“计算物理学”第 2 次大作业

李聪乔 1500011425

2019 年 5 月 19 日

目录

前言

虽然本次作业提交地的较晚，但事实上我已在 5 月 6 日之前完成的所有代码的编写（起初在 Jupyter 上编写，后移植为 `.py` 文件），并得到了正确的结果. 这一点许英伦、王任飞同学可以作证. 但不幸的是，撰写这份文档着实花费了我许久的时间，因为这学期的事情实在太多（有 30 学分课程及两个科研同期进行），且调整本文档的格式也比较费时费力，所以一直没有最终完稿. 拖到现在，真的非常抱歉，还请助教哥哥与姐姐手下领情. 另外，虽然笔者身为大四狗，但由于决定 gap 一年，所以这学期的绩点还要继续保持，因此真切地希望本次作业不会扣很多分，非常感谢!

1 Mathieu 方程求解：实对称矩阵的实用 QR 算法求本征值问题

题目：为求解正则形式的 *Mathieu* 方程

$$-\frac{\mathrm{d}^2\Psi(\phi)}{\mathrm{d}\phi^2}+2q\cos(2\phi)\Psi(\phi)=A\Psi(\phi),$$

可利用离散变量表示法求解. 利用傅里叶基矢 $\Phi_n(\phi)=\frac{1}{\sqrt{2\pi}}\exp(\mathrm{i}n\phi)$, $n=-M,-(M-1),\cdots,M$, 构造基函数

$$f_k(\phi)=\frac{1}{2M+1}\sum_{n=-M}^{M}\Phi_n\left(\frac{2\pi}{2M+1}k\right)\Phi_n(\phi),\quad k=1,2,\cdots,2M+1$$

可将 *Mathieu* 方程本征函数 $\Psi(\phi)$ 展开为

$$\Psi(\phi)=\sum_{k=1}^{2M+1}C_kf_k(\phi).$$

可以证明在 $f_k(\phi)$ 基底展开下等价求解方程

$$\boldsymbol{H}\Psi=A\Psi$$

其中

$$\boldsymbol{H}=\begin{cases}(-1)^{j-k}\frac{\cos[\pi(j-k)/(2M+1)]}{2\sin^2[\pi(j-k)/(2M+1)]}, & j\neq k\\ \frac{M(M+1)}{3}+2q\cos\left(\frac{2\pi}{2M+1}j\right), & j=k\end{cases}$$

(1) 写出最一般的实对称矩阵的实用 *QR* 算法，包括 *Householder–Hessenberg* 约化，*Givens* 旋转变换、原点位移等.

(2) $M = 50$, 对参数 $q \in [0, 20]$ 计算头 11 个本征值，作 (q, A) 图.

(3) 算法收敛性：对比 $M = 5, 40$ 两种情形下的头 5 个偶宇称解的本征值，画在一张 (q, A) 图上.

(4) 对 $M = 50$, $q = 10.0$ 求头 6 个偶宇称的本征值和本征矢 $\Psi_i^{(\mathrm{even})}(\phi)$.

1.1 解题思路

本题的核心即在于如何求解一般的实对称矩阵的本征值及本征矢量. 根据提示，我们采用一般的实用 QR 算法进行求解. 一个显然的结论是，对任意矩阵 $\boldsymbol{H}$，其相似变换后的矩阵 $\boldsymbol{QHQ}^{\mathrm{T}}$ 的本征值与原矩阵 $\boldsymbol{H}$ 的本征值是相同的，并且它们的本征矢之间差一个 $\boldsymbol{Q}$ 的正交变换，这可作为我们的出发点. 实用 QR 算法通过不断对原矩阵 $\boldsymbol{H}$ 作相似变换使之最终成为对角矩阵，即可由对角阵的对角元得到本征值. 将该过程中的所有相似变换矩阵按照顺序相乘即得到整体的正交变换矩阵 $\boldsymbol{Q}$，这样 $\boldsymbol{Q}$ 的每一列对应着相应的本征矢量.

具体来讲，实用 QR 算法包含如下几步：

(i) 首先，使用 Householder–Hessenberg 变换将原矩阵变为上 Hessenberg 矩阵，其特征是 $A_{ij} = 0$, (任意 $i - j > 1$). 方法是找到合适的 Householder 矩阵 $\boldsymbol{H} = \boldsymbol{I} - 2\frac{\boldsymbol{w}\boldsymbol{w}^{\mathrm{T}}}{\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w}}$，它作用在任意矢量上可得到该矢量沿以 $\boldsymbol{w}$ 为法线方向的镜面的反射矢量. 利用 Householder 矩阵构造正交相似变换可使得原矩阵的 $H_{i1} = 0$ $(i = 3, 4, \cdots, n)$. 用相似地思路重复多次，每次相似变换时将某一列的 -1 次对角线下方的元素变为 0，最终可使矩阵转化为上 Hessenberg 矩阵.

值得指出的是，由于本算例中原矩阵 $\boldsymbol{H}$ 是对称矩阵，因此转化后的上 Hessenberg 矩阵保留其对称性，为三对角矩阵.

(ii) 其次，使用 Givens 变换将上面得到的上 Hessenberg 矩阵转化为上三角矩阵，方法是构造多个局域的旋转矩阵 $\boldsymbol{g}(i-1, i)$，这里 $i = 2, 3, \cdots, m$，m 为矩阵阶数. 将 $\boldsymbol{g}(i-1, i)$ 左乘在 $\boldsymbol{H}$ 上可消掉 $H_{i,i-1}$ 元，重复多次可使原矩阵变成上三角矩阵，这些局域矩阵构成一个正交阵 $\boldsymbol{G} = \boldsymbol{g}(1, 2) \cdots, \boldsymbol{g}(m-1, m)$，在将 $\boldsymbol{G}^{\mathrm{T}}$ 右乘在 $\boldsymbol{H}$ 上，相当于与本步开始前的矩阵关系为 $\boldsymbol{H}' = \boldsymbol{GHG}^{\mathrm{T}}$，完成一次迭代过程. 定理表明，不断进行该步骤即可收敛于对角矩阵. 为加快收敛速度，可使用原点位移方法，即先将本步开始前先作代换 $\widetilde{\boldsymbol{H}} = \boldsymbol{H} - s\boldsymbol{I}$，完成相似变换 $\widetilde{\boldsymbol{H}}' = \boldsymbol{G}\widetilde{\boldsymbol{H}}\boldsymbol{G}^{\mathrm{T}}$ 后再原点位移回来: $\widetilde{\boldsymbol{H}} = \widetilde{\boldsymbol{H}}' + s\boldsymbol{I}$，可见依然有 $\boldsymbol{H}' = \boldsymbol{GHG}^{\mathrm{T}}$. 经验表明选取待变换矩阵最末一位对角元作为 s 可使收敛速度最快.

1.2 算法提纲

本例使用实用 QR 算法求解矩阵本征值的算法思路如上节所述,其包含 Householder–Hessenberg 变换、Givens 旋转变换、原点位移等技巧. 第一步:Householder–Hessenberg 算法的伪代码如下所示.

```
## 首先利用 Householder–Hessenberg 算法将矩阵 H 化为上 Hessenberg 矩阵
for k = 1, 2, ··· , m − 2   (维数为 m)
        x = H_{k+1:m, k}   (当前步的列向量)
        v_k = sign(x_1)‖x‖_2 e_1 + x   (构造反射法向矢量)
        v_k = v_k/‖v_k‖_2   (归一化)
        H_{k+1:m, k:m} = H_{k+1:m, k:m} − 2v_k(v_k^T H_{k+1:m, k:m})   (相当于左乘反射阵)
        H_{1:m, k+1:m} = H_{1:m, k+1:m} − 2(H_{1:m, k+1:m} v_k)v_k^T   (相当于右乘反射阵转置)
end for
```

此算法后矩阵 $\boldsymbol{H}$ 已变为上 Hessenberg 矩阵. 其后进行 QR 分解迭代过程，用到了 Givens 变换与原点位移技术，伪代码如下所示.

```
## 其次利用 QR 分解不断进行相似变换的迭代，使之收敛于对角阵 k := n
(对前 k 行、k 列执行 QR 算法，k 从 n 不断减为 1 为止)
while (k > 1) do
        if H_{k,k−1} < 1 × 10^{−6} then
                k := k − 1   (找到本征值 H_{k,k}，后面只需继续对 H_{1:k, 1:k} 进行迭
代)
        end if

        s := a_{kk}
        for j = 1, 2, ··· , k
                a_{jj} := a_{jj} − s   (原点位移，将对角元位移 s)
        end for
        for j = 1, 2, ··· , k − 1
                计算局域旋转矩阵 g(j, j + 1)，使左乘 H 后 H_{j+1,j} = 0
                H_{1:k, 1:k} = g(j, j + 1)H_{1:k, 1:k}   (注意！此处基于 g 和 H 的性质采
取了特殊的矩阵乘法操作，见后述)
                记录 g(j, j + 1)
        end for   (接下页)
```

```
## 此时完成了 QR 分解，H 已变成上三角矩阵
    for j = 1, 2, ··· , k − 1
        H_{1:k,1:k} = H_{1:k,1:k} g^T(j, j + 1)  （再右乘旋转矩阵，完成一次迭代）
    end for
    for j = 1, 2, ··· , k
        a_jj := a_jj + s  （原点移回来）
    end for
## 完成一次迭代过程
end while
```

每迭代一次，矩阵应由上 Hessenberg 矩阵更加向对角矩阵靠拢，也即其非对角元在过程中逐渐趋于 0. 在本例中，$\boldsymbol{H}$ 为对称矩阵，上 Hessenberg 矩阵实际为三对角矩阵，则在迭代过程中两条 ± 1 次对角元在逐渐趋于 0. 当 $k = 1$ 时，说明所有本征值已找到, $\boldsymbol{H}$ 趋于对角矩阵，其各对角元即为需要求得的本征值.

值得指出的是，由于 Givens 旋转矩阵非常特殊，计算它与 $\boldsymbol{H}$ 的乘积时可以不用常规的矩阵乘法，从而使运算简化. 对 k 阶矩阵相乘，常规方法的时间复杂度为 $\mathcal{O}(k^3)$. 但 Givens 旋转矩阵 $\boldsymbol{g}$ 作用在任意矩阵时，仅相当于对该矩阵的某两行（$\boldsymbol{g}$ 前乘）或两列（$\boldsymbol{g}$ 后乘）的向量重新作一线性组合，其计算复杂度仅为 $\mathcal{O}(k)$. 又因为我们处理的是矩阵 $\boldsymbol{H}$ 为三对角矩阵，对于相邻行、列的局域旋转，实际可以只提取长度至多为 4 的向量进行线性组合，从而将计算复杂度进一步缩减为 $\mathcal{O}(1)$.

本题第 (3) 小问需确定本征值相应本征矢的宇称. 求解本征矢的方法有两种，如下所述. 本题采取第一种方法.

(i) 记录下 Householder–Hessenberg 约化过程的每一步 Householder 矩阵，同时记录下 QR 迭代中每一步的 Givens 旋转矩阵，将这些正交矩阵依次相乘得到 $\boldsymbol{Q}$, 则有 $\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{Q} = \boldsymbol{D}$，$\boldsymbol{D}$ 为对角矩阵，因此 $\boldsymbol{Q}$ 的每个列向量即为 $\boldsymbol{D}$ 中相应对角元的本征矢.

(ii) 可采用 QR 方法无关的求解本征矢算法：利用反幂法求得每个本征值相应的本征矢量.

在求得本征矢的基础上，可利用本例中基函数 $f_k(\phi)$ 的性质判断本征矢的奇偶性. 由于基函数表达式为

$$f_k(\phi) = \frac{1}{2M+1} \sum_{n=-M}^{M} \Phi_n\left(\frac{2\pi}{2M+1}k\right)\Phi_n(\phi), \quad k = 1, 2, \cdots, 2M+1,$$

可以推知，在 $M \to \infty$ 时有

$$f_k(\phi) \to \delta\left(\phi - \frac{2\pi}{2M+1}k\right),$$

考虑到本征矢是周期为 2π 的函数，因此相对于 $\phi = 0$ 的奇偶性 $\Psi(\phi) = \pm\Psi(-\phi)$ 等价于 $\Psi(\phi) = \pm\Psi(2\pi - \phi)$，表明函数关于轴 $\phi = \pi$ 的奇偶性与关于 $\phi = 0$ 的奇偶性相同，于是可通过 C_1 与 C_{2M} 的符号关系来确定本征矢 $\Psi = (C_1, \cdots, C_{2M+1})^{\mathrm{T}}$ 的宇称态：若 C_1 与 C_{2M} 同号，则为偶宇称；异号则为奇宇称[1].

最后说明一下本算例调用的库与代码结构. 除最后作图外,代码仅调用了 `math` 库中的基本函数. 进行矩阵基本操作时沿用了第一次作业中自编的 `mymatrixlib` 包，用以简化代码. 不过正如上文所述，本例中很多矩阵的乘法可用利用矩阵的特点，用时间复杂度更低的方式实现（如与 Givens 矩阵有关的乘法），而不必使用传统的矩阵乘法，这一部分的具体细节可参见章末代码的注释. 本题的代码结构如下：`find_eigenvalues` 函数集成了实用 QR 算法求解本征值并排序、求本征矢与确定奇偶宇称的算法；`mat_initialize` 函数利用指定的参数 M, q 对矩阵 $\boldsymbol{H}$ 进行初始化. 以上两个个函数单独写在文件 `func01. py` 中，此后每一小问的计算各自编成一个文件，需要调用 `func01` 中的函数，以及上级文件夹中的矩阵运算包 `mymatrixlib` . 详情参见章末的源代码.

1.3 结果与讨论

首先笔者建议,为了用最便捷的方法快速测试结果,可运行 `01_mathieu_solver/simple_example. py`. 代码主要部分节选如下：

```
H = mat_initialize(M=2, q=1)  # 初始化矩阵
H0 = H.copy()  # 把初始 H 记下来
result = find_eigenvalues(H, need_eigenstates=True)  # 计算本征值、本征矢
## 测试结果
('H = '); H0.print()
print('Q = '); result['Q'].print()
print('Q^T H Q = '); (result['Q'].T() * H0 * result['Q']).print()  #
    验证结果为对角阵
```

下面为各小问的结果.

[1]但值得指出的是，如果某一本征值下出现奇、偶能级简并，则实际求得的本征函数可能是二者的任意叠加，函数的宇称将会消失. 遇到这种问题时需要进行额外的考虑. 本例中，在 $q = 0$ 时方程退化为谐振子方程，即会出现上述问题.

第 (1) 问

有关算法的描述请详见上一节“算法提纲”.

第 (2) 问

指定 $M = 50; q \in [0, 20]$, 间距 $\Delta q = 1.0$ 的参数空间，绘制前 11 个本征值（由小到大排序）$A_{1,\cdots,11}$ 随 q 变化的曲线，如图 1 所示. 实际代码测试时，直接运行 `01_mathieu_solver/01_2_plot_eigenvalue. py` 即可,程序将自动使用 `matplotlib` 包作图.

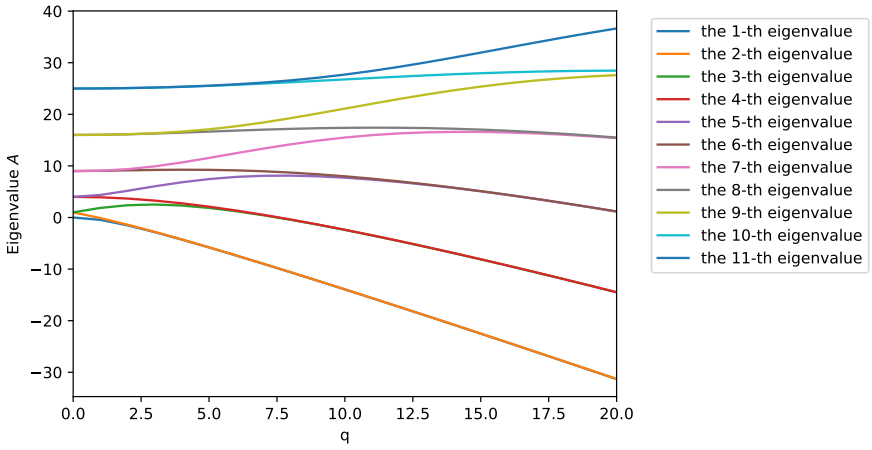

图 1: $M = 50; q \in [0, 20]$, 间距 $\Delta q = 1.0$ 的参数空间下前 11 个本征值 $A_{1,\cdots,11}$ 随 q 变化的曲线.

第 (3) 问

指定 $M = 5, 40; q \in [0, 20]$, 间距 $\Delta q = 1.0$ 的参数空间，绘制前 5 个偶宇称态（由小到大排序）$A^{\text{even}}_{1,\cdots,11}$ 随 q 变化的曲线，如图 2 所示. 实际代码测试时，直接运行 `01_mathieu_solver/01_3_plot_even_value. py` 即可，程序将自动使用 `matplotlib` 包作图.

需要指出，当 $q = 0$ 时，Mathieu 方程的解退化为三角函数，此时每个本征值可认为具有奇、偶宇称简并的两个本征矢，因此实际求解的本征矢可能是二者的叠加，无法从本征矢来判断宇称性. 实际的程序求解过程中，我们将 $q = 0$ 的本征值排好序后，按照 $q = 1.0$ 排序后的本征值相应的奇偶性来进行分配，这样可以保证对应的能级是随参数 q 连续变化的，从而使得绘制的曲线是也是随 q 连续变化的.

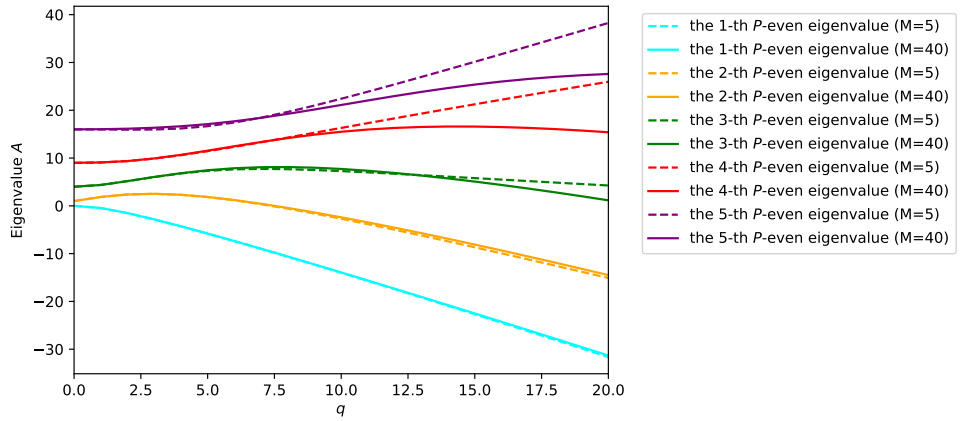

图 2: $M = 5, 40$; $q \in [0, 20]$, 间距 $\Delta q = 1.0$ 的参数空间下前 5 个偶宇称态 $A^{\text{even}}_{1,\cdots,11}$ 随 q 变化的曲线.

第 (4) 问

指定 $M = 50$; $q = 10.0$, 绘制前 6 个个偶宇称态（由小到大排序）本征函数 $\Psi_i^{\text{even}}(\Phi)$. 实际运算中, 取 ϕ 为 $[0, \pi/2]$ 区间上的 40 等分格点, 对基函数 $f_k(\phi)$ 采样并数值计算, 可以得到一组实函数. 由误差引起的微小的虚部直接忽略即可. 将本征矢 $\Psi = (C_1, \cdots, C_{2M+1})^{\mathrm{T}}$ 代入 $\Psi(\phi) = \sum_k C_k f_k(\phi)$ 即可计算出 Mathieu 方程的解, 如图 4 所示. 值得指出, 本算例没有对最终的解进行归一化, 故将每条曲线乘以任意非零系数（可以是负数, 甚至更加一般的复数）仍然是满足要求的.

实际代码测试时, 直接运行 `01_mathieu_solver/01_4_plot_even_state.py` 即可, 程序将自动使用 `matplotlib` 包作图.

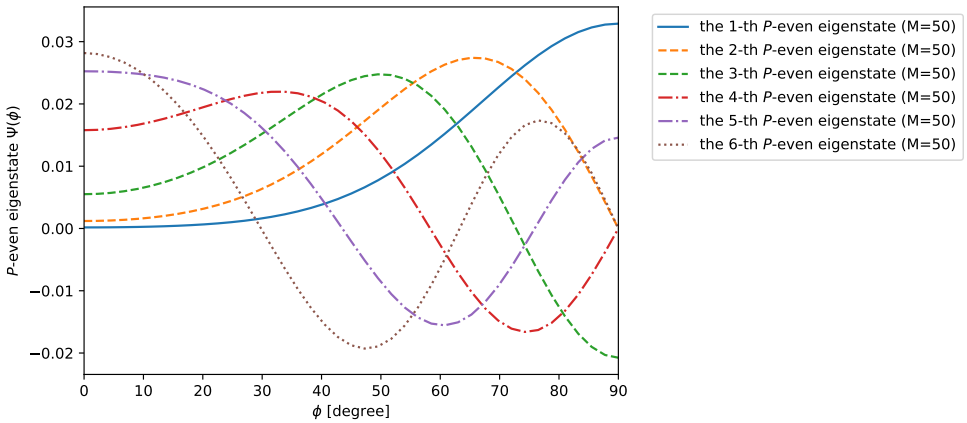

图 3: $M = 50$; $q = 10.0$ 时, 前 6 个个偶宇称态的本征函数 $\Psi_i^{\text{even}}(\Phi)$.

1.4 源代码

主函数代码

包含 `find_eigenvalues` 与 `mat_initialize` 函数的文件: `func01.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 numpy import sin, cos, sqrt, exp, sign, pi

# 通过指定参数 M, q 初始化哈密顿量矩阵
def mat_initialize(M, q):
    # 初始化矩阵: 对角元为 M(M+1)/3 + V_ii, 非对角元为较复杂的 T_ij
    return matrix([[M*(M+1)/3 + 2*q*cos(4*pi*(j+1)/(2*M+1)) if j==k else
                (-1)**(j-k) * cos(pi*(j-k)/(2*M+1)) / (2 *
   sin(pi*(j-k)/(2*M+1))**2 )
                for k in range(0, 2*M+1)] for j in range(0, 2*M+1)])


# 输入矩阵, 求解本征值、本征矢并判断宇称
def find_eigenvalues(H, need_eigenstates=False): # H
    为输入矩阵, need_eigenstates为是否需要输出本征矢, 如不需要可简化运算.

    # 有关 Givens 矩阵乘法的简易乘法运算
    def mat_update_from_given(Mat, ndim, j, angle, pos='front',
   bandshape=True):
        # angle: 旋转的角度, 格式为[sin,cos];
        # pos: Givens矩阵是前/后乘在某矩阵上;
        # bandshape表示矩阵是否为三对角的 (若是, 可进一步简化)
        if bandshape==True:  # 是三对角阵
            nstart = max(0, j-1)  # 只需截取长度1*4的向量即可
            nend = min(ndim, j+3)
        else:  # 不是, 需截取整行/整列向量
            nstart = 0
            nend = ndim
```

```
    if pos=='front':  # 前乘在某矩阵上
        row1 = Mat.slice(j, j+1, nstart, nend)  # 切片下一个行向量
        row2 = Mat.slice(j+1, j+2, nstart, nend)
        Mat.paste_on(row1.mul(angle[1]) + row2.mul(angle[0]), j,
nstart)  # 重新组合后“贴”回原矩阵
        Mat.paste_on(row1.mul(-angle[0]) + row2.mul(angle[1]), j+1,
nstart)
        return Mat
    if pos=='back':  # 前乘在某矩阵上
        col1 = Mat.slice(nstart, nend, j, j+1)  # 切片下一个列向量
        col2 = Mat.slice(nstart, nend, j+1, j+2)
        Mat.paste_on(col1.mul(angle[1]) + col2.mul(angle[0]), nstart,
j)  # 重新组合后“贴”回原矩阵
        Mat.paste_on(col1.mul(-angle[0]) + col2.mul(angle[1]),
nstart, j+1)
        return Mat

m = H.nrow  # 矩阵阶数
# 注意: 本例中矩阵 index 都是从 0 开始的, 与数学习惯不同
# 以下为 Householder-Hessenberg 约化
vkts = []  # 储存每个反射矩阵相应的法向量vkt
for k in range(1, m-1): # k=1...m-2
    xt = matrix([[H.elem[i][k-1] for i in range(k, m)]])  #
截取当前步的列矢量 A_{k+1:m, k}, 不过这里用行矢量存储了
    vkt = matrix([[sign(xt.elem[0][0]) * xt.mod() if i==0 else 0 for
i in range(m-k)]]) + xt  # 反射线法矢量
    vkt = vkt.mul(1 / vkt.mod())  # 归一化

    # 第一步操作: 左乘反射矩阵 (用讲义中算法)
    u = [sum([vkt.elem[0][i-k] * H.elem[i][j] for i in range(k, m)])
for j in range(k-1, m)]
    for i in range(k, m):
        for j in range(k-1, m):
            H.elem[i][j] -= 2 * vkt.elem[0][i-k] * u[j-k+1]

    # 第二步操作: 右乘反射矩阵转置
```

```
    u = [sum([H.elem[i][j] * vkt.elem[0][j-k] for j in range(k, m)])
for i in range(0, m)]
    for i in range(0, m):
        for j in range(k, m):
            H.elem[i][j] -= 2 * u[i] * vkt.elem[0][j-k]
    vkts.append(vkt)

# 此时 H 已约化为上 Hessenberg 矩阵

# 以下进行 Givens 旋转变换
k = m  # 找本征值的指标, 从 m 逐渐减小
angles_repo = [] # 储存正交矩阵的每个旋转角度
it = 0
while k > 1:
    if abs(H.elem[k-1][k-2]) < 1e-6:  # 找到一个本征值的判停标准
        k = k-1

    # 原点位移
    s = H.elem[k-1][k-1]
    for j in range(k):  # j=0...k-1
        H.elem[j][j] -= s

    angles = []
    for j in range(k-1):  # j=0...k-2
        norm = sqrt(H.elem[j+1][j]**2 + H.elem[j][j]**2)
        angles.append([H.elem[j+1][j]/norm, H.elem[j][j]/norm]) #
当前的[sin, cos]
        mat_update_from_given(H, k, j, angles[-1], pos='front') #
前乘正交阵G使得A_{j+1,j}=0
    # 此时 H 已旋转至上三角矩阵, 角度储存在angles中

    angles_repo.append(angles)
    for j in range(k-1):
        mat_update_from_given(H, k, j, angles[j], pos='back') #
再后乘矩阵G^T
```

```
        # 原点移回来
        for j in range(k):  # j=0...k-1
            H.elem[j][j] += s
        it += 1

        if it > 500:
            return 'fail!' # 循环太多，就不算了
    eigens = [H.elem[i][i] for i in range(m)]

    # 此时所有本征值已经存入eigens，下面对eigens进行冒泡排序
    order = [i for i in range(m)]
    def swap(a, b):
        return b, a
    for i in range(m-1, 0, -1):
        for j in range(i):
            if eigens[j] > eigens[j+1]:
                eigens[j], eigens[j+1] = swap(eigens[j], eigens[j+1])
                order[j], order[j+1] = swap(order[j], order[j+1]) #
    也要将原始位置按相同方式重排，为了后面本征矢按相同规则换序

    if need_eigenstates == False:  # 若无需输出本征矢
        return eigens  # 只返回本征值即可啦

    else:  # 若需要本征矢，下面要对各个正交矩阵相乘
        Q = matrix.identity(m)  # 从单位矩阵开始

        # Householder-Hessenberg 约化过程的各个正交阵乘在Q上
        for k in range(1, m-1): # k=1...m-2
            u = [sum([Q.elem[i][j] * vkts[k-1].elem[0][j-k] for j in
    range(k, m)]) for i in range(0, m)]
            for i in range(0, m):
                for j in range(k, m):
                    Q.elem[i][j] -= 2 * u[i] * vkts[k-1].elem[0][j-k]

        # Givens过程的各个正交阵乘在Q上
        for angles in angles_repo:
```

```
            for j in range(len(angles)):
                mat_update_from_given(Q, m, j, angles[j], pos='back',
    bandshape=False)
        # 此时Q的各个列向量即为本征矢

        # 对Q中的列向量（本征矢）按照本征值排序时的索引调换顺序，得到Q_sort
        Q_sort = matrix.identity(m)
        for i in range(m):
            Q_sort.paste_on(Q.slice(0, m, order[i], order[i]+1), 0, i)

        # 有了本征矢可以判断宇称，存入is_even
        is_even = []
        for i in range(m):
            if Q_sort.elem[0][i]/Q_sort.elem[m-2][i] > 0:
                is_even.append(True)
            else:
                is_even.append(False)
        dic = {'eigens': eigens, 'Q': Q_sort, 'is_even': is_even}
        return dic  # 返回本征值，本征矢，是否偶宇称
```

第 (2) 问代码

第 (2) 问代码：`01_2_plot_eigenvalue.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 func01 import mat_initialize, find_eigenvalues

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



print('第(2)问：\n')
M = 50
eigen_list = []  #储存每个q下的一系列本征值
q_list = [i for i in range(0,21,1)]  # q的各参数点
```

```
for i, q in enumerate(q_list):
    H = mat_initialize(M, q)  # 先初始化原矩阵 H
    eigen_list.append(find_eigenvalues(H, need_eigenstates=False)[0:11])
   # 只计算本征值, 并储存头11个本征值
    print('正在对不同的 q 计算本征值: [%-3d / %-3d] 进行中...'%(i,
    len(q_list)))
print('计算完成, 下面画图')



########################## 使用 matplotlib 画图 ##########################
from matplotlib import pyplot as plt
fig, ax1 = plt.subplots(figsize=(9,4.5)) # 调节画布大小等
box = ax1.get_position()
ax1.set_position([box.x0, box.y0, box.width* 0.7 , box.height])
for i in range(11):
    plt.plot(q_list, [eigen_list[iq][i] for iq in range(len(q_list))],
    label='the %d-th eigenvalue'%(i+1))
print('画图完成')
plt.xlim(0,20)
plt.legend(bbox_to_anchor=(1.05,1.0))
plt.xlabel('q')
plt.ylabel(r'Eigenvalue $A$')
plt.savefig('01_2.pdf')
plt.show()
```

第 (3) 问代码

第 (3) 问代码: `01_3_plot_even_value.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 func01 import mat_initialize, find_eigenvalues

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

```
def f(m, k, phi):  # 定义基函数 f_k(phi)
    return 1/(m*2*pi) *
    sum([exp(-1j*(n-(m-1)/2)*(2*pi*k/m))*exp(1j*(n-(m-1)/2)*phi) for n in
    range(m)]).real


print('第(3)问: 大概要耗时 1.5min, 请稍等~ \n')
eigen_list_M5 = []
eigen_list_M40 = []
q_list = [iq*1. for iq in range(0,21,1)]
for iq, q in enumerate(q_list):
    print('M = 5, 40, 正在对不同的 q
    计算“本征值”和“本征矢（花费时间较多）”:   [%-3d / %-3d]
    进行中...'%(iq, len(q_list)))
    H = mat_initialize(5, q)  # M=5
    eigen_list_M5.append(find_eigenvalues(H, need_eigenstates=True)) #
    将计算出的本征值和本征矢都存入eigen_list
    H = mat_initialize(40, q)  # M=40
    eigen_list_M40.append(find_eigenvalues(H, need_eigenstates=True)) #
    将计算出的本征值和本征矢都存入eigen_list

# 下面从is_even列表中提取出偶宇称
eigen_even_list_M5 = []  # M=5 偶宇称态的本征值列表
eigen_even_list_M40 = []  # M=40 偶宇称态的本征值列表
for iq in range(len(q_list)):
    iqe = iq if iq!=0 else 1  # 因奇偶宇称的简并, 需对 q=0
    宇称态情形的特殊处理 (详见“算法提纲”一节)
    eigens_M5 = []
    eigens_M40 = []
    for i, flag in enumerate(eigen_list_M5[iqe]['is_even']):
        if flag == True:  # 如果是奇宇称
            eigens_M5.append(eigen_list_M5[iq]['eigens'][i])
    for i, flag in enumerate(eigen_list_M40[iqe]['is_even']):
        if flag == True:
            eigens_M40.append(eigen_list_M40[iq]['eigens'][i])
```

```
    eigen_even_list_M5.append(eigens_M5)
    eigen_even_list_M40.append(eigens_M40)
print('计算完成, 下面画图')


########################## 使用 matplotlib 画图 ##########################
from matplotlib import pyplot as plt
fig, ax1 = plt.subplots(figsize=(10,4.5)) # 调节画布大小等
box = ax1.get_position()
ax1.set_position([box.x0, box.y0, box.width* 0.65 , box.height])
color = ['cyan', 'orange', 'green', 'red', 'purple']
for i in range(5):
    plt.plot(q_list, [eigen_even_list_M5[iq][i] for iq in
   range(len(q_list))], color=color[i], linestyle='--', label=r'the %d-th
   $P$-even eigenvalue (M=5)'%(i+1))
    plt.plot(q_list, [eigen_even_list_M40[iq][i] for iq in
   range(len(q_list))], color=color[i], label='the %d-th $P$-even
   eigenvalue (M=40)'%(i+1))
print('画图完成')
plt.xlim(0,20.)
plt.legend(bbox_to_anchor=(1.05,1.0))
plt.xlabel(r'$q$')
plt.ylabel(r'Eigenvalue $A$')
plt.savefig('01_3.pdf')
plt.show()
```

第 (4) 问代码

第 (4) 问代码: `01_4_plot_even_state.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 func01 import mat_initialize, find_eigenvalues

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

```
def f(m, k, phi):  # 定义基函数 f_k(phi)
    return 1/(m*2*pi) *
    sum([exp(-1j*(n-(m-1)/2)*(2*pi*k/m))*exp(1j*(n-(m-1)/2)*phi) for n in
    range(m)]).real


print('第(4)问: 大概耗时 ~10s \n')
M = 50
q = 10.
m = 2*M + 1
H = mat_initialize(M, q)  # M=5 初始化矩阵
eigendic_M50 = find_eigenvalues(H, need_eigenstates=True) #
    将计算出的本征值和本征矢都存入eigen_list
print('计算完成, 下面画图')


########################## 使用 matplotlib 画图 ##########################
from matplotlib import pyplot as plt
fig, ax1 = plt.subplots(figsize=(10,4.5)) # 调节画布大小等
box = ax1.get_position()
ax1.set_position([box.x0, box.y0, box.width* 0.65 , box.height])
Nphi = 40  # 40个  的格点
phi_list = [i*0.5*pi/Nphi for i in range(0, Nphi+1)]  #  区间[0, /2]
linestyle = ['-', '--', '--', '-.', '-.', ':']
iplot = 0
for i, flag in enumerate(eigendic_M50['is_even']):
    if flag == True:  # 如果是偶宇称解, 画图!
        plt.plot([phi*180/pi for phi in phi_list],
    [-sum([eigendic_M50['Q'].elem[n][i]*f(m, n+1, phi) for n in range(m)])
    for phi in phi_list],\
                linestyle=linestyle[iplot], label='the %d-th $P$-even
    eigenstate (M=50)'%(iplot+1))
        iplot += 1
        if iplot == 6:  # 画满 6 个为止
```

```
            break
print('画图完成')
plt.xlim(0,90.)
plt.legend(bbox_to_anchor=(1.05,1.0))
plt.xlabel(r'$\phi$ [degree]')
plt.ylabel(r'$P$-even eigenstate $\Psi(\phi)$')
plt.savefig('01_4.pdf')
plt.show()
```

2 氢原子在强激光场下电离：鞍点近似法

题目：在外加强激光场的偶极作用下，一个电子从氢原子基态跃迁至激光场中的本征态 *(Volkov* 态*)* 并获得动量 $\boldsymbol{p}$ 的跃迁几率可由 *Dyson* 方程并直接积分计算为

$$M_{\boldsymbol{p}}^{0}(t_f,\, t_i)_{\mathrm{DI}} = 2^{\frac{7}{2}}(2I_p)^{\frac{5}{4}} \int_{t_i}^{t_f} \mathrm{d}t\, \frac{\boldsymbol{q}\cdot\boldsymbol{E}(t)}{\pi(\boldsymbol{q}2+2I_p)^3} \mathrm{e}^{\mathrm{i}S(t)}$$

其中

$$S(t) = \int_0^t \mathrm{d}\tau \left[\frac{1}{2}(\boldsymbol{p}+\boldsymbol{A}(\tau))^2 + I_p\right].$$

可用鞍点法给出其跃迁振幅的近似表达式

$$M_{\boldsymbol{p}}^{0}(t_f,\, t_i)_{\mathrm{SPM}} \ = -\frac{(2I_p)^{\frac{5}{4}}}{\sqrt{2}} \sum_{\alpha} \frac{1}{S''(t_{s\alpha})} \mathrm{e}^{\mathrm{i}S(t_{s\alpha})}$$

其中 $t_{s\alpha} = t_{r\alpha} + \mathrm{i}t_{i\alpha}$ 为鞍点方程

$$\frac{\partial S(t)}{\partial t} = \frac{1}{2}(\boldsymbol{p}+\boldsymbol{A}(t))^2 + I_p = \frac{1}{2}\left[p_z + A_0 \sin(\omega t)\sin^2\left(\frac{\omega t}{4}\right)\right]^2 + I_p = 0$$

的根. 忽略虚部 $t_i < 0$ 的非物理的根，则在区间 $t_i = 0,\, t_f = 4\pi/\omega$ 区间内一共有 6 个物理的鞍点.

(1) 选择合适的非线性求根方法，求 $p_z = 1\,(\mathrm{a.u.})$ 时鞍点方程的 6 个解 $t_{s1} \sim t_{s6}$.

(2) 在 $p_z \in [0.01,\, 2]$ 区间上，取间隔 $\Delta p_z = 0.01$，给出鞍点近似得到的能谱即 $\left|M_{\boldsymbol{p}}^{0}(t_f,\, t_i)_{\mathrm{SPM}}\right|^2$ 与 E_k 的关系曲线.

(3) 利用直接积分的方法求 $\left|M_{\boldsymbol{p}}^{0}(t_f,\, t_i)_{\mathrm{DI}}\right|^2$ 与 E_k 的关系曲线.

2.1 解题思路

题目中的鞍点方程是非线性复数方程，我们选取 Müller 法进行求解. 该方法考虑通过三个给定初始点 $(z_0,\, f(z_0)),\, (z_1,\, f(z_1)),\, (z_2,\, f(z_2))$ 的二次曲线与 z 平面的交点作为下一个近似值 z_3，再用 $z_1,\, z_2,\, z_3$ 及相应函数值作为新的初始值重复此过程，直到三个点接近一个函数零点，达到合适的精度为止. 由于本算例要求解实部在 $[0,\, 4\pi/\omega]$ 上的 6 个复根，可以不断选取不同的初始点使用 Müller 算法来求根，直到找到 6 个满足要求的根为止.

对于直接积分的算法，我们可采取最简单的 $n = 1$ 的牛顿–柯斯特积分方法求解，也即将积分简化为求一组梯形面积之和即可.

2.2 算法提纲

Müller 算法的伪代码与将以上基本一致，记录如下：

输入初始点 t_0, t_1, t_2
$i = 1$ （循环次数）
while $i \leq N_{\max}$ **do**
　　$h_0 = t_0 - t_2$; $h_1 = t_1 - t_2$; $\delta_0 = \frac{f(t_0)-f(t_2)}{h_0}$; $\delta_1 = \frac{f(x_1)-f(x_2)}{h_1}$; $d = \frac{\delta_0-\delta_1}{h_0-h_1}$
　　$b = \delta_0 - h_0 d$; $D = \sqrt{b^2 - 4f(t_2)d}$ （二次函数系数相关计算）
　　if $(|b - D| < |b + D|)$ **then**
　　　　$x_3 = x_2 + \frac{-2f(x_2)}{b+D}$
　　else
　　　　$x_3 = x_2 + \frac{-2f(x_2)}{b-D}$ （找接近根的零点）
　　if $|h| < 1 \times 10^5$ **then**
　　　　return x_3 （精度达到要求，找到近似根）
　　$x_0 = x_1$; $x_1 = x_2$; $x_2 = x_3$ （替换初始值，进行新一轮迭代）
end while

值得指出，由鞍点方程的性质可以首先对根做一判断：考虑到它是实数方程，因此对每个根 t_s，其复共轭 t_s^* 也是方程的根，所以若找到了虚部小于 0 的根，可不必将其略去，而考察其复共轭即可. 此外，容易发现对任意实数 t 有 $f(t) \geq I_p > 0$，因此鞍点方程的零点不可能为实数.

为能够找全 6 个物理的根，我们采用“广撒网”的方式：对 $[t_i, t_f]$ 区间作 $N = 201$ 等分，每次选取相邻的三个等分点作为初始值，利用 Müller 算法求根. 若返回根的虚部小于 0，则直接将其取复共轭，使其变成物理的符合要求的根. 需要注意，Müller 算法返回的值可能并不是原方程的根，故我们需将结果带回方程检验，以确保算法的正确性. 在一次“广撒网”的循环中，将每次得到的真实的，且与先前找到的互不相同的新的根放入列表中，最后判断是否找全 6 个根即可.

为计算最后的跃迁振幅，还需计算 $S''(t_s)$, $S(t_s)$ 的值. 考虑到 $S'(t)$ 只含三角函数，可以对其进行解析求解. 使用 Mathematica 进行解析求解的结果如下：

$$S''(t) = E_0 \sin\frac{\omega t}{4}\left(\frac{E_0}{\omega}\sin\omega t \sin\frac{\omega t}{4} + \frac{1}{2}\sin\omega t\cos\frac{\omega t}{4}\right)$$

$$\begin{aligned} S(t) = {} & \frac{3E_0^2 t}{32\omega^2} + \frac{1}{2}(2I_p + p_z)^2 t + \frac{p_z E_0}{6\omega^2}\left(-1 + 2\cos\frac{\omega t}{2} - 3\cos\omega t + \cos\frac{3\omega t}{2}\right) \\ & + \frac{E_0^2}{\omega^3}\left(-\frac{1}{4}\sin\frac{\omega t}{2} + \frac{1}{64}\sin\omega t + \frac{1}{24}\sin\frac{3\omega t}{2} - \frac{3}{64}\sin 2\omega t + \frac{1}{40}\sin\frac{5\omega t}{2} - \frac{1}{192}\sin 3\omega t\right) \end{aligned}$$

在代码初始编写阶段，也尝试过使用数值求积法计算 $S(t)$，所得结果差异不大，但为达到所需精度，所需计算时间更长. 故本程序中最终采取了解析计算 $S(t)$ 的方法.

2.3 结果与讨论

首先想说明的是，由于第 (1) 问待求参数点 $p_z = 1$ (a.u.) 包含在后两问的参数空间内，因此只需在一次求解过程中将 $p_z = 1$ 的结果单独输出即可，无需分不同小问各自编写代码.

本例中，运行程序 `02_saddle/02_saddle.py` 可以获得如下最终输出结果

```
第(1)问
p_z = 1.0 时，鞍点方程的 6 个解为
(24.059818099446375+119.54353926969002j)
(73.87985704437483+108.16876190566282j)
(263.98337385783367+36.807849485520784j)
(416.73193730586644+27.879798113027242j)
(677.1009140604434+64.20228674144018j)
(750.6437460397864+84.50511547794287j)

第(2)，(3)问： 参见两个图：两种方法的曲线比较 & 整体因子曲线
```

并获得 $p_z \in [0.01, 2]$ (a.u.) 区间内，以 $\Delta p_z = 0.01$ (a.u.) 为步长的参数空间内，用鞍点近似（SPM）与直接积分（DI）法求得的电子能谱 $\left|M^0_{\boldsymbol{p}}(t_f, t_i)_{\mathrm{SPM}}\right|^2$ 与 $\left|M^0_{\boldsymbol{p}}(t_f, t_i)_{\mathrm{DI}}\right|^2$ 随动能 $E_k = \frac{p_z^2}{2m_e}$ 的关系曲线图（绘制于一张图上）以及二者之比作为整体因子的变化曲线. 如图 4 与 5 所示.

最后对所得结果进行说明. 由图 4 可见两种方法各自计算的曲线振荡方式相似，都是在振荡过程中下降. 电子的末态获得动能 E_k 越大，其发生的概率越小，这也是符合实际的. 图 5 表明整体因子介于 (1, 2) 之间，大约维持在 1.30 附近，但也会随着能谱的振荡呈现出轻微的振荡. 分析认为这应当与本程序的计算误差无关（相对误差应在 10^{-5} 量级），而是由鞍点法存在系统误差导致的，是无法避免的.

2.4 源代码

第 (1)–(3) 问代码

第 (1)–(3) 问代码：`02_saddle/02_saddle.py`

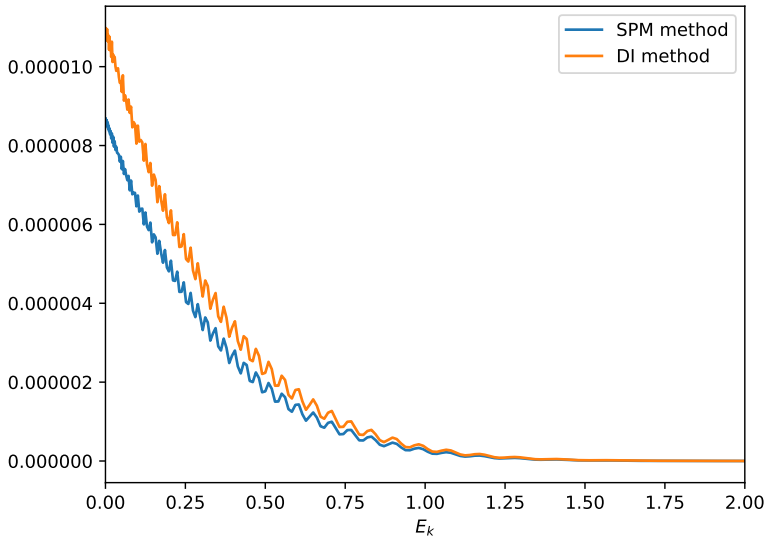

图 4: 用鞍点近似（SPM）与直接积分（DI）法求得的电子能谱 $\left|M_{\boldsymbol{p}}^{0}(t_f,\, t_i)_{\mathrm{SPM}}\right|^2$ 与 $\left|M_{\boldsymbol{p}}^{0}(t_f,\, t_i)_{\mathrm{DI}}\right|^2$ 随动能 E_k 的关系曲线图的比较.

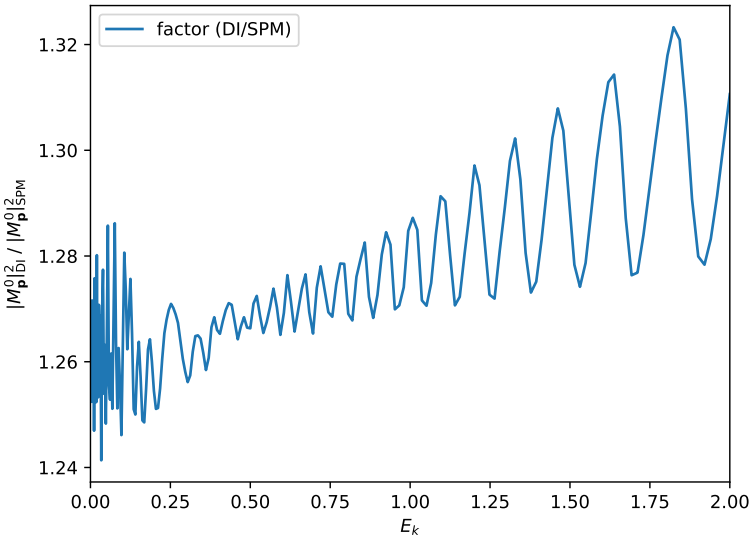

图 5: 用鞍点近似（SPM）与直接积分（DI）法求得的电子能谱 $\left|M_{\boldsymbol{p}}^{0}(t_f,\, t_i)_{\mathrm{SPM}}\right|^2$ 与 $\left|M_{\boldsymbol{p}}^{0}(t_f,\, t_i)_{\mathrm{DI}}\right|^2$ 之比（也即整体因子）随动能 E_k 的关系曲线图.

```
# -*- coding: utf-8 -*-
from cmath import sin, cos, sqrt, exp, pi  # 复数运算基本函数
```

```
def sort_real(array):  # 对数列的实部排序 (冒泡)
    def swap(a, b):
        return b, a
    for i in range(len(array)-1, 0, -1):
        for j in range(i):
            if array[j].real > array[j+1].real:
                array[j], array[j+1] = swap(array[j], array[j+1])
    return array

def dSfunc(t): # S'函数的解析形式
    return 0.5*(E0/ome * sin(ome*t) * sin(ome*t/4)**2 + pz)**2 + Ip

def ddSfunc(t): # S''函数的解析形式: 手动求导得到
    return E0*sin(ome*t/4) * (E0/ome * sin(ome*t) * sin(ome*t/4)**2 + pz)
    \
            * (cos(ome*t)*sin(ome*t/4) + 0.5*sin(ome*t)*cos(ome*t/4))

def Sfunc(t): # S函数的解析形式: 手动积分得到
    return 1/(ome**3)* (3/32*E0**2 *ome*t + 1/2*(2*Ip+pz**2)*ome**3 *t \
            + 1/6*pz*E0*ome*(3*cos(ome*t/2) - 3*cos(ome*t) +
    cos(3*ome*t/2))
            + E0**2 *(-1/4*sin(ome*t/2) + 1/64*sin(ome*t) +
    1/24*sin(3*ome*t/2) \
            - 3/64*sin(2*ome*t) + 1/40*sin(5*ome*t/2) -
    1/192*sin(3*ome*t))) \
            - E0*pz/(6*ome**2)

def muller_findroot(x0, x1, x2): # Muller求根法, 与讲义的格式相同
    it = 2
    while it <= 100:
        h0 = x0 - x2  # 以下计算二次函数相关变量
        h1 = x1 - x2
        del0 = (dSfunc(x0)-dSfunc(x2)) / h0
        del1 = (dSfunc(x1)-dSfunc(x2)) / h1
        d = (del0 - del1) / (h0 - h1)
```

```
        b = del0 - h0 * d
        D = sqrt(b**2 - 4*dSfunc(x2)*d)
        x3 = x2 - 2*dSfunc(x2)/(b+D) if abs(b-D) < abs(b+D) else x2 -
    2*dSfunc(x2)/(b-D)  # 距离零点较近的根
        if abs(x3-x2) < 1e-5: # 精度够了, 返回零点
            return x3
        x0 = x1  # 替换初始值
        x1 = x2
        x2 = x3
        it += 1
    else:
        return 'null'  # 循环大于100次, 没找到根

#  下面开始
print('先计算, 再输出各小问结果')
ome = 45.5633525316 / 3200  # 题中的常数
E0 = sqrt(5 / 3509.4448314)
Ip = 13.6 / 27.2
tmax = 4*pi/ome
pzstep = 0.01
pzlist = [0.01+pzstep*i for i in range(int(2./pzstep))]  # pz的参数点

## 下面用鞍点法 (SPM) 求解跃迁振幅
Mp02list_SPM = []  # 储存求得的跃迁振幅模方
N = 201 # 找 N 次以确定 6 个不同的根
ipz = 1  # pz的计数
for pz in pzlist:  # 遍历每个pz参数点
    solu = []
    for i in range(N-1):
        try:
            ts = muller_findroot(tmax/N*i, tmax/N*(i+1), tmax/N*(i+2))  #
    Muller法求根!
        except:  # 如果求根过程中报错 (比如除以0) 直接跳过本次求根
            continue
        if ts=='null' or ts.real < 0 or ts.real > tmax or
    abs(dSfunc(ts))>1e-2:  #
```

```
    没找到根，根出界了，或找到了错根，都直接跳过本次求根
            continue

        # 运行到此处说明找到了真实的根
        ts = ts.conjugate() if ts.imag < 0 else ts  # 如果虚部 <0
    则取共轭，共轭也是真实的根
        for s in solu:
            if abs(ts.real-s.real)<1e-4 : #
    和已经存入的物理的根相同吗？相同说明已经找到过它了，不存入solu
                break
        else:
            solu.append(ts) # 没有找到过，将其存入solu

    if len(solu) != 6: # 是不是找到了 6 个根？如果不是说明出错了
        print(pz, '不是 6 个根!')
    if pz == 1.: # 刚好把第一问结果存一下，一会输出
        solu_pz1 = sort_real(solu)

    Mp0 = 0  # 下面代入鞍点求解的公式
    for tsa in solu:
        Mp0 += exp(1j*Sfunc(tsa)) /ddSfunc(tsa)  # 依次求和
    exp(iS(t))/S''(t)
    Mp0 *= -(2*Ip)**(5/4) / sqrt(2)
    Mp02list_SPM.append(abs(Mp0)**2)  #
    求解完毕，将振幅模方存入Mp02list_SPM

    if ipz % 20 == 0:
        print('正在鞍点法 (SPM) 求解： [%-3d / %-3d]  正在进行中...'%(ipz,
    int(2./pzstep)))
    ipz += 1

## 下面直接积分 (DI) 求解跃迁振幅
pzstep = 0.01
pzlist = [0.01+pzstep*i for i in range(int(2./pzstep))]
Mp02list_DI = []
ipz = 1
```

```
for pz in pzlist:
    tstep = 0.1
    I = 0
    ti = tstep/2
    while ti < tmax:
        qz = pz + E0/ome * sin(ome*ti) * sin(ome*ti/4)**2
        I += qz*E0*(cos(ome*ti)*sin(ome*ti/4)**2 +
   1/4*sin(ome*ti)*sin(ome*ti/2)) \
             /(pi*(qz**2 + 2*Ip)**3) * exp(1j*Sfunc(ti)) * tstep
        ti += tstep
    I *= 2**(7/2) * (2*Ip)**(5/4)
    Mp02list_DI.append(abs(I)**2)

    if ipz%20==0:
        print('正在直接积分法 (DI) 求解:   [%-3d / %-3d]
    正在进行中...'%(ipz, int(2./pzstep)))
    ipz += 1

print('计算完毕, 下面输出各小问结果')
print('-------------------------------------------\n')
print('第(1)问\np_z = 1.0 时, 鞍点方程的 6 个解为')
for z in solu_pz1:
    print(z)
print('\n第(2), (3)问:  参见两个图: 两种方法的曲线比较 & 整体因子曲线')
########################## 使用 matplotlib 画图 ##########################
from matplotlib import pyplot as plt
plt.plot([pz**2/2 for pz in pzlist], Mp02list_SPM, label='SPM method') #
    画 SPM 振幅模方曲线
plt.plot([pz**2/2 for pz in pzlist], Mp02list_DI, label='DI method')  #
    画 DI 振幅模方曲线
plt.xlabel(r'$E_k$')
plt.ylabel(r'$|M_\mathbf{p}^0|^2$')
plt.legend()
plt.xlim(0., 2.)
plt.savefig('03_SPM_DI.pdf')
plt.show()
```

```
# 再画下整体因子曲线看看
plt.figure()
plt.plot([pz**2/2 for pz in pzlist], [Mp02list_DI[i]/Mp02list_SPM[i] for
    i in range(len(pzlist))], label='factor (DI/SPM)')
plt.xlabel(r'$E_k$')
plt.ylabel(r'$|M_\mathbf{p}^0|^2_{\mathrm{DI}}\;/\; \
           |M_\mathbf{p}^0|^2_{\mathrm{SPM}}$')
plt.legend()
plt.xlim(0., 2.)
plt.savefig('03_factor.pdf')
plt.show()
```

3 随机游走过程

题目：随机游走过程的一系列理论推演与程序模拟计算.

(1) 一维随机游走过程：向左概率为 p，向右概率为 q，求走了 *3* 步以后离原点距离 x 的概率 $P(x)$ 及 x 的期望值.

(2) 二维随机游走，向上、下、左、右概率相同，编写程序输出游走 *50* 步的每步位置与距离 R.

(3) 二维随机游走（以下同），任意方向行走，步长为 *1*，夹角随机，输出游走 *100* 步每一步的距离 R 与期望值 $\mathbb{E}(R)$，作 $\mathbb{E}(R)$ 与步数 n 的散点图，用 $E(R) = a\sqrt{n}$ 拟合.

(4) 解析证明随机变量 X_n 与 Y_n 期望值 *0*，计算方程.

(5) 说明 X_n 与 Y_n 是否相关？是否独立？

(6) 解释 n 较大时 (X_n, Y_n) 联合概率密度分布可表示为两个独立正态分布的乘积的形式（用到 *(4), (5)* 问）.

(7) 推导 R 分布函数与概率密度函数.

(8) 计算 $\mathbb{E}(R_n)$，对比程序结果.

3.1 解题思路

本算例中设计代码实现的只有 (2), (3) 两问. 其余小问的理论分析与推导请直接参见“结果与讨论”一节.

第 (2) 问程序实现很简单，需用 Python 的 `random` 库生成 $[0, 1)$ 之间的随机数，然后将区间四等分可转化为概率相同的四种情形，对应向上、下、左、右移动. 执行随机行走 50 步，过程中记录轨迹并计算位移即可. 第 (3) 问需要输出 $[0, 2\pi)$ 区间内的随机变量作为单次行走的角度 θ，这可以直接用 $[0, 1)$ 间的随机数乘以 2π 实现，其余实现方法类似. 本程序共进行了 10000 次试验，统计第 n 步（$n = 1, \cdots, 100$）达到位置到原点距离 R_n 的去平均，作为 $E(R_n)$ 的实验估计值. 按照要求进行拟合即可（拟合直接使用了 `scipy.optimize` 库的 `curve_fit` 函数，未自主编写代码）.

3.2 算法提纲

(2), (3) 问有关程序实现部分的算法提纲已在上一节中介绍，算法简明，代码直接. 请看下节的结果与分析.

3.3 结果与讨论

以下内容为每一问的解析推导与相应的程序运行结果.

第 (1) 问

采用枚举法，对每一种步行的概率进行计算，请见下表所示.

步型 $\{\cdot,\cdot,\cdot\}$	$\{-,-,-\}$	$\{-,-,+\}$	$\{-,+,-\}$	$\{-,+,+\}$
概率 P	p^3	p^2q	p^2q	pq^2
距离 x	3	1	1	1
步型 $\{\cdot,\cdot,\cdot\}$	$\{+,-,-\}$	$\{+,-,+\}$	$\{+,+,-\}$	$\{+,+,+\}$
概率 P	p^2q	pq^2	pq^2	q^3
距离 x	1	1	1	3

容易得到 x 的概率分布为

$$P(x)=\begin{cases}p^3+q^3, & x=3\\ 3p^2q+3pq^2, & x=1\\ 0, & \text{其他}\end{cases}$$

x 的期望值为

$$\mathbb{E}(x)=\sum_x xP(x)=3(p^3+q^3+p^2q+pq^2)=2(p^3+q^3)+1$$

最后一个等号用到了 $p+q=1$. 当然，还有其他的等价表述，这里不再一一展示.

第 (2) 问

运行程序 `03_random_walk/03_1_grid.py` 可得如下输出结果:

```
第  1 步,  ( X, Y ) = (  0,  -1 ),  R = 1.000000
第  2 步,  ( X, Y ) = (  0,  -2 ),  R = 2.000000
第  3 步,  ( X, Y ) = ( -1,  -2 ),  R = 2.236068
第  4 步,  ( X, Y ) = ( -2,  -2 ),  R = 2.828427
第  5 步,  ( X, Y ) = ( -2,  -1 ),  R = 2.236068
第  6 步,  ( X, Y ) = ( -3,  -1 ),  R = 3.162278
第  7 步,  ( X, Y ) = ( -3,  -2 ),  R = 3.605551
第  8 步,  ( X, Y ) = ( -4,  -2 ),  R = 4.472136
第  9 步,  ( X, Y ) = ( -5,  -2 ),  R = 5.385165
```

```
第 10 步,  ( X, Y ) = ( -5,  -1 ),  R = 5.099020
第 11 步,  ( X, Y ) = ( -5,  -2 ),  R = 5.385165
第 12 步,  ( X, Y ) = ( -5,  -1 ),  R = 5.099020
第 13 步,  ( X, Y ) = ( -6,  -1 ),  R = 6.082763
第 14 步,  ( X, Y ) = ( -6,  -2 ),  R = 6.324555
第 15 步,  ( X, Y ) = ( -6,  -1 ),  R = 6.082763
第 16 步,  ( X, Y ) = ( -7,  -1 ),  R = 7.071068
第 17 步,  ( X, Y ) = ( -7,  -2 ),  R = 7.280110
第 18 步,  ( X, Y ) = ( -7,  -3 ),  R = 7.615773
第 19 步,  ( X, Y ) = ( -7,  -4 ),  R = 8.062258
第 20 步,  ( X, Y ) = ( -7,  -5 ),  R = 8.602325
第 21 步,  ( X, Y ) = ( -8,  -5 ),  R = 9.433981
第 22 步,  ( X, Y ) = ( -8,  -4 ),  R = 8.944272
第 23 步,  ( X, Y ) = ( -8,  -5 ),  R = 9.433981
第 24 步,  ( X, Y ) = ( -8,  -4 ),  R = 8.944272
第 25 步,  ( X, Y ) = ( -9,  -4 ),  R = 9.848858
第 26 步,  ( X, Y ) = ( -8,  -4 ),  R = 8.944272
第 27 步,  ( X, Y ) = ( -9,  -4 ),  R = 9.848858
第 28 步,  ( X, Y ) = ( -9,  -3 ),  R = 9.486833
第 29 步,  ( X, Y ) = ( -9,  -4 ),  R = 9.848858
第 30 步,  ( X, Y ) = ( -9,  -5 ),  R = 10.295630
第 31 步,  ( X, Y ) = (-10,  -5 ),  R = 11.180340
第 32 步,  ( X, Y ) = (-10,  -6 ),  R = 11.661904
第 33 步,  ( X, Y ) = (-11,  -6 ),  R = 12.529964
第 34 步,  ( X, Y ) = (-11,  -7 ),  R = 13.038405
第 35 步,  ( X, Y ) = (-11,  -6 ),  R = 12.529964
第 36 步,  ( X, Y ) = (-10,  -6 ),  R = 11.661904
第 37 步,  ( X, Y ) = (-10,  -5 ),  R = 11.180340
第 38 步,  ( X, Y ) = (-11,  -5 ),  R = 12.083046
第 39 步,  ( X, Y ) = (-11,  -6 ),  R = 12.529964
第 40 步,  ( X, Y ) = (-11,  -5 ),  R = 12.083046
第 41 步,  ( X, Y ) = (-10,  -5 ),  R = 11.180340
第 42 步,  ( X, Y ) = (-11,  -5 ),  R = 12.083046
第 43 步,  ( X, Y ) = (-11,  -6 ),  R = 12.529964
第 44 步,  ( X, Y ) = (-10,  -6 ),  R = 11.661904
第 45 步,  ( X, Y ) = (-11,  -6 ),  R = 12.529964
```

```
第 46 步,  ( X, Y ) = (-11,  -7 ),  R = 13.038405
第 47 步,  ( X, Y ) = (-10,  -7 ),  R = 12.206556
第 48 步,  ( X, Y ) = (-10,  -6 ),  R = 11.661904
第 49 步,  ( X, Y ) = ( -9,  -6 ),  R = 10.816654
第 50 步,  ( X, Y ) = ( -9,  -7 ),  R = 11.401754
```

输出内容分别是随机行走的每一步 (X, Y) 的坐标以及到原点的距离 R. 需要指出，由于程序中使用了相同的种子产生随机数：`random.seed(0)`，故每次的运行结果应该都与上面展示的相同.

同时，程序还输出了将这 50 步随机行走进行可视化的坐标图，如图 6 所示.

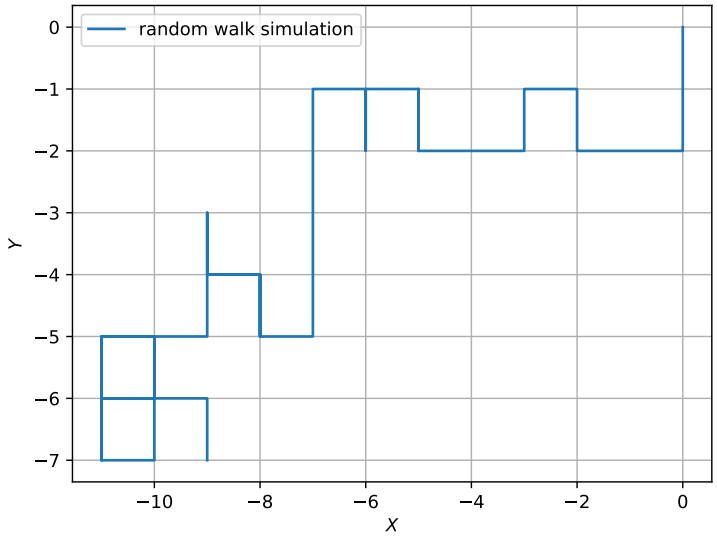

图 6: 随机行走 50 步过程中的可视化坐标图.

第 (3) 问

本小问需要求解每步行走方向 θ 可随机为 $[0, 2\pi)$ 的情形下的到原点距离 R 与随机行走次数 n 的关系. 运行程序 `03_random_walk/03_2_rtheta.py`，可得到模拟 20000 次过程中，给定行走步数下距离的期望值 $\mathbb{E}(R)$ 随 n 的变化关系，并利用 $R = a\sqrt{n}$ 函数使用 `scipy.optimize` 库进行了曲线拟合. 输出图片请见图 7

最终计算 a 的结果为 0.88674. 后文将进行与理论结果的比较.

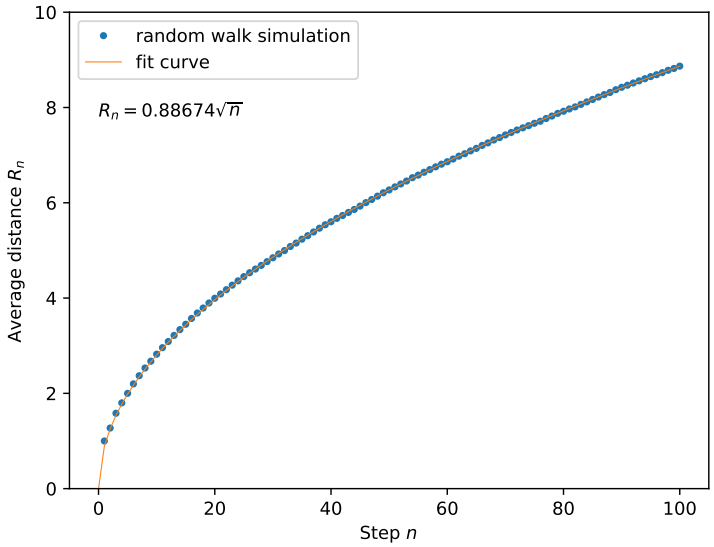

图 7: 模拟 20000 次过程中，获得的行走步数为 n 时距离的期望值 $\mathbb{E}(R)$ 随 n 的变化关系图.

第 (4) 问

依照定义，第 n 步后的位置坐标 (X_n, Y_n) 满足

$$X_n = \sum_{k=1}^{n} \cos\theta_k, \quad Y_n = \sum_{k=1}^{n} \sin\theta_k$$

其中 θ_k 变量服从 $[0, 2\pi)$ 区间的均匀分布，概率密度函数为

$$f(\theta_k) = \frac{1}{2\pi}, \quad \theta \in [0, 2\pi)$$

因此 X_n, Y_n 的期望值为

$$\mathbb{E}(X_n) = \mathbb{E}\left(\sum_{k=1}^{n} \cos\theta_k\right) = \sum_{k=1}^{n} \mathbb{E}(\cos\theta_k) = \sum_{k=1}^{n} \int_0^{2\pi} \frac{1}{2\pi} \cos\theta_k \, \mathrm{d}\theta_k = 0$$

$$\mathbb{E}(Y_n) = \mathbb{E}\left(\sum_{k=1}^{n} \sin\theta_k\right) = \sum_{k=1}^{n} \mathbb{E}(\sin\theta_k) = \sum_{k=1}^{n} \int_0^{2\pi} \frac{1}{2\pi} \sin\theta_k \, \mathrm{d}\theta_k = 0$$

X_n 的方差为

$$\begin{aligned}\mathbb{D}(X_n) &= \mathbb{E}(X_n^2) - [\mathbb{E}(X_n)]^2 = \mathbb{E}\left[\left(\sum_{k=1}^{n}\cos\theta_k\right)^2\right] \\ &= \mathbb{E}\left(\sum_{k=1}^{n}\cos^2\theta_k + 2\sum_{k<m}\cos\theta_k\cos\theta_m\right) \\ &= \sum_{k=1}^{n}\mathbb{E}(\cos^2\theta_k) + 2\sum_{k<m}\mathbb{E}(\cos\theta_k\cos\theta_m)\end{aligned}$$

由于第 k 次与第 m 次完全独立，因此可继续写作

$$\begin{aligned}\mathbb{D}(X_n) &= \sum_{k=1}^{n}\mathbb{E}(\cos^2\theta_k) + 2\sum_{k<m}\mathbb{E}(\cos\theta_k)\mathbb{E}(\cos\theta_m) \\ &= \sum_{k=1}^{n}\int_0^{2\pi}\frac{1}{2\pi}\cos^2\theta_k\,\mathrm{d}\theta_k \\ &= \sum_{k=1}^{n}\frac{1}{2} = \frac{n}{2}\end{aligned}$$

利用相同的思路容易得到计算 Y_n 的方差，只需将上式所有 $\cos(\cdot)$ 替换为 $\sin(\cdot)$ 即可，结果也为 $\frac{n}{2}$. 还有另一种论证方法，即题中的随机行走模式是对于 $[0,\,2\pi)$ 中各个方向地位完全是均等的，故任意方向上随机行走的投影值的方差均与 X_n 的方差相同，为 $\frac{n}{2}$.

第 (5) 问

计算 X_n 与 Y_n 的协方差：

$$\begin{aligned}\operatorname{cov}(X_n,\,Y_n) &= \mathbb{E}\big[(X_n - \mathbb{E}(X_n))(Y_n - \mathbb{E}(Y_n))\big] = \mathbb{E}(X_nY_n) \\ &= \mathbb{E}\left[\left(\sum_{k=1}^{n}\cos\theta_k\right)\left(\sum_{k=1}^{n}\sin\theta_k\right)\right] \\ &= \mathbb{E}\left(\sum_{k=1}^{n}\cos\theta_k\sin\theta_k\right) + \mathbb{E}\left(\sum_{k\neq m}\cos\theta_k\sin\theta_m\right) \\ &= \sum_{k=1}^{n}\mathbb{E}(\cos\theta_k\sin\theta_k) + \sum_{k\neq m}\mathbb{E}(\cos\theta_k)\mathbb{E}(\sin\theta_m) \\ &= \sum_{k=1}^{n}\int_0^{2\pi}\cos\theta_k\sin\theta_k\,\mathrm{d}\theta_k + 0 = \sum_{k=1}^{n}0 = 0\end{aligned}$$

上式也用到了第 k 次与第 m 次完全独立的性质. 计算表明协方差为 0，说明 X_n 与 Y_n 是非相关的.

但是，我们知道非相关性是独立性的必要非充分条件，因此不能论证二者也是独立的. 事实上，二者并不独立. 取 $n=1$ 时为例，$X_1=\cos\theta_1$, $Y_1=\sin\theta_1$. 容易计算

$$\mathrm{P}\left(X_1>\frac{\sqrt{2}}{2}\right)=\frac{1}{2},\quad \mathrm{P}\left(Y_1>\frac{\sqrt{2}}{2}\right)=\frac{1}{2},$$

而

$$\mathrm{P}\left(\left(X_1>\frac{\sqrt{2}}{2}\right)\wedge\left(Y_1>\frac{\sqrt{2}}{2}\right)\right)=0\neq\mathrm{P}\left(X_1>\frac{\sqrt{2}}{2}\right)\mathrm{P}\left(Y_1>\frac{\sqrt{2}}{2}\right)$$

这说明 X_1 与 Y_1 不是独立的.

第 (6) 问

笔者经过仔细思考，认为无法顺承前两问的结果，论证 $n\to\infty$ 时 X_n 与 Y_n 相互独立，可写作两个独立正态分布的乘积.

一种看似有道理的解释是这样的. 根据大数定理与中心极限定理，由于 $\mathbb{E}(X_n)=\mathbb{E}(Y_n)=0$, $\mathbb{D}(X_n)=\mathbb{D}(Y_n)=\frac{n}{2}$, $n\to\infty$ 时 X_n、Y_n 都将收敛与正态分布 $\mathrm{N}(0,\frac{n}{2})$. 通过两个正态分布的联合分布仍为正态分布这一结论，联合正态分布中两个变量的独立性由参数 ρ 给出，其中

$$\rho=\frac{\operatorname{cov}(X_n,\,Y_n)}{\sqrt{\mathbb{D}(X_n)\mathbb{D}(Y_n)}}=0$$

所以在 $n\to\infty$ 时 X_n, Y_n 是趋向于独立的.

但笔者认为，以上的论述使用了“两个正态分布的联合分布仍为正态分布”这一看似显然的结论，但它却并不是显然成立. 若我们已知 X_n, Y_n 联合分布是一个二维正态分布，那么 X_n, Y_n 必然各自满足一维正态分布. 但是，上述的逆命题是否成立呢? 也即，是否存在一个联合分布 $f(X_n, Y_n)$，使得它在 X_n, Y_n 上的投影也分别为正态分布呢? 笔者没有发现任何已知定理支持这一结论.

因此，我们应采取稍复杂的方法论证上述的命题. 我们可以从中心极限定理的证明为切入点，仿照这一证明过程说明“二维”情形下也有类似的中心极限定理，即: **对符合任意概率密度分布 $f(X_n, Y_n)$ 的二维随机变量 (X_n, Y_n)，$n\to\infty$ 时 $(\bar{X}, \bar{Y})$ 符合二维正态分布**. 只有用类似这种更高级的方法论证了联合分布的确是二维正态分布，才可以通过 $\rho=0$ 来说明两个变量的独立性. 对这一更高级的命题的论证如下.

证明:

对于任意概率密度 $f(x, y)$ 的二维随机变量分布，定义其二维特征函数为

$$\varphi(t,\,s)=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\mathrm{e}^{\mathrm{i}(tx+sy)}f(x,\,y)\,\mathrm{d}x\,\mathrm{d}y\,.$$

特征函数的优势在于，可以将样本空间中概率密度的卷积转化为特征函数空间中的乘积. 例如，采集两次样本后 $(x,\,y)=(x_1,\,y_1)+(x_2,\,y_2)$ 符合的概率密度分布应当为

$$F(x,\,y)=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}f(x_1,\,y_1)f(x-x_1,\,y-y_1)\,\mathrm{d}x_1\,\mathrm{d}y_1$$

它的特征函数为

$$\begin{aligned}\Phi(t,\,s)&=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\mathrm{d}x\,\mathrm{d}y\,\mathrm{e}^{\mathrm{i}(tx+sy)}\left(\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}f(x_1,\,y_1)f(x-x_1,\,y-y_1)\,\mathrm{d}x_1\,\mathrm{d}y_1\right)\\&=\left(\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\mathrm{d}x_1\,\mathrm{d}y_1\,\mathrm{e}^{\mathrm{i}(tx_1+sy_1)}f(x_1,\,y_1)\right)\cdot\\&\qquad\left(\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\mathrm{d}(x-x_1)\,\mathrm{d}(y-y_1)\,\mathrm{e}^{\mathrm{i}(t(x-x_1)+s(y-y_1))}f(x-x_1,\,y-y_1)\right)\\&=[\varphi(t,\,s)]^2\end{aligned}$$

不失一般性，我们认为 $\mathbb{E}(x)=\mathbb{E}(y)=0$（若不成立只需做整体平移即可），对于 n 次取样，可定义变量

$$X=\frac{1}{\sigma_x/\sqrt{n}}\frac{\sum_{k=1}^{n}x_k}{n},\quad Y=\frac{1}{\sigma_y/\sqrt{n}}\frac{\sum_{k=1}^{n}y_k}{n}$$

则 $(X,\,Y)$ 随机变量的特征函数为

$$\Psi(t,\,s)=\left[\varphi\left(\frac{t}{\sigma_x/\sqrt{n}},\,\frac{s}{\sigma_y/\sqrt{n}}\right)\right]^n$$

由于

$$\frac{\partial\varphi(0,\,0)}{\partial t}=\mathbb{E}(x)=0,\quad\frac{\partial\varphi(0,\,0)}{\partial s}=\mathbb{E}(y)=0$$

$$\frac{\partial^2\varphi(0,\,0)}{\partial t^2}=-\mathbb{D}(x)=-\sigma_x^2,\quad\frac{\partial^2\varphi(0,\,0)}{\partial s^2}=-\mathbb{D}(y)=-\sigma_y^2,\quad\frac{\partial\varphi(0,\,0)}{\partial t\partial s}=-\mathrm{cov}(x,\,y)=-\rho\sigma_x\sigma_y$$

最后的 ρ 可视作定义式，因此可将 $\varphi(t,\,s)$ 写作泰勒级数和

$$\varphi(t,\,s)=1-\frac{1}{2}\sigma_x^2t^2-\frac{1}{2}\sigma_y^2s^2-\rho\sigma_x\sigma_yts+\text{高阶项}$$

因此

$$\begin{aligned}\Psi(t,\,s)&=\left[\varphi\left(\frac{t}{\sigma_x/\sqrt{n}},\,\frac{s}{\sigma_y/\sqrt{n}}\right)\right]^n=\left[1-\frac{1}{2n}t^2-\frac{1}{2n}s^2-\frac{1}{n}\rho ts+\mathcal{O}\left(\frac{1}{n^2}\right)\right]^n\\&\xrightarrow{n\to\infty}\exp\left(-\frac{t^2}{2}-\frac{s^2}{2}-\rho ts\right)\end{aligned}$$

这说明 $(X,\,Y)$ 变量在 $n\to\infty$ 时的确符合耦合参量为 ρ 的二维正态分布，证毕.

在上述定理的帮助下，我们可由 $\rho = 0$ 得到 $(X_n,\, Y_n)$ 联合分布应为两个 $\mathrm{N}(0,\, \frac{n}{2})$ 的正态分布的乘积. 因此

$$f(X_n,\, Y_n) = \frac{1}{2\pi \cdot \frac{n}{2}} \mathrm{e}^{\frac{X_n^2+Y_n^2}{2\cdot\frac{n}{2}}} = \frac{1}{\pi n} \mathrm{e}^{-\frac{X_n^2+Y_n^2}{n}}$$

第 (7) 问

将上式概率密度函数换作极坐标表述，可知

$$1 = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(X_n,\, Y_n)\,\mathrm{d}X_n\,\mathrm{d}Y_n = \int_0^{\infty} \frac{1}{\pi n}\mathrm{e}^{-R_n^2/n}\, 2\pi R_n\,\mathrm{d}R_n = \int_0^{\infty} f_{R_n}(R_n)\,\mathrm{d}R_n$$

故可以看出 R_n 变量的概率密度为

$$f_{R_n}(R_n) = \frac{1}{\pi n}\mathrm{e}^{-R_n^2/n}\, 2\pi R_n = \frac{2R_n}{n}\mathrm{e}^{-R_n^2/n}$$

分布函数为

$$F_{R_n}(R_n) = \int_0^{R_n} f_{R_n}(R_n')\,\mathrm{d}R_n' = 1 - \mathrm{e}^{-R_n^2/n}, \quad R_n \in [0,\, \infty)$$

第 (8) 问

由上问结果，R_n 的期望值为

$$\begin{aligned}
\mathbb{E}(R_n) &= \int_0^{\infty} R_n f_{R_n}(R_n)\,\mathrm{d}R_n = \int_0^{\infty} \frac{2R_n^2}{n}\mathrm{e}^{-R_n^2/n} \\
&= -R_n\mathrm{e}^{-R_n^2/n}\Big|_0^{\infty} + \int_0^{\infty} \mathrm{e}^{-R_n^2/n}\,\mathrm{d}R_n \\
&= \frac{\sqrt{\pi}}{2}\cdot\sqrt{n}
\end{aligned}$$

上式用到了分部积分法以及高斯积分公式. 对照第 (3) 问，可知 $R_n = a\sqrt{n}$ 中 a 的理论值为

$$a = \frac{\sqrt{\pi}}{2} = 0.88622692545$$

与蒙特卡洛模拟的结果有三位有效数字重合. 相对误差计算为 $0.058\,\%$，非常接近.

3.4 源代码

第 (2) 问代码

第 (2) 问代码：`03_random_walk/03_1_grid.py`

```
# -*- coding: utf-8 -*-
from numpy import sin, cos, sqrt, pi  # 导入运算基本函数
import random  # Python生成随机数的工具

N = 50  # 步数
X = 0  # 初始位置
Y = 0
Xlist = [0]
Ylist = [0]
random.seed(0)  # 指定seed确保每次运行产生的随机数结果相同
for i in range(N):
    r = random.random()  # 产生[0,1)之间随机数
    if r <= 0.25:  # 区间四等分，分别对应X+1, X-1, Y+1, Y-1
        X += 1
    elif r <= 0.50:
        X -= 1
    elif r <= 0.75:
        Y += 1
    else:
        Y -= 1
    Xlist.append(X)
    Ylist.append(Y)
    print('第 %2d 步，  ( X, Y ) = (%3d, %3d ),  R = %f'%(i+1, X, Y,
    sqrt(X**2+Y**2)))


########################## 使用 matplotlib 画图 ##########################
from matplotlib import pyplot as plt  # 作图工具包
plt.plot(Xlist, Ylist, label='random walk simulation')
plt.xlabel(r'$X$')
plt.ylabel(r'$Y$')
plt.legend()
plt.grid()
plt.savefig('03_1.pdf')
plt.show()
```

第 (3) 问代码

第 (3) 问代码: `03_random_walk/03_2_rtheta.py`

```
# -*- coding: utf-8 -*-
from numpy import sin, cos, sqrt, pi  # 导入运算基本函数
import random  # Python生成随机数的工具

random.seed(0)  # 指定seed确保每次运行产生的随机数结果相同
NMC = 20000  # 蒙卡运行20000次
N = 100  # 随机行走100步
Rlist = []
for iMC in range(NMC):
    X = 0.
    Y = 0.
    Rsingle = []
    for i in range(N):
        theta = random.random() * 2*pi  # 生成[0,2 )随机数
        X += cos(theta)
        Y += sin(theta)
        Rsingle.append(sqrt(X**2 + Y**2))  # 储存100步中每步的R_n
    Rlist.append(Rsingle)  # 将每次蒙卡的Rsingle储存进来
    if iMC % 2000 == 0:
        print('正在进行随机行走模拟: [%-5d / %-5d] 进行中...'%(iMC, NMC))

R = [sum([Rlist[i][j] for i in range(NMC)])/NMC for j in range(N)]  #
    对10000次蒙卡下第n步的R_n取平均, 即为期望E(R_n)的实验值



########################## 使用 matplotlib 画图 ##########################
from matplotlib import pyplot as plt  # 作图工具包
from scipy.optimize import curve_fit  # 拟合工具包
import numpy as np
def func(x, a):
    return a * sqrt(x)
a_fit, cov = curve_fit(func, np.arange(1,N+1), np.array(R))
plt.plot(range(1,N+1), R, '.', label='random walk simulation')
```

```
plt.plot(range(N+1), [a_fit[0] * sqrt(i) for i in range(N+1)],
    linewidth=0.6, label='fit curve')
plt.xlabel(r'Step $n$')
plt.ylabel(r'Average distance $R_n$')
plt.legend()
plt.text(0, 7.8, r'$R_n = %.5f\sqrt{n}$'%a_fit[0])
plt.ylim([0, 100])
plt.ylim([0, 10])
plt.savefig('03_2.pdf')
plt.show()
```

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、矩阵的硬拷贝、全零与单位矩阵的生成、取出子阵与子阵块的粘贴，以及用好看的形式打印矩阵等功能函数. `mymatrixlib/matrix.py` 源代码如下：

```
# -*- coding: utf-8 -*-
from numpy import sqrt

# 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.elem = element[:] # 行向量, [[],[],[]]格式

    def __add__(self, other): # 重载+为矩阵加法
        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.elem[i][j] + other.elem[i][j] for j in
    range(self.ncolumn)] for i in range(self.nrow)])

    def __sub__(self, other): # 重载-为矩阵减法
        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.elem[i][j] - other.elem[i][j] for j in
    range(self.ncolumn)] for i in range(self.nrow)])
```

```
def __mul__(self, other): # 重载*为矩阵乘法
    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.elem[i][m] * other.elem[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): # 矩阵数乘
    return matrix([[n * self.elem[i][j] for j in range(self.ncolumn)]
for i in range(self.nrow)])

def T(self): # 矩阵转置
    return matrix([[self.elem[i][j] for i in range(self.nrow)] for j
in range(self.ncolumn)])

def mod2(self): # 向量的模方
    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.elem[0][i]**2 for i in range(self.ncolumn)])
    if self.ncolumn == 1:
        return sum([self.elem[i][0]**2 for i in range(self.nrow)])

def mod(self): # 向量的模
    ans = self.mod2()
    if ans == 'error':
        return 'error'
    else:
        return sqrt(ans)
```

```
    def copy(self): # 矩阵的自身复制
        return matrix([[self.elem[i][j] for j in range(self.ncolumn)] for
    i in range(self.nrow)])

    def paste_on(self, other, row_st=0, column_st=0): #
    将给定矩阵粘贴在原矩阵指定区域上
        for i in range(other.nrow):
            for j in range(other.ncolumn):
                self.elem[row_st+i][column_st+j] = other.elem[i][j]

    def slice(self, row_st, row_end, column_st, column_end): #
    取原矩阵的指定区域的子阵
        return matrix([[self.elem[i][j] for j in
    range(column_st,column_end)] for i in range(row_st,row_end)])

    def zeros(nrow, ncolumn): # 全零矩阵
        return matrix([[0. for j in range(ncolumn)] for i in range(nrow)])

    def identity(nrow): # 对角阵
        return matrix([[1. if i==j else 0. for j in range(nrow)] for i in
    range(nrow)])

    def print(self, n=3): # 按矩阵格式打印
        for i in range(self.nrow):
            print([round(self.elem[i][j], n) for j in
    range(self.ncolumn)])
        print('\n')
```