\sqrt{24}}\,\mathrm{e}^{-r/2}\,r\,Y_1^m(\theta,\phi)$$

若我们对 $m = \pm 1$ 重新线性组合：

$$\psi_x(r,\,\theta,\,\phi) = \frac{1}{\sqrt{2}}(\Psi_{211} + \Psi_{21-1})$$
$$\psi_y(r,\,\theta,\,\phi) = \frac{1}{\sqrt{2}}(\Psi_{211} - \Psi_{21-1})$$

则波函数的模方，也即概率分布可最终表为

$$|\psi_x(r,\,\theta,\,\phi)|^2 = \frac{1}{32\pi}\,\mathrm{e}^r\,r^2\,\sin^2\theta\cos^2\phi$$
$$|\psi_y(r,\,\theta,\,\phi)|^2 = \frac{1}{32\pi}\,\mathrm{e}^r\,r^2\,\sin^2\theta\sin^2\phi$$
$$|\Psi_{210}(r,\,\theta,\,\phi)|^2 = \frac{1}{32\pi}\,\mathrm{e}^r\,r^2\,\cos^2\theta$$

画图时，只需要现将 $x-y$ 平面格点化，然后转化为极坐标 $(r,\,\phi)$，将 $(r,\,\theta=\frac{\pi}{2},\,\phi)$ 代入上面三个式子得到电子云概率分布，作 2D 等高线图即可观赏.

第 (4) 问

对题目中的情形，波函数的径向部分简化为

$$R_{31}(r) = \sqrt{\frac{1}{486}}\,\mathrm{e}^{-\rho/2}\,\rho\,L_1^3(\rho)\,Y_1^1(\theta,\,\phi),\quad \rho = \frac{2r}{3}$$

而能量的求解可以分为径向部分和角向部分. 因为球谐函数是归一化的，角向部分显然为 1，即

$$\int_0^{2\pi}\mathrm{d}\phi\int_0^{\pi}\sin\theta\,\mathrm{d}\theta\;|Y_{31}(\theta,\,\phi)|^2 = 1$$

再化简径向部分积分，可得

$$E = \int_0^{\infty} R_{31}(r)\left(-\frac{r^2}{2}\frac{\mathrm{d}^2}{\mathrm{d}r^2} - r\frac{\mathrm{d}}{\mathrm{d}r} - r + 1\right)R_{31}$$

将求导进行差分处理，即取离散格点 r_i 并求其上函数值 f_i, 格点间距 Δr，则

$$\frac{\mathrm{d}f}{\mathrm{d}r} = \frac{f_{i+1} - f_{i-1}}{2\Delta r},\quad \frac{\mathrm{d}^2 f}{\mathrm{d}r^2} = \frac{f_{i+1} - 2f_i + f_{i-1}}{\Delta r^2}$$

最后求解积分使用最简单的矩形面积相加即可. 实际计算中，我们选取 r 的积分范围为 $(0,\,60]$，共取了 100000 个等间距点.

7.3 结果与讨论

第 (1) 问

运行程序 `07/01_laguerre.py`，将得到输出

```
L_3^2 (0.001) = 9.990002499833334
L_3^2 (1) = 2.3333333333333335
L_3^2 (100) = -142656.66666666666
L_3^20 (0.001) = 1770.7470114998332
```

```
L_3^20 (1) = 1529.3333333333333
L_3^20 (100) = -75195.66666666667
L_3^40 (0.001) = 12340.097021499832
L_3^40 (1) = 11459.333333333334
L_3^40 (100) = -29625.666666666668
L_10^2 (0.001) = 65.7802473680385
L_10^2 (1) = -5.954546682098765
L_10^2 (100) = 7117724862237.075
L_10^20 (0.001) = 30030710.776123226
L_10^20 (1) = 18348617.842650738
L_10^20 (100) = 566337213392.425
L_10^40 (0.001) = 10269773004.722675
L_10^40 (1) = 8019281167.169131
L_10^40 (100) = 7028595559.77072
L_30^2 (0.001) = 491.0579464750633
L_30^2 (1) = 4.914163594212897
L_30^2 (100) = -1.7601474343641294e+20
L_30^20 (0.001) = 47061929154802.664
L_30^20 (1) = 10329978992206.137
L_30^20 (100) = 4.805064043941966e+16
L_30^40 (0.001) = 5.530725569029713e+19
L_30^40 (1) = 2.6212505365081723e+19
L_30^40 (100) = 230136783858607.47
```

即求得了相应的 $L_n^{\alpha}(x)$ 的值. 通过与 Mathematica 结果比较，发现在很高精度范围内正确.

第 (2) 问

运行程序 `07/02_spherical.py`，将得到输出

```
Y_100,1 (theta=pi/1000,phi=pi/5) = -0.504439E0 - 0.366497E0j
Y_100,1 (theta=3pi/10,phi=pi/5) = 0.877371E-1 + 0.637448E-1j
Y_100,1 (theta=501pi/1000,phi=pi/5) = -0.799609E-1 - 0.580950E-1j
Y_100,1 (theta=pi/1000,phi=pi/5) = -0.504439E0 - 0.366497E0j
Y_100,1 (theta=3pi/10,phi=pi/5) = 0.877371E-1 + 0.637448E-1j
Y_100,1 (theta=501pi/1000,phi=pi/5) = -0.799609E-1 - 0.580950E-1j
Y_100,10 (theta=pi/1000,phi=pi/5) = 1.039775E-14 - 0.254671E-29j
Y_100,10 (theta=3pi/10,phi=pi/5) = -3.548406E-1 + 0.869109E-16j
Y_100,10 (theta=501pi/1000,phi=pi/5) = -3.034827E-1 + 0.743318E-16j
Y_100,99 (theta=pi/1000,phi=pi/5) = -1.793376E-247 + 1.302964E-247j
Y_100,99 (theta=3pi/10,phi=pi/5) = -0.493097E-8 + 0.358256E-8j
Y_100,99 (theta=501pi/1000,phi=pi/5) = 0.341070E-1 - 0.247802E-1j
Y_500,1 (theta=pi/1000,phi=pi/5) = -4.094036E0 - 2.974492E0j
Y_500,1 (theta=3pi/10,phi=pi/5) = 0.883250E-1 + 0.641719E-1j
Y_500,1 (theta=501pi/1000,phi=pi/5) = -2.575186E-1 - 1.870982E-1j
Y_500,5 (theta=pi/1000,phi=pi/5) = 2.013461E-2 - 0.246578E-17j
```

```
Y_500,5 (theta=3pi/10,phi=pi/5) = -1.032990E-1 + 0.126505E-16j
Y_500,5 (theta=501pi/1000,phi=pi/5) = 3.183182E-1 - 0.389827E-16j
Y_500,50 (theta=pi/1000,phi=pi/5) = 1.593181E-69 - 0.195108E-83j
Y_500,50 (theta=3pi/10,phi=pi/5) = -2.316251E-2 + 0.283659E-16j
Y_500,50 (theta=501pi/1000,phi=pi/5) = -2.008539E-3 + 0.245975E-17j
Y_500,499 (theta=pi/1000,phi=pi/5) = -4.334244E-1248 + 3.149013E-1248j
Y_500,499 (theta=3pi/10,phi=pi/5) = -2.509394E-45 + 1.823181E-45j
Y_500,499 (theta=501pi/1000,phi=pi/5) = 1.136489E-1 - 0.825708E-1j
Y_1000,1 (theta=pi/1000,phi=pi/5) = -2.899262E0 - 2.106437E0j
Y_1000,1 (theta=3pi/10,phi=pi/5) = 0.883990E-1 + 0.642256E-1j
Y_1000,1 (theta=501pi/1000,phi=pi/5) = 4.042062E-4 + 2.936730E-4j
Y_1000,10 (theta=pi/1000,phi=pi/5) = 2.547470E-4 - 0.623950E-19j
Y_1000,10 (theta=3pi/10,phi=pi/5) = -3.403240E-1 + 0.833553E-16j
Y_1000,10 (theta=501pi/1000,phi=pi/5) = 3.183183E-1 - 0.779655E-16j
Y_1000,100 (theta=pi/1000,phi=pi/5) = 4.803968E-138 - 0.117663E-151j
Y_1000,100 (theta=3pi/10,phi=pi/5) = -3.491428E-1 + 0.855153E-15j
Y_1000,100 (theta=501pi/1000,phi=pi/5) = -3.190786E-1 + 0.781517E-15j
Y_1000,999 (theta=pi/1000,phi=pi/5) = -2.736426E-2499 + 1.988130E-2499j
Y_1000,999 (theta=3pi/10,phi=pi/5) = -4.018669E-91 + 2.919734E-91j
Y_1000,999 (theta=501pi/1000,phi=pi/5) = 1.906272E-1 - 1.384987E-1j
```

即求得了相应的 $Y_{lm}(\theta, \phi)$ 的值. 将上述结果与 Mathematica 结果比较，发现在很高精度范围内正确.

第 (3) 问

运行程序 `07/03_plot.py`，将得到 $|\psi_x|^2$, $|\psi_y|^2$, $|\Psi_{210}|^2$ 三个概率分布的 2D 等高线图，请见图 5. 根据量子力学经验可知求解正确.

第 (4) 问

运行程序 `07/04_hydrogen.py`，将得到输出

```
E_311 = -0.05555555598124555
```

与实际值 $E_{311} = -\frac{1}{18} = -0.05555\cdots$ 的误差仅为 8×10^{-9}，满足要求.

7.4 源代码

`07/01_laguerre.py` 源代码如下

```
# -*- coding: utf-8 -*-

def laguerreL(n, a, x):
    n0 = 1
    La, La1 = -x+a+1, -x+a+2 # 初始 n=1 时的 L_n^a(x), L_n^(a+1)(x)
    while n0 < n: # 不断迭代
```

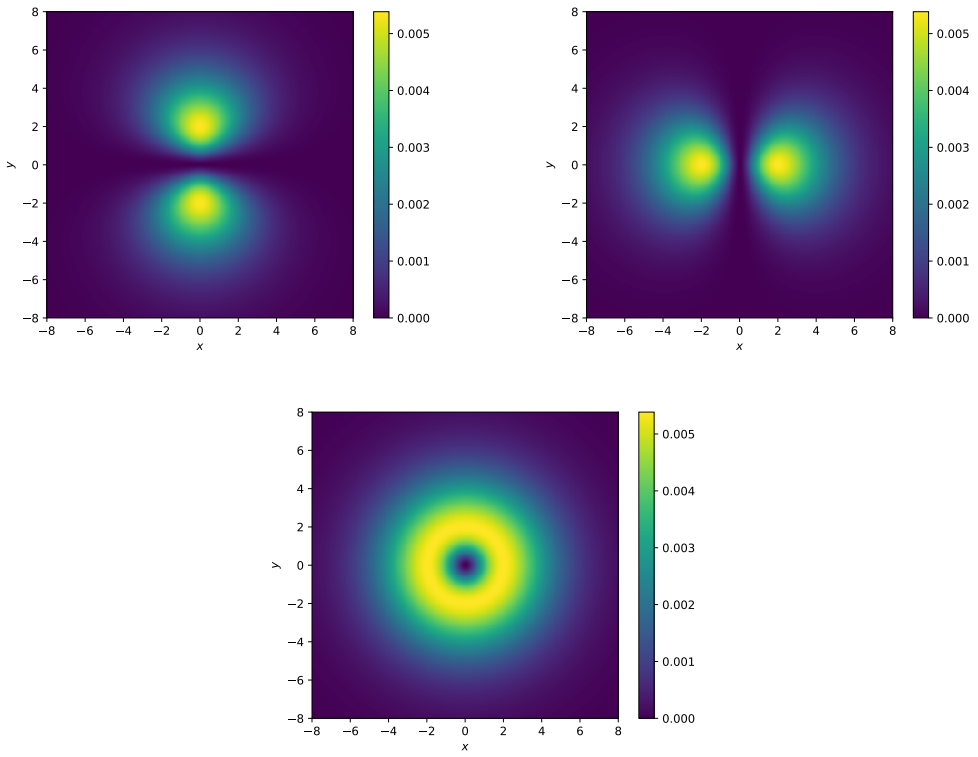

图 5: $\psi_x = \frac{1}{\sqrt{2}}(\Psi_{211} + \Psi_{21-1})$ 模方后的概率分布（左图）；$\psi_y = \frac{1}{\sqrt{2}}(\Psi_{211} - \Psi_{21-1})$ 模方后的概率分布（中图）；Ψ_{210} 模方后的概率分布（右图）.

```
        n0 += 1
        La, La1 = (-x*La1+(n0+a)*La)/n0, (-(x-n0)*La1+(n0+a)*La)/n0
    return La

for n in [3, 10, 30]:
    for a in [2, 20, 40]:
        for x in [0.001, 1, 100]:
            print("L_{n}^{a}({x}) = {ans}".format(n=n, a=a, x=x, ans=laguerreL(n,a,x)))
```

`07/02_spherical.py` 源代码如下

```
# -*- coding: utf-8 -*-

from numpy import sin, cos, pi, sqrt, log10, sign # 导入函数
from cmath import exp # e指数需要从复数中导入

class strongdouble(object): # 编写“超精度”浮点运算, sig存significand; deci存幂次部分
    def normalize(self): # 标准化
        digit = int(log10(abs(self.sig)))
        self.sig /= 10**digit
```

```
        self.deci += digit
    def __init__(self, sig, deci):
        self.sig = sig
        self.deci = deci
        self.normalize()
    def __add__(self, other): # 加法
        dist = self.deci - other.deci
        if dist < 0:
            self, other = other, self
        self.sig += other.sig*10**(-abs(dist))
        return self
    def times(self, num): # 乘以一个普通double
        self.sig *= num
        self.normalize()
        return self
    def sprint(self): # 打印成字符串
        return "%fE%d"%(self.sig, self.deci)

def sphericalharmonicY(l, m, theta, phi): # 输入参数，输出球谐函数“超精度”字符串结果
    x = cos(theta)
    Pm, Pm1 = strongdouble(1, 0), strongdouble(1, -10000) # 作为每一步 P_l-1^m, P_l, m
    进行递归，初始为 1; 0
    l0 = m
    for mul in [2*i+1 for i in range(m)]:
        Pm.times(-mul*sin(theta))
    while l0 < l: # 递归
        tmp = strongdouble(Pm.sig, Pm.deci)
        Pm, Pm1 = Pm.times(x*(2*l0+1)/(l0+1-m))+Pm1.times(-(l0+m)/(l0+1-m)), tmp
        l0 += 1
    for mul in [i for i in range(l-m+1, l+m+1)]:
        Pm.times(1/sqrt(mul))
    Pm.times(sqrt((2*l+1)/(4*pi)))
    PmRe = strongdouble(Pm.sig*cos(m*phi), Pm.deci)
    PmImabs = strongdouble(abs(Pm.sig*sin(m*phi)), Pm.deci)
    PmImsign = sign(Pm.sig*sin(m*phi))
    return "{re} {sign} {im}j".format(re=PmRe.sprint(), im=PmImabs.sprint(), sign='+' if
    PmImsign>=0 else '-')

for l in [100, 500, 1000]:
    for m in [1, int(l/100), int(l/10), l-1]:
        print("Y_{l},{m} (theta=pi/1000,phi=pi/5) = {ans}".format(l=l, m=m,
    ans=sphericalharmonicY(l, m, pi/1000, pi/5)))
        print("Y_{l},{m} (theta=3pi/10,phi=pi/5) = {ans}".format(l=l, m=m,
    ans=sphericalharmonicY(l, m, 3*pi/10, pi/5)))
        print("Y_{l},{m} (theta=501pi/1000,phi=pi/5) = {ans}".format(l=l, m=m,
```

```
    ans=sphericalharmonicY(l, m, 501*pi/1000, pi/5)))
```

07/03_plot.py 源代码如下

```
# -*- coding: utf-8 -*-

from numpy import sin, cos, pi, exp, sqrt, arctan # 导入函数

xmax, Nx = 8, 200 # [-xmax, xmax]的范围, 划为 2Nx+1 个点
X = [i*xmax/Nx for i in range(-Nx, Nx+1)]
Y = [i*xmax/Nx for i in range(-Nx, Nx+1)]
Z1p = [[0 for i in range(-Nx, Nx+1)] for i in range(-Nx, Nx+1)] # 对应 Ψ_311+Ψ_31-1
Z1m = [[0 for i in range(-Nx, Nx+1)] for i in range(-Nx, Nx+1)] # 对应 Ψ_311-Ψ_31-1
Z0 = [[0 for i in range(-Nx, Nx+1)] for i in range(-Nx, Nx+1)] # 对应 Ψ_310
for j in range(len(Z0)):
    for i in range(len(Z0[j])):
        phi = arctan(Y[j]/X[i]) if X[i]!=0 else pi/2
        r2 = X[i]**2 + Y[j]**2
        fac = 1/(32*pi) * exp(-sqrt(r2)) * r2
        Z1p[i][j] = fac * cos(phi)**2
        Z1m[i][j] = fac * sin(phi)**2
        Z0[i][j] = fac
Z1p = Z1p[::-1]
Z1m = Z1m[::-1]
Z0 = Z0[::-1]

##############################################################################
## 使用 matlibplot 作图
from matplotlib import pyplot as plt
im = plt.imshow(Z1p, extent=(-xmax, xmax, -xmax, xmax))
plt.colorbar(im)
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.savefig('Q7_3_Psi211p.pdf')
plt.show()
im = plt.imshow(Z1m, extent=(-xmax, xmax, -xmax, xmax))
plt.colorbar(im)
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.savefig('Q7_3_Psi211m.pdf')
plt.show()
im = plt.imshow(Z0, extent=(-xmax, xmax, -xmax, xmax))
plt.colorbar(im)
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.savefig('Q7_3_Psi210.pdf')
```

```
plt.show()
```

07/04_hydrogen.py 源代码如下

```
# -*- coding: utf-8 -*-

from numpy import sin, cos, pi, sqrt, exp

def laguerreL(n, a, x):
    n0 = 1
    La, La1 = -x+a+1, -x+a+2 # 初始 n=1 时的 L_n^a(x), L_n^(a+1)(x)
    while n0 < n: # 不断迭代
        n0 += 1
        La, La1 = (-x*La1+(n0+a)*La)/n0, (-(x-n0)*La1+(n0+a)*La)/n0
    return La

n, l = 2, 1
Nr = 100000
dr = 60/Nr
r = [(i+1)*60/Nr for i in range(Nr)]
Psi = [sqrt(1/486)*exp(-r[i]/3)*2*r[i]/3*laguerreL(1,3,2*r[i]/3) for i in range(Nr)] #
    Ψ的格点函数值
HPsi = [-0.5*r[i]**2*(Psi[i+1]-2*Psi[i]+Psi[i-1])/dr**2 - r[i]*(Psi[i+1]-Psi[i-1])/(2*dr)
    +
        Psi[i]*(1 - r[i]) for i in range(1, Nr-1)] # 算符作用在Ψ后的格点函数值
res = sum([Psi[i]*HPsi[i-1] for i in range(1, Nr-2)]) * dr
print("E_311 = {res}".format(res=res))
```

A 实现简易矩阵运算的库

为实现矩阵运算，编写了 `mymatrixlib` 库，其中包含了 `matrix` 类方便管理矩阵形式并定义了多种运算，如矩阵的加法、减法、乘法（重载运算符 `+`，`-`，`*`）、数乘、转置、取模、取模方. `mymatrixlib/matrix.py` 源代码如下：

```
# -*- coding: utf-8 -*-
import numpy as np  # 只用numpy里的sprt, sin, cos简单函数啊喵

# 定义 矩阵类
class matrix(object):
    def __init__(self, element):
        self.nrow = len(element) # 行数
        self.ncolumn = len(element[0]) # 列数
        for i in range(self.nrow):
            if len(element[i]) != self.ncolumn:
                print('矩阵维数不对啊亲')
                del self
        self.row = element[:] # 行向量, [[],[],[]]格式
        self.column = [[element[i][j] for i in range(self.nrow)] for j in
    range(self.ncolumn)] # 列向量, [[],[],[]]格式

    def __add__(self, other): # 矩阵加法
        if self.nrow != other.nrow or self.nrow != other.nrow:
            print('行列不匹配啊亲: A: %d * %d, B: %d * %d'%(self.nrow, self.ncolumn,
    other.nrow, other.ncolumn))
            return 'error'
        return matrix([[self.row[i][j] + other.row[i][j] for j in range(self.ncolumn)]
    for i in range(self.nrow)])

    def __sub__(self, other): # 矩阵减法
        if self.nrow != other.nrow or self.nrow != other.nrow:
            print('行列不匹配啊亲: A: %d * %d, B: %d * %d'%(self.nrow, self.ncolumn,
    other.nrow, other.ncolumn))
            return 'error'
        return matrix([[self.row[i][j] - other.row[i][j] for j in range(self.ncolumn)]
    for i in range(self.nrow)])

    def __mul__(self, other): # 矩阵乘法
        if self.ncolumn != other.nrow:
            print('行列不匹配啊亲: A: %d * %d, B: %d * %d'%(self.nrow, self.ncolumn,
    other.nrow, other.ncolumn))
            return 'error'
        C = matrix([[sum([self.row[i][m] * other.row[m][j] for m in range(self.ncolumn)])
    for j in range(other.ncolumn)] for i in range(self.nrow)])
        # python就是妙，一行完成
```

```
        return C

    def mul(self, n):
        return matrix([[n * self.row[i][j] for j in range(self.ncolumn)] for i in
    range(self.nrow)])

    def T(self): # 矩阵转置
        return matrix(self.column)

    def mod2(self):
        if self.nrow != 1 and self.ncolumn != 1:
            print('向量不是 1 * n 格式的啊亲')
            return 'error'
        if self.nrow == 1:
            return sum([self.row[0][i]**2 for i in range(self.ncolumn)])
        if self.ncolumn == 1:
            return sum([self.row[i][0]**2 for i in range(self.nrow)])

    def mod(self):
        ans = self.mod2()
        if ans == 'error':
            return 'error'
        else:
            return np.sqrt(ans)

    def copy(self): # 矩阵复制
        return matrix([[self.row[i][j] for j in range(self.ncolumn)] for i in
    range(self.nrow)])
```