

计算物理第四次作业

第一题：计算矩阵本征值

解答

1. QR 算法

QR 算法通过反复进行 QR 分解 $A_k = Q_k R_k$ 并更新 $A_{k+1} = R_k Q_k$ 来使矩阵收敛到上三角矩阵，对于本题对称矩阵为对角矩阵。此处采用的收敛判断标准为非对角元素的绝对值和小于容差 10^{-6} 。

QR 分解采用Householder变换法，原理见课件。

核心代码如下：

```
1 def householder_qr(A: np.ndarray):
2     """
3         使用Householder变换进行QR分解
4         :param A: 输入矩阵(m*n, m≥n, 列满秩)
5         :return: 正交矩阵Q(m*m), 上三角矩阵R(m*n)
6     """
7     # 复制矩阵并转换为浮点型, 避免原矩阵被修改
8     m, n = A.shape
9     R = A.copy().astype(float)
10    # 初始化Q为m阶单位矩阵
11    Q = np.eye(m, dtype=float)
12
13    # 遍历每一列
14    for k in range(n):
15        # 提取当前列的子向量 (从第k行到最后一行)
16        x = R[k:, k].copy()
17        # 计算x的2-范数
18        norm_x = np.linalg.norm(x, ord=2)
19        # 如果范数为0, 说明该列已经是0, 跳过
20        if norm_x == 0:
21            continue
22
23        # 构造单位向量e1 (长度和x一致)
24        e1 = np.zeros_like(x)
25        e1[0] = 1.0
26        # 选择符号, 避免数值抵消 (和x[0]同号)
27        sign = np.sign(x[0]) if x[0] != 0 else 1.0
28        # 构造Householder向量v并归一化
29        v = x + sign * norm_x * e1
30        v /= np.linalg.norm(v, ord=2)
31
32        # 构造m-k阶的Householder矩阵H
33        h_size = len(v)
34        H = np.eye(h_size, dtype=float) - 2.0 * np.outer(v, v)
35
36        # 步骤1: 将Householder变换应用到R的子矩阵 (左乘H)
37        R[k:, k:] = H @ R[k:, k:]
```

```

38
39      # 步骤2: 构造m阶的完整Householder矩阵 (嵌入到单位矩阵的右下角)
40      H_full = np.eye(m, dtype=float)
41      H_full[k:, k:] = H
42
43      # 步骤3: 将完整的Householder矩阵应用到Q (右乘H_full)
44      Q = Q @ H_full
45
46      # 对于nxn矩阵, R的下三角部分 (除了上三角) 可以置0, 避免数值误差导致的小值
47      for i in range(m):
48          for j in range(n):
49              if i > j:
50                  R[i, j] = 0.0
51
52      return Q, R
53
54 def qr_algorithm(A, max_iter=50, tol=1e-6):
55     n = A.shape[0]
56     Ak = A.copy()
57     print(f"初始矩阵:\n{Ak}")
58
59     for k in range(1, max_iter + 1):
60         Q, R = householder_qr(Ak)
61         Ak = R @ Q
62
63         if k % 5 == 0:
64             print(f"\n第 {k} 次迭代:")
65             print(Ak)
66
67         # 检查收敛性, 判断非对角元素的绝对值和是否小于容差
68         off_diagonal = np.sum(np.abs(Ak)) - np.sum(np.abs(np.diag(Ak)))
69         if off_diagonal < tol:
70             print(f"\n{k} 次迭代后收敛。")
71             break
72
73     print("\nQR算法特征值:", np.diag(Ak))
74     return np.diag(Ak)

```

2. Jacobi 算法

Jacobi 算法通过一系列相似变换 (即旋转) 将对称矩阵变为实舒尔形式, 每次选择模最大的非对角元素进行消除。此处采用的收敛判断标准为**最大非对角元素的绝对值小于容差 10^{-8}** 。

核心代码如下:

```

1 def jacobi_algorithm(A, max_iter=1000, tol=1e-8):
2     n = A.shape[0]
3     Ak = A.copy()
4
5     for k in range(max_iter):
6         # 找到最大的非对角元素
7         max_val = 0.0
8         p, q = -1, -1

```

```

9     for i in range(n):
10        for j in range(i + 1, n):
11            if abs(Ak[i, j]) > max_val:
12                max_val = abs(Ak[i, j])
13                p, q = i, j
14
15    if max_val < tol:
16        print(f"\n{k}次迭代后收敛。")
17        break
18
19    # 计算旋转参数
20    if Ak[p, q] == 0:
21        c = 1.0
22        s = 0.0
23    else:
24        eta = (Ak[q, q] - Ak[p, p]) / (2 * Ak[p, q])
25        if eta >= 0:
26            t = 1.0 / (eta + np.sqrt(1 + eta**2))
27        else:
28            t = -1.0 / (-eta + np.sqrt(1 + eta**2))
29
30        c = 1.0 / np.sqrt(1 + t**2)
31        s = t * c
32
33    # 构造Jacobi旋转矩阵
34    J = np.eye(n)
35    J[p, p] = c
36    J[q, q] = c
37    J[p, q] = s
38    J[q, p] = -s
39
40    Ak = J.T @ Ak @ J
41
42    print("\nJacobi算法特征值:", np.diag(Ak))
43    return np.diag(Ak)

```

3. Sturm 序列 + 对分法

利用 Sturm 序列的性质计算给定区间内本征值的个数，结合对分法确定本征值。此处采用的收敛判断标准为区间宽度小于容差 10^{-6} 。算法原理见课件。

核心代码如下：

```

1 def sturm_sequence_count(d, e, lam):
2     """
3         计算Sturm序列在入处的符号变化次数
4         :param d: 矩阵的对角元素
5         :param e: 矩阵的副对角元素
6     """
7     n = len(d)
8
9     # 初始化sturm序列, P0(λ) = 1.0
10    seq = [1.0]

```

```

11
12     # 计算P1(λ) = d[0] - λ
13     val = d[0] - lam
14
15     # 用一个非常小的数替换零值, 方便符号变化的计算
16     if val == 0:
17         val = 1e-15
18     seq.append(val)
19
20     # 递推计算P_k(λ): P_k(λ) = (d[k-1] - λ) * P_{k-1}(λ) - (e[k-2])^2 * P_{k-2}(λ)
21     for k in range(2, n + 1):
22         val = (d[k-1] - lam) * seq[-1] - (e[k-2]**2) * seq[-2]
23         # 替换零值
24         if val == 0:
25             val = 1e-15
26         seq.append(val)
27
28     # 计算符号变化次数
29     changes = 0
30     for i in range(len(seq) - 1):
31         if (seq[i] > 0 and seq[i+1] < 0) or (seq[i] < 0 and seq[i+1] > 0):
32             changes += 1
33     return changes
34
35 def sturm_bisection(A, tol=1e-6):
36     # 提取对角线和副对角线元素
37     d = np.diag(A)
38     e = np.diag(A, k=1)
39     n = len(d)
40
41     # 利用Gershgorin圆盘定理计算特征值的初始搜索区间
42     max_val = -np.inf
43     min_val = np.inf
44     for i in range(n):
45         row_sum = np.sum(np.abs(A[i, :])) - abs(A[i, i])
46         if (row_sum + A[i, i]) > max_val:
47             max_val = row_sum + A[i, i]
48         if (A[i, i] - row_sum) < min_val:
49             min_val = A[i, i] - row_sum
50
51     # 避免边界问题, 稍微扩大区间
52     low = min_val - 1.0
53     high = max_val + 1.0
54
55     print(f"\n搜索区间: [{low}, {high}]")
56
57     found_evals = []
58     # 求解第1到第n个特征值 (对应k=1到n)
59     for k in range(1, n + 1):
60         a, b = low, high
61
62         for _ in range(100): # 最多迭代100次
63             mid = (a + b) / 2

```

```

64     count = sturm_sequence_count(d, e, mid)
65     if count < k:
66         # 符号变化次数小于k, 说明第k个特征值在mid右侧, 更新左边界
67         a = mid
68     else:
69         # 符号变化次数≥k, 说明第k个特征值在mid左侧或等于mid, 更新右边界
70         b = mid
71
72     if abs(b - a) < tol:
73         break
74
75     found_evals.append((a + b) / 2)
76
77 print("\nSturm序列+二分法特征值:", np.array(found_evals))
78 return np.array(found_evals)

```

结果

三种方法计算得到的本征值高度一致:

- **QR 算法:** [4.74528124, 3.17728292, 1.82271708, 0.25471876]
- **Jacobi 算法:** [0.25471876, 3.17728292, 1.82271708, 4.74528124]
- **Sturm 序列 + 二分法:** [0.25471872, 1.82271689, 3.17728311, 4.74528128]

QR算法迭代过程中的矩阵为:

```

1 第 5 次迭代:
2 [[ 4.29276628e+00 -7.21313977e-01  3.04034881e-16 -1.52321146e-16]
3 [-7.21313977e-01  3.55611356e+00 -3.34967464e-01 -2.34843862e-16]
4 [ 0.00000000e+00 -3.34967464e-01  1.89640130e+00 -3.99652715e-04]
5 [ 0.00000000e+00  0.00000000e+00 -3.99652715e-04  2.54718859e-01]]
6
7 第 10 次迭代:
8 [[ 4.73418406e+00 -1.31448547e-01  3.97342075e-16 -2.28200139e-17]
9 [-1.31448547e-01  3.18812610e+00 -1.85822775e-02 -2.92768574e-16]
10 [ 0.00000000e+00 -1.85822775e-02  1.82297108e+00 -2.07643132e-08]
11 [ 0.00000000e+00  0.00000000e+00 -2.07643133e-08  2.54718760e-01]]
12
13 第 15 次迭代:
14 [[ 4.74507887e+00 -1.78120256e-02  4.03511097e-16 -1.43164002e-18]
15 [-1.78120256e-02  3.17748431e+00 -1.15075434e-03 -2.94120430e-16]
16 [ 0.00000000e+00 -1.15075434e-03  1.82271806e+00 -1.10655208e-12]
17 [ 0.00000000e+00  0.00000000e+00 -1.10658664e-12  2.54718760e-01]]
18
19 第 20 次迭代:
20 [[ 4.74527757e+00 -2.39738535e-03  4.04242455e-16  1.46021165e-18]
21 [-2.39738535e-03  3.17728658e+00 -7.14944440e-05 -2.94147725e-16]
22 [ 0.00000000e+00 -7.14944440e-05  1.82271708e+00 -2.46592525e-17]
23 [ 0.00000000e+00  0.00000000e+00 -5.89784840e-17  2.54718760e-01]]
24
25 第 25 次迭代:
26 [[ 4.74528117e+00 -3.22632786e-04  4.04339745e-16  1.84942387e-18]]

```

```

27 [-3.22632786e-04 3.17728299e+00 -4.44210350e-06 -2.94147234e-16]
28 [ 0.00000000e+00 -4.44210350e-06 1.82271708e+00 3.43015275e-17]
29 [ 0.00000000e+00 0.00000000e+00 -3.14341662e-21 2.54718760e-01]]
30
31 第 30 次迭代:
32 [[ 4.74528124e+00 -4.34188356e-05 4.04352902e-16 1.90180274e-18]
33 [-4.34188356e-05 3.17728292e+00 -2.75997750e-07 -2.94147006e-16]
34 [ 0.00000000e+00 -2.75997750e-07 1.82271708e+00 3.43037661e-17]
35 [ 0.00000000e+00 0.00000000e+00 -1.67536827e-25 2.54718760e-01]]
36
37 第 35 次迭代:
38 [[ 4.74528124e+00 -5.84316066e-06 4.04354678e-16 1.90885171e-18]
39 [-5.84316066e-06 3.17728292e+00 -1.71483532e-08 -2.94146967e-16]
40 [ 0.00000000e+00 -1.71483531e-08 1.82271708e+00 3.43037101e-17]
41 [ 0.00000000e+00 0.00000000e+00 -8.92932493e-30 2.54718760e-01]]
42
43 第 40 次迭代:
44 [[ 4.74528124e+00 -7.86352881e-07 4.04354918e-16 1.90980034e-18]
45 [-7.86352881e-07 3.17728292e+00 -1.06546534e-09 -2.94146961e-16]
46 [ 0.00000000e+00 -1.06546526e-09 1.82271708e+00 3.43037066e-17]
47 [ 0.00000000e+00 0.00000000e+00 -4.75912342e-34 2.54718760e-01]]
48
49 42 次迭代后收敛。

```

第二题：幂次法求矩阵最大模本征值

解答

第一小问

注意到 $x'' = -Ax$ ，再由 $x'' = (-i\omega)^2 x = -\omega^2 x$ 即可得

$$Ax = \omega^2 x = \lambda x, \quad \lambda = \omega^2$$

第二小问

由题可得，矩阵 A 的形式为：

```

1 矩阵 A:
2 [[ 2. -1. 0. 0. 0. 0. 0. 0. -1.]
3 [-1. 2. -1. 0. 0. 0. 0. 0. 0.]
4 [ 0. -1. 2. -1. 0. 0. 0. 0. 0.]
5 [ 0. 0. -1. 2. -1. 0. 0. 0. 0.]
6 [ 0. 0. 0. -1. 2. -1. 0. 0. 0.]
7 [ 0. 0. 0. 0. -1. 2. -1. 0. 0.]
8 [ 0. 0. 0. 0. 0. -1. 2. -1. 0.]
9 [ 0. 0. 0. 0. 0. 0. -1. 2. -1.]
10 [ 0. 0. 0. 0. 0. 0. 0. -1. 2.]
11 [-1. 0. 0. 0. 0. 0. 0. -1. 2.]]

```

迭代公式： $z^{(k)} = Aq^{(k-1)}$, $q^{(k)} = z^{(k)} / \|z^{(k)}\|$; $q^T A q$ 即为本征值。

```

1 def power_method(A, max_iter=2000, tol=1e-10):
2     """
3         使用幂次法求解矩阵的最大特征值及其对应的特征向量。
4     """
5     n = A.shape[0]
6     # 从随机向量开始，使用固定种子以保证结果可复现
7     np.random.seed(42)
8     q = np.random.rand(n)
9     q = q / np.linalg.norm(q)
10
11    lambda_k = 0.0
12
13    print(f"{'迭代次数':<10} {'特征值':<20} {'相邻两次迭代的误差':<20}")
14    print("-" * 50)
15
16    for k in range(1, max_iter + 1):
17        z = A @ q
18
19        # 计算瑞利商作为特征值估计
20        lambda_next = np.dot(q, z)
21
22        # 归一化得到新的q
23        norm_z = np.linalg.norm(z)
24        q_next = z / norm_z
25
26        diff = np.abs(lambda_next - lambda_k)
27
28        if k % 10 == 0 or k == 1:
29            print(f"{k:<10} {lambda_next:<20.8f} {diff:<20.2e}")
30
31        if diff < tol:
32            print(f"\n在第 {k} 次迭代收敛")
33            return lambda_next, q_next
34
35        q = q_next
36        lambda_k = lambda_next
37
38    print("\n达到最大迭代次数。")
39    return lambda_k, q

```

结果

对于 $N = 10$ 的情形：

- **最大本征值** $\omega_{\max}^2: 4.00000000$
- **理论值:** $4 \sin^2(N\pi/2N) = 4$ 。
- **本征矢:** [-0.31623271, 0.31622465, -0.31621779, 0.31621473, -0.31621666, 0.31622282, -0.31623088, 0.31623774, -0.3162408, 0.31623887] 对应于相邻原子反向振动的模式。

第三题：孤立子数值解

解答

采用 Zabusky-Kruskal 格式，利用 Runge-Kutta 4 (RK4) 方法进行时间推进。

RK4 方法具有更高的精度和更好的数值稳定性，非常适合长时间模拟。将 KdV 方程写作：

$$\frac{du}{dt} = F(u)$$

其中空间离散算子 F 定义为：

$$F(u_j) = - \left[\frac{u_{j+1}^n + u_j^n + u_{j-1}^n}{3} \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} + \delta^2 \frac{u_{j+2}^n - 2u_{j+1}^n + 2u_{j-1}^n - u_{j-2}^n}{2(\Delta x)^3} \right]$$

RK4 方法通过下面的方式进行时间步进：

$$\begin{aligned} k_1 &= F(u^n) \\ k_2 &= F\left(u^n + \frac{\Delta t}{2}k_1\right) \\ k_3 &= F\left(u^n + \frac{\Delta t}{2}k_2\right) \\ k_4 &= F(u^n + \Delta t k_3) \\ u^{n+1} &= u^n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

核心代码：

```
1 # 使用差分法计算右端项
2 def compute_rhs(u):
3     # 使用 np.roll 处理周期性边界条件
4     # u_x 近似为 (u_{j+1} - u_{j-1}) / 2dx
5     # u_u_x 采用 Zabusky-Kruskal 守恒格式: 1/3 (u_{j+1} + u_j + u_{j-1}) * (u_{j+1} - u_{j-1}) / 2dx
6
7     u_jp1 = np.roll(u, -1)
8     u_jm1 = np.roll(u, 1)
9     u_jp2 = np.roll(u, -2)
10    u_jm2 = np.roll(u, 2)
11
12    # 非线性项: 1/3 * (u_{j+1} + u_j + u_{j-1}) * (u_{j+1} - u_{j-1}) / (2*dx)
13    nonlinear = (u_jp1 + u + u_jm1) / 3.0 * (u_jp1 - u_jm1) / (2 * dx)
14
15    # 色散项: delta^2 * (u_{j+2} - 2u_{j+1} + 2u_{j-1} - u_{j-2}) / (2*dx^3)
16    dispersion = delta_sq * (u_jp2 - 2*u_jp1 + 2*u_jm1 - u_jm2) / (2 * dx**3)
17
18    return -(nonlinear + dispersion)
19
20 # RK4 时间步进
21 def rk4_step(u, dt):
22     k1 = compute_rhs(u)
23     k2 = compute_rhs(u + 0.5 * dt * k1)
```

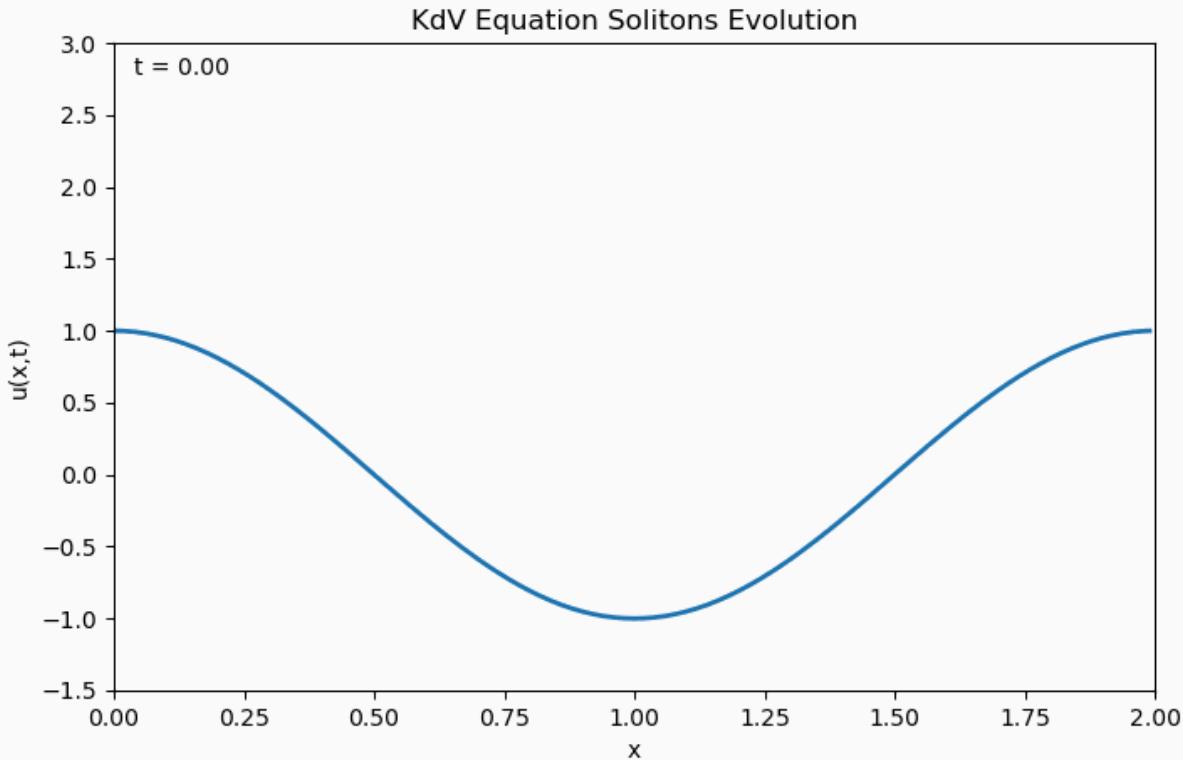
```

24     k3 = compute_rhs(u + 0.5 * dt * k2)
25     k4 = compute_rhs(u + dt * k3)
26     return u + (dt / 6.0) * (k1 + 2*k2 + 2*k3 + k4)

```

结果

初始的余弦波分裂成一系列孤立子，孤立子在传播过程中互不干扰，保持形状和速度不变。



第四题：二维波动方程

解答

(a) 解析解

设解形如

$$u(x, y, t) = X(x) Y(y) T(t).$$

代入 PDE:

$$X''(x)Y(y)T(t) + X(x)Y''(y)T(t) = X(x)Y(y)T''(t).$$

两边同除以 $X(x)Y(y)T(t)$ 得

$$\frac{X''(x)}{X(x)} + \frac{Y''(y)}{Y(y)} = \frac{T''(t)}{T(t)}.$$

左边仅含 x, y , 右边仅含 t , 因此两边等于常数。先对空间部分分离, 令

$$\frac{X''(x)}{X(x)} + \frac{Y''(y)}{Y(y)} = -\omega^2,$$

则时间部分满足

$$T''(t) + \omega^2 T(t) = 0.$$

进一步把空间项分成两常数：

$$\frac{X''(x)}{X(x)} = -\lambda, \quad \frac{Y''(y)}{Y(y)} = -\mu,$$

并且 $\lambda + \mu = \omega^2$ 。

因此得到三个常微分方程：

$$\begin{cases} X'' + \lambda X = 0, \\ Y'' + \mu Y = 0, \\ T'' + \omega^2 T = 0, \quad \omega^2 = \lambda + \mu. \end{cases}$$

边界为矩形且 $u = 0$ ，所以

$$X(0) = X(1) = 0, \quad Y(0) = Y(1) = 0.$$

于是得到：

$$X_n(x) = \sin(n\pi x), \quad \lambda_n = (n\pi)^2, \quad n = 1, 2, \dots$$

$$Y_m(y) = \sin(m\pi y), \quad \mu_m = (m\pi)^2, \quad m = 1, 2, \dots$$

于是每对 (n, m) 对应空间模式

$$\Phi_{nm}(x, y) = \sin(n\pi x) \sin(m\pi y),$$

对应的角频率

$$\omega_{nm} = \sqrt{\lambda_n + \mu_m} = \pi\sqrt{n^2 + m^2}.$$

对于每个模式 (n, m) ，时间函数为

$$T_{nm}(t) = A_{nm} \cos(\omega_{nm} t) + B_{nm} \sin(\omega_{nm} t).$$

因此一般解可以写成

$$u(x, y, t) = \sum_{n=1}^{\infty} \sum_{m=1}^{\infty} (A_{nm} \cos(\omega_{nm} t) + B_{nm} \sin(\omega_{nm} t)) \sin(n\pi x) \sin(m\pi y).$$

初值条件 $u(x, y, 0) = f(x, y) = \sin(\pi x) \sin(2\pi y)$ 给出

$$u(x, y, 0) = \sum_{n,m} A_{nm} \sin(n\pi x) \sin(m\pi y) = \sin(\pi x) \sin(2\pi y).$$

初始速度为零给出

$$\left. \frac{\partial u}{\partial t} \right|_{t=0} = \sum_{n,m} B_{nm} \omega_{nm} \sin(n\pi x) \sin(m\pi y) = 0$$

对所有 (x, y) 成立，所以对任意 n, m 有 $B_{nm} = 0$ 。

只需求 A_{nm} 。利用基函数的正交性：

$$\int_0^1 \int_0^1 \sin(n\pi x) \sin(m\pi y) \sin(p\pi x) \sin(q\pi y) dx dy = \frac{\delta_{np}\delta_{mq}}{4},$$

由此系数可写为

$$A_{pq} = 4 \int_0^1 \int_0^1 f(x, y) \sin(p\pi x) \sin(q\pi y) dx dy.$$

对 $f(x, y) = \sin(\pi x) \sin(2\pi y)$ ：

$$A_{pq} = 4 \int_0^1 \int_0^1 \sin(\pi x) \sin(2\pi y) \sin(p\pi x) \sin(q\pi y) dx dy.$$

仅当 $p = 1$ 且 $q = 2$ 时非零。计算该项：

$$A_{12} = 4 \left(\int_0^1 \sin^2(\pi x) dx \right) \left(\int_0^1 \sin^2(2\pi y) dy \right) = 4 \cdot \frac{1}{2} \cdot \frac{1}{2} = 1.$$

其余 $A_{nm} = 0$ 。

故最终解为

$$u(x, y, t) = \cos(\omega_{12}t) \sin(\pi x) \sin(2\pi y)$$

其中

$$\omega_{12} = \pi \sqrt{1^2 + 2^2} = \pi\sqrt{5}.$$

即

$$u(x, y, t) = \cos(\pi\sqrt{5}t) \sin(\pi x) \sin(2\pi y).$$

(b) 数值解

对于边界条件，每一步时间更新后显式将边界格点置零：

```

1 | u_np1[0, :] = 0.0
2 | u_np1[-1, :] = 0.0
3 | u_np1[:, 0] = 0.0
4 | u_np1[:, -1] = 0.0

```

对于初始条件，有：

- 对于 $u(x, y, 0) = \sin(\pi x) \sin(2\pi y)$ 直接在网格上赋值：

```

1 | u_prev[:, :] = np.sin(np.pi * X) * np.sin(2 * np.pi * Y)

```

- 对于 $\partial_t u|_{t=0} = 0$ ，将函数在初始点做 Taylor 展开：

$$u(x, y, \Delta t) = u(x, y, 0) + \Delta t u_t(x, y, 0) + \frac{\Delta t^2}{2} u_{tt}(x, y, 0) + O(\Delta t^3).$$

而波动方程给出 $u_{tt} = \nabla^2 u$, 故当初速度 $u_t(x, y, 0) = 0$ 时, 上式变为

$$u^1 = u^0 + \frac{\Delta t^2}{2} \nabla^2 u^0 + O(\Delta t^3).$$

即下面的代码中的第11行:

```

1 # 初始条件: u(x,y,0)
2 u_prev[:, :] = np.sin(np.pi * x) * np.sin(2 * np.pi * y)
3 u_nm1 = u_prev.copy() # u^{n-1}
4 u_n = u_nm1.copy() # u^n
5
6 # 计算 u^n, 使用 u_t(0) = 0
7 u_nm1_xx = u_nm1[2:, 1:-1] - 2*u_nm1[1:-1, 1:-1] + u_nm1[:-2, 1:-1]
8 u_nm1_yy = u_nm1[1:-1, 2:] - 2*u_nm1[1:-1, 1:-1] + u_nm1[1:-1, :-2]
9 rx2 = (dt / dx)**2
10 ry2 = (dt / dy)**2
11 u_n[1:-1, 1:-1] = u_nm1[1:-1, 1:-1] + 0.5 * (rx2 * u_nm1_xx + ry2 * u_nm1_yy)

```

结果如下:

(b) 数值求解并与解析解对比 ($\lambda=1.0$)...

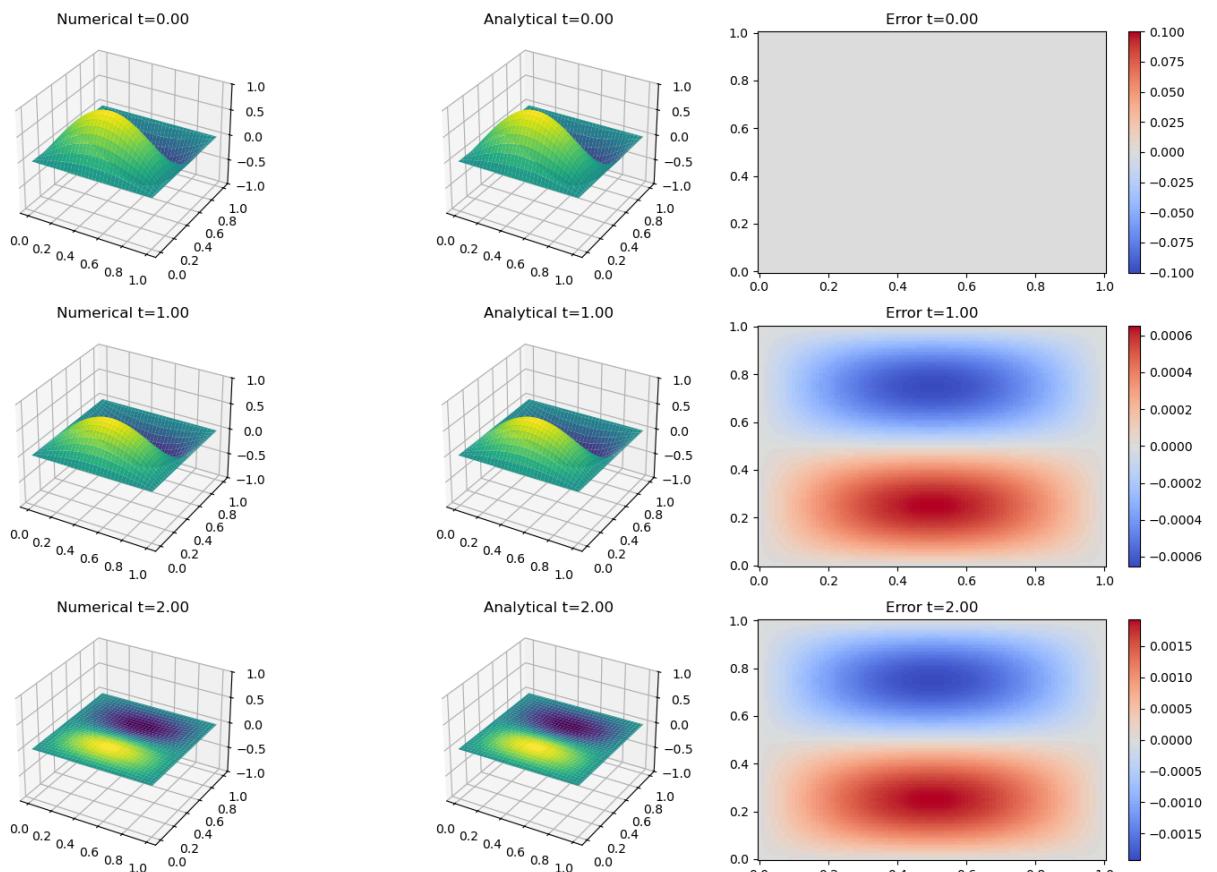
-> 误差分析:

$t=0.00$: Max Error=0.0000e+00, MSE=0.0000e+00

$t=1.00$: Max Error=6.5333e-04, MSE=1.0461e-07

$t=2.00$: Max Error=1.9278e-03, MSE=9.1084e-07

对比图已保存到 hw4/assets/wave_comparison.png



可以看到，由于按照初始值设定初始状态，初始时刻误差为零；随时间推移图中误差条最大值变大，即误差变大；整体来看，数值解的精度还不错，最大误差控制在 10^{-3} 量级，均方误差控制在 10^{-7} 量级。

(c) 数值表现

时间步长

稳定性条件： $\Delta t \leq \frac{1}{\sqrt{\lambda}} \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} \right)^{-1/2}$ ，且 $\lambda \geq 1$ 。

取 $\Delta t = \frac{1}{\sqrt{\lambda}} \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} \right)^{-1/2}$ 、 $Nx=Ny=100$ 并取不同的 λ ，得到结果如下：

```
1 **测试不同时间步长dt下的数值稳定性**
2
3 取lambda=3.0计算得到的时间步长 dt = 4.08248e-03
4 -> 误差分析 (lambda=3.0):
5     t=0.00: Max Error=0.0000e+00, MSE=0.0000e+00
6     t=1.00: Max Error=5.0157e-04, MSE=6.1654e-08
7     t=2.00: Max Error=1.4779e-03, MSE=5.3528e-07
8 计算稳定，结束时刻 t=5.00
9
10 取lambda=2.0计算得到的时间步长 dt = 5.00000e-03
11 -> 误差分析 (lambda=2.0):
12     t=0.00: Max Error=0.0000e+00, MSE=0.0000e+00
13     t=1.00: Max Error=4.1944e-04, MSE=4.3115e-08
14     t=2.00: Max Error=1.2375e-03, MSE=3.7529e-07
15 计算稳定，结束时刻 t=5.00
16
17 取lambda=1.01计算得到的时间步长 dt = 7.03598e-03
18 -> 误差分析 (lambda=1.01):
19     t=0.00: Max Error=0.0000e+00, MSE=0.0000e+00
20     t=1.00: Max Error=1.7906e-04, MSE=7.8580e-09
21     t=2.00: Max Error=5.3123e-04, MSE=6.9161e-08
22 计算稳定，结束时刻 t=5.00
23
24 取lambda=0.99计算得到的时间步长 dt = 7.10669e-03
25 -> 误差分析 (lambda=0.99):
26     t=0.00: Max Error=0.0000e+00, MSE=0.0000e+00
27     t=1.00: Max Error=1.7904e-04, MSE=7.4045e-09
28     t=1.84: Max Error=8.3026e+04, MSE=1.1241e+09
29 计算发散，失稳时刻 t=1.85
30
31 取lambda=0.8计算得到的时间步长 dt = 7.90569e-03
32 -> 误差分析 (lambda=0.8):
33     t=0.00: Max Error=0.0000e+00, MSE=0.0000e+00
34     t=0.43: Max Error=8.7746e+04, MSE=7.3076e+08
35     t=0.43: Max Error=8.7746e+04, MSE=7.3076e+08
36 计算发散，失稳时刻 t=0.43
```

可以看到，当 $\lambda \geq 1$ 时，数值解是稳定的；而当 $\lambda < 1$ 时，数值解发散。

空间步长

固定 dt ，取不同的 Nx 、 Ny ，得到结果如下：

```

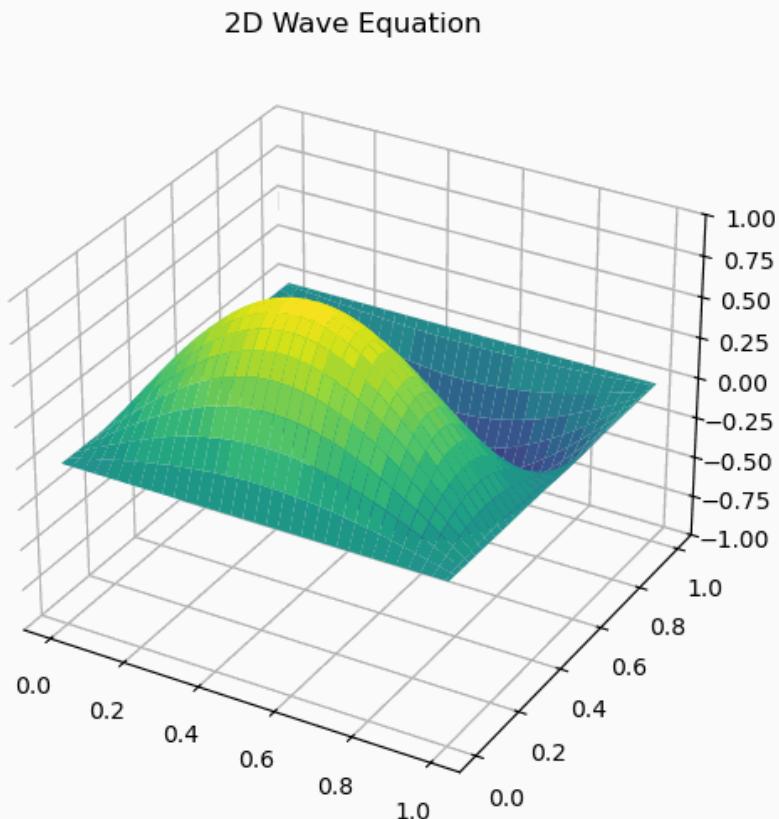
1 ***测试不同空间网格划分下的数值稳定性**:
2
3 -> 误差分析 (Nx=50, Ny=50, dt=0.001):
4     t=0.00: Max Error=0.0000e+00, MSE=0.0000e+00
5     t=1.00: Max Error=2.6327e-03, MSE=1.6721e-06
6     t=2.00: Max Error=7.7789e-03, MSE=1.4598e-05
7 计算稳定, 结束时刻 t=5.00
8
9 -> 误差分析 (Nx=100, Ny=100, dt=0.001):
10    t=0.00: Max Error=0.0000e+00, MSE=0.0000e+00
11    t=1.00: Max Error=6.5333e-04, MSE=1.0461e-07
12    t=2.00: Max Error=1.9278e-03, MSE=9.1084e-07
13 计算稳定, 结束时刻 t=5.00
14
15 -> 误差分析 (Nx=150, Ny=150, dt=0.001):
16    t=0.00: Max Error=0.0000e+00, MSE=0.0000e+00
17    t=1.00: Max Error=2.8498e-04, MSE=2.0045e-08
18    t=2.00: Max Error=8.4072e-04, MSE=1.7445e-07
19 计算稳定, 结束时刻 t=5.00

```

可以看到, 当 `dt` 一定时, 随着网格划分越来越精细, 即 `Nx`、`Ny` 越来越大, 数值解的误差越来越小。另外, 还可以注意到, 在同一个模拟中, 随着时间 `t` 的推移, 数值解误差越来越大, 即数值解误差是随着时间逐渐累积的。

(d) 动画演示

取网格数 `Nx=Ny=100`, 模拟时间 `T_max=5.0`, 步长 `dt=0.001` 得到动画:



第五题：随机数产生器

解答

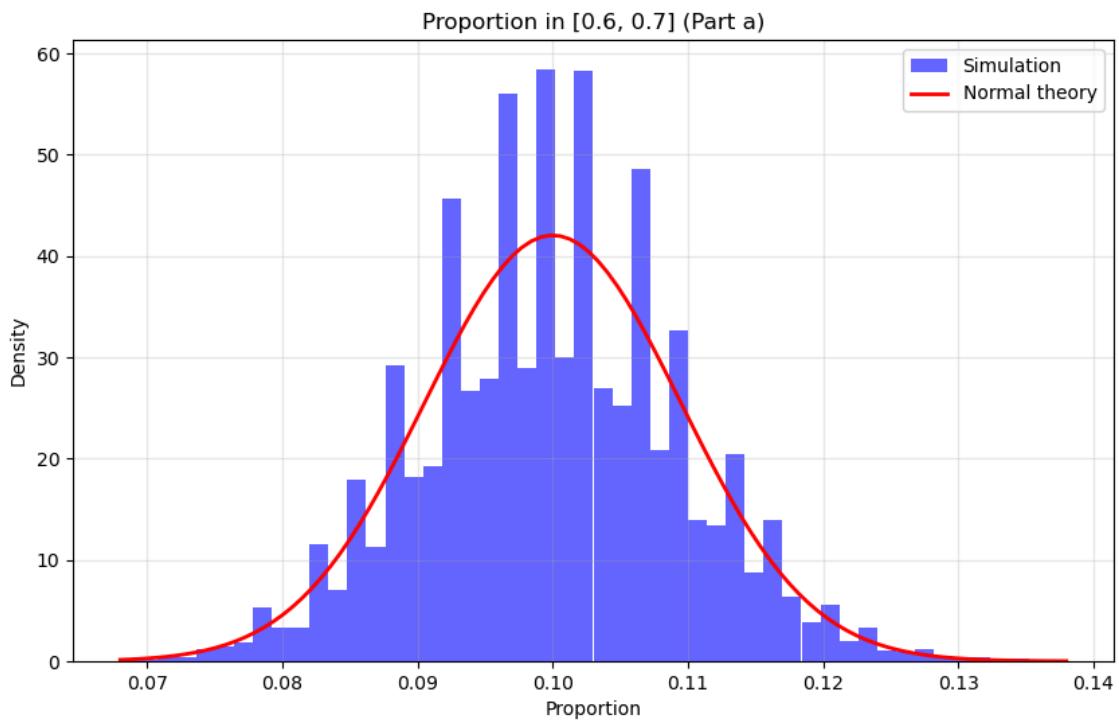
通过定义模拟函数 `run_simulation()` 来完成前两问：

```
1 def run_simulation(N_samples=1000, N_experiments=10000, n_bins=10):
2     # 理论上每个区间的期望频数
3     expected_count = N_samples / n_bins
4
5     print(f"正在运行 {N_experiments} 次实验，每次 {N_samples} 个样本...")
6
7     # 生成所有随机数
8     all_randoms = np.random.rand(N_experiments, N_samples)
9
10    # 区间划分 [0,0.1), [0.1,0.2), ..., [0.9,1.0], bin 索引 0 到 9
11    bin_indices = np.floor(all_randoms * n_bins).astype(int)
12
13    # 处理 rand 正好为 1.0 的边缘情况
14    bin_indices[bin_indices == n_bins] = n_bins - 1
15
16    # 统计每个实验在各区间的频数
17    bin_counts = np.zeros((N_experiments, n_bins))
18    for b in range(n_bins):
19        # 统计落在bin索引为 b 的样本在每行中出现的次数
20        bin_counts[:, b] = np.sum(bin_indices == b, axis=1)
21
22    # (a) 部分数据，区间 [0.6, 0.7] 对应 bin 索引 6
23    counts_06_07 = bin_counts[:, 6]
24    proportions_06_07 = counts_06_07 / N_samples
25
26    # (b) 部分数据，每个实验的卡方统计量：sum (O - E)^2 / E
27    chi_squared = np.sum((bin_counts - expected_count)**2 / expected_count, axis=1)
28
29    return proportions_06_07, chi_squared
```

其中，`all_randoms` 存储了所有实验的随机数，每一行对应一次实验；`bin_indices` 计算每个随机数落在哪个区间；`bin_counts` 统计每次实验中每个区间的频数，行为实验次数，列为区间索引。返回两个结果：落在 [0.6, 0.7] 区间的比例 `proportions_06_07` 和每次实验的 χ^2 统计量 `chi_squared`。

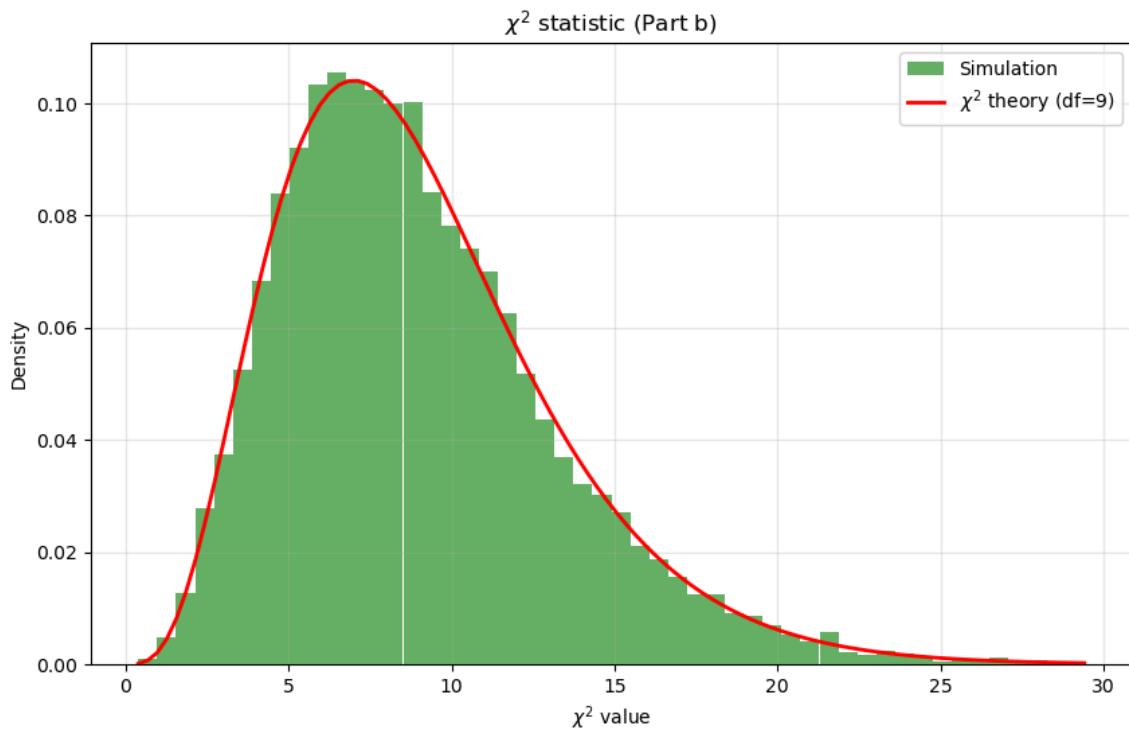
(a) 分布检验

统计落入 [0.6, 0.7] 区间的比例，结果符合正态分布。



(b) χ^2 检验

计算 $\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$, 结果符合自由度为 9 的 χ^2 分布。



(c) 16807 产生器

实现线性同余发生器：

```
1 class LCG16807:  
2     def __init__(self, seed=1):  
3         self.state = seed  
4         self.a = 16807  
5         self.m = 2147483647  
6  
7     def next(self):  
8         self.state = (self.a * self.state) % self.m  
9         return self.state
```

验证结果：

```
--- (c) 16807 随机数生成器验证 ---  
x(10000) = 1043618065  
验证成功，结果与题目一致。
```