实验四

王为　　2311605

一、矩阵类

1、类的定义与继承

定义 Matrix 类和它的子类 SquareMatrix（方阵）类，定义实例属性 data 存储矩阵，定义普通方法来完成矩阵操作。定义实例方法完成矩阵运算，定义静态构造方法辅助完成矩阵的构造。

```python
# mymatrix.py
import random

class Matrix:
    def __init__(self, data):
        self.data = data
        self.check_matrix()
```

```python
class SquareMatrix(Matrix):
    def __init__(self, data):
        super().__init__(data)  # 调用父类构造方法
        if not self.is_square():
            raise ValueError("矩阵必须为方阵")
```

2、检测函数

（1）简介

设置检测函数以确保输入的矩阵满足相应计算的预期的格式，避免在后续计算或操作中发生错误。

（2）check_matrix 函数

该函数用于检测矩阵是否合法，若矩阵非法则无法进行后续任何矩阵计算，抛出异常并由主函数 main()处理异常；若函数合法则返回 True，由主调函数进行后续运算。

代码：

```python
def check_matrix(self):
        """检查矩阵是否有效"""
        if self.data is None:
            raise ValueError("输入矩阵不能为空（None）")
        if not isinstance(self.data, list) or not all(isinstance(row,
list) for row in self.data):
```

```
            raise ValueError("输入必须是一个二维列表")
        if len(self.data) == 0:
            raise ValueError("输入矩阵不能为空")
        if len(self.data[0]) == 0:
            raise ValueError("输入矩阵的第一行不能为空")

        num_cols = len(self.data[0])
        for row in self.data:
            if len(row) != num_cols:
                raise ValueError("输入矩阵的每一行必须具有相同的列数")

        for i, row in enumerate(self.data):
            for j, elem in enumerate(row):
                if not isinstance(elem, (int, float)):
                    raise ValueError(f"矩阵元素 ({i}, {j}) 不是数字:
{elem}")
```

（3）is_square 函数

该函数用于检测矩阵是否为方阵，有些运算要求输入必须为方阵或者最好为方阵（如求行列式、求逆等）。该函数返回一个 bool 值，表示矩阵是否为方阵，主调函数可以根据该值进行下一步操作（若为 False 由主调函数决定是否抛出异常）。

代码：

```
def is_square(self):
        """判断矩阵是否为方阵"""
        return len(self.data) > 0 and all(len(row) == len(self.data) for
row in self.data)
```

（4）are_SameShape 函数

该函数用于检测矩阵是否形状相同，有些运算要求输入的两个矩阵必须形状相同或者最好相同（如矩阵间的加减及哈达玛积等）。函数返回值同（3）。

代码：

```
def same_shape(self, other):
        """判断矩阵是否形状相同"""
        return len(self.data) == len(other.data) and len(self.data[0])
== len(other.data[0])
```

2、静态构造方法

这里提前封装一些类对象的静态构造方法，它们可以根据需求返回不同初始化的矩阵类对象，而不需要人工输入，使代码具有更高的组织性、可读性和灵活性，避免潜在的错误和误用。

代码：

```
    @staticmethod
    def generate_zero_matrix(n, m):
        """创建一个 n*m 的零初始化矩阵"""
        return Matrix([[0 for _ in range(m)] for _ in range(n)])

    @staticmethod
    def generate_random_matrix(n, m, lower_bound=0, upper_bound=100):
        """创建一个 n*m 的随机初始化矩阵"""
        return Matrix([[random.randint(lower_bound, upper_bound) for _
in range(m)] for _ in range(n)])

    @staticmethod
    def generate_identity_matrix(n):
        """创建一个 n*n 的单位矩阵"""
        return Matrix([[1 if i == j else 0 for j in range(n)] for i in
range(n)])
```

此处展示的是父类中的静态构造方法，这些方法会在子类中重写，以构造方阵。

3、矩阵加减及哈达玛积运算函数

（1）简介

矩阵的哈达玛积即逐元素积。这三种运算的运算逻辑相似，即将 A、B 两矩阵逐元素相加、减、乘，故使用相同算法实现。

（2）算法实现

- **形状检查**：首先调用 `are_SameShape` 检查两个矩阵 A 和 B 的形状是否相同（矩阵 A、B 的合法性检测内置于 `are_SameShape` 函数中）。
- **逐元素相加/减/乘**：使用嵌套的列表推导式来进行逐元素相加/减/乘（以

此算法为例介绍嵌套列表推导式的使用，以后从略）：

- 外层循环遍历矩阵 A、B 的行。
- 内层循环遍历每一行的元素，即遍历列。
- 每个元素 $A[i][j]$和 $B[i][j]$相加/减/乘，并将结果存储在新的列表中。

- **返回结果**：返回一个新的矩阵，包含每个位置上对应元素的和/差/积。

代码（仅展示哈达玛积，其他类似）：

```
def hadamard_product(self, other):
        """计算矩阵哈达玛积"""
        if not self.same_shape(other):
            raise ValueError("哈达玛积要求两个矩阵形状相同")
        return Matrix([[self.data[i][j] * other.data[i][j] for j in
range(len(self.data[0]))] for i in range(len(self.data))])
```

4、矩阵的转置运算函数

（1）矩阵的合法性检测在构造时已被检测，无需再次检测。（以后从略）

（2）使用嵌套列表推导式进行转置，通过组合两层循环，构造出新矩阵的第j行第i列元素为原矩阵的第i行第j列，实现矩阵的转置。

代码:

```
def transpose(self):
        """计算矩阵的转置"""
        return Matrix([[self.data[i][j] for i in range(len(self.data))]
for j in range(len(self.data[0]))])
```

5、矩阵的乘积运算函数

（1）传统算法（暴力法）

- 数学公式：

$$C[i][j] = \sum_{k=1}^{n} A[i][k] \times B[k][j]$$

- 算法逻辑：

使用三层循环分别遍历 A 的行及 B 的行列，按照矩阵乘法公式进行计算。在内层循环中，将 A 的第 i 行和 B 的第 j 列对应元素相乘并累加到 result 矩阵的 (i,j) 元素上。（详见代码及注释）

• 复杂度反思：

该算法使用 3 层循环实现乘法，时间复杂度为 $O(n^3)$，在矩阵维度增大时，时间开销大幅增加，考虑对其进行优化。

代码：

```
def multiply(self, other):
        """计算矩阵乘积"""
        if len(self.data[0]) != len(other.data):
            raise ValueError("矩阵乘法要求第一个矩阵的列数等于第二个矩阵的行数")

        result = [[0 for _ in range(len(other.data[0]))] for _ in range(len(self.data))]
        for i in range(len(self.data)):
            for j in range(len(other.data[0])):
                for k in range(len(other.data)):
                    result[i][j] += self.data[i][k] * other.data[k][j]
        return Matrix(result)
```

(2) 传统算法的基于硬件优化

• 优化原理：

现代计算机使用缓存（Cache）来提高数据访问速度。由于缓存的局部性原理，当程序访问某个内存地址时，周围的数据也会被加载到缓存中。若算法能够使相邻的数据在同一时间被访问，那么可以有效利用缓存，提高性能。

默认的访问顺序 i,j,k 在访问矩阵 A 和 B 时导致不良的内存访问模式，尤其是 $B[k][j]$ 的访问会在内存中是非连续的，可能导致多次缓存缺失。

• 算法优化：

改变访问顺序，使用 i,k,j 可以提高对矩阵 B 的访问效率，因为它是对列的连续访问，能更好地利用缓存。

代码：

```
for i in range(len(self.data)):
    for k in range(len(other.data)):
         for j in range(len(other.data[0])):
```

```
                result[i][j] += self.data[i][k] * other.data[k][j]
# 其余部分同（2）
```

6、矩阵的行列式运算函数（子类方法）

(1) 递归算法

- 处理特殊情况（设置递归边界）。如果矩阵是1×1，直接返回该元素的值作为行列式;如果矩阵是2×2，使用标准的行列式公式 $det(A) = a_{11}a_{22} - a_{12}a_{21}$ 直接计算并返回结果。
- 对于大于2×2的方阵，初始化行列式值为 0，并遍历第一行的每个元素。
- 对于每个元素，生成对应的余子矩阵（即去掉当前元素所在的行和列）。
- 递归调用 determinant 函数计算余子矩阵的行列式，并根据当前元素的索引计算行列式的总值，使用公式：

$$det(A) += (-1)^c \times a_{1c} \times det(M_{1c})$$

其中c是当前列的索引，a_{1c}是第一行第 c 列的元素，M_{1c}是对应的余子矩阵。

代码：

```
def determinant(self):
        """计算矩阵的行列式"""
        n = len(self.data)

        if n == 1:
            return self.data[0][0]
        if n == 2:
            return self.data[0][0] * self.data[1][1] - self.data[0][1] *
self.data[1][0]

        det = 0
        for c in range(n):
            minor = [[self.data[i][j] for j in range(n) if j != c] for i
in range(1, n)]
            det += ((-1) ** c) * self.data[0][c] * SquareMatrix
(minor).determinant()

        return det
```

(2) 高斯消元法

使用高斯消元法将矩阵化为上三角矩阵（若主元为 0 则向下寻找不为 0 的主元），该矩阵的行列式等于各主元的乘积。

```
def gaussian_determinant(self):
        """计算矩阵的行列式"""
        n = len(self.data)

        if not self.is_square():
            raise ValueError("矩阵必须为方阵")

        A = [row[:] for row in self.data]
        det = 1

        for i in range(n):
            # 查找主元
            if A[i][i] == 0:
                for j in range(i + 1, n):
                    if A[j][i] != 0:
                        A[i], A[j] = A[j], A[i]  # 交换行
                        det *= -1  # 行交换会改变行列式的符号
                        break

            pivot = A[i][i]
            if pivot == 0:
                return 0

            for j in range(i + 1, n):
                ratio = A[j][i] / pivot
                for k in range(i, n):
                    A[j][k] -= ratio * A[i][k]

            det *= pivot

        return det
```

（3）算法对比

• 递归法：

该方法通过递归计算行列式，每次递归调用都需要生成一个 $(n-1)\times(n-1)$的余子矩阵。对于一个 $n\times n$矩阵，递归会调用 n 次，每次调用需要 $O(n)$的时间来生成余子矩阵。因此，时间复杂度为：

$$T(n) = n \cdot T(n-1) + O(n) \implies T(n) = O(n!)$$

递归法在处理大矩阵时非常低效。

• 高斯消元法：

高斯消元法的核心在于将矩阵转换为上三角矩阵。处理每一行时，内层循环将需要操作 n 次，且对于每一行，外层和中层循环结合起来可以看作是对 n 行和 n 列的操作。整体复杂度为：$O(n^3)$

适合处理较大规模的矩阵，效率显著更高。

7、计算矩阵的伴随矩阵函数（子类方法）

(1) 数学公式：

$$C[j][i] = (-1)^{i+j} * \det(A_{ij})$$

其中 A_{ij} 是矩阵 A 去掉第 i 行和第 j 列后的子矩阵。

(2) 算法逻辑：

• 获取矩阵的维度 n 并创建一个 $n \times n$ 的零矩阵 用于存储伴随矩阵的结果。

• 通过从原矩阵中删除第 i 行和第 j 列来生成 A_{ij}（minor）

• 计算伴随矩阵中位置 (j, i) 的元素，其值为 minor 的行列式乘以 $(-1)^{i+j}$

代码：

```
    def get_adjugate(self):
        """计算伴随矩阵"""
        n = len(self.data)
        adjugate = [[0] * n for _ in range(n)]  # 创建零矩阵

        for i in range(n):
            for j in range(n):
                # 代数余子式
                minor = [r[:j] + r[j + 1:] for r in (self.data[:i] +
self.data[i + 1:])]
                # 伴随矩阵的元素
                adjugate[j][i] = ((-1) ** (i + j)) *
SquareMatrix(minor).gaussian_determinant()  # 计算余子式

        return adjugate
```

8、计算矩阵的逆矩阵函数（子类方法）

（1） 伴随矩阵法

• 数学公式：

$$A^{-1} = \frac{1}{det(A)} \cdot adj(A)$$

• 算法逻辑：

调用 determinant 函数计算$det(A)$，调用 get_adjugate 函数计算$adj(A)$，然后按照公式将$adj(A)$矩阵逐元素除以$det(A)$即可（详见代码）。

代码：

```
def adjugate_inverse(self):
        """使用伴随矩阵法计算逆矩阵"""
        det = self.gaussian_determinant()

        if det == 0:
            raise ValueError("该矩阵不可逆。")

        adjugate = self.get_adjugate()
        inverse = [[adjugate[i][j] / det for j in range(len(adjugate))]
for i in range(len(adjugate))]
        return inverse
```

（2）高斯消元法

• 创建一个增广矩阵，将输入矩阵与单位矩阵拼接在一起。增广矩阵的左半部分是原矩阵，右半部分是单位矩阵。

• 偏序寻找主元，选择当前列中绝对值最大的元素作为主元。增加计算的数值稳定性，减少由于舍入误差导致的计算不准确。（若最大主元为 0，则不可逆）

• 使用高斯消元法将增广矩阵转化为行最简形式。通过行操作，消去当前主元列下方的所有元素，形成上三角矩阵。

• 从最后一行开始，逐行向上消去上方的元素，形成单位矩阵的左侧，最终将增广矩阵转化为单位矩阵与逆矩阵的组合。

• 从增广矩阵中提取右侧部分，得到逆矩阵。

代码：

```
def gaussian_inverse(self):
        """计算矩阵的逆"""
        n = len(self.data)  # 获取矩阵的行数（方阵的行数等于列数）

        # 创建增广矩阵，将原矩阵与单位矩阵拼接在一起
```

```
        aug_matrix = [row[:] + [1 if i == j else 0 for j in range(n)]
for i, row in enumerate(self.data)]

        # 执行高斯消元法以将增广矩阵转换为行最简形式
        for i in range(n):
            # 寻找主元
            max_row = i  # 初始化最大行索引
            for j in range(i + 1, n):
                # 找到绝对值最大的主元
                if abs(aug_matrix[j][i]) > abs(aug_matrix[max_row][i]):
                    max_row = j

            # 如果找到的主元行不等于当前行，进行交换
            if max_row != i:
                aug_matrix[i], aug_matrix[max_row] =
aug_matrix[max_row], aug_matrix[i]  # 交换行

            # 检查主元是否为零
            if aug_matrix[i][i] == 0:
                raise ValueError("矩阵不可逆，主元为零")

            # 归一化当前主元行，使主元变为 1
            pivot = aug_matrix[i][i]
            for k in range(2 * n):
                aug_matrix[i][k] /= pivot  # 将当前行的每个元素除以主元的值

            # 消元，调整当前行以下的所有行
            for j in range(i + 1, n):
                factor = aug_matrix[j][i]  # 计算消元因子
                for k in range(2 * n):
                    aug_matrix[j][k] -= factor * aug_matrix[i][k]  # 消
元，调整当前行

        # 进行反向消元，消去上三角中的元素
        for i in range(n - 1, -1, -1):  # 从最后一行开始向上处理
            for j in range(i - 1, -1, -1):  # 处理当前行以上的所有行
                factor = aug_matrix[j][i]  # 计算消元因子
                for k in range(2 * n):
                    aug_matrix[j][k] -= factor * aug_matrix[i][k]  # 消
元，调整当前行

        # 提取逆矩阵，逆矩阵位于增广矩阵的右侧部分
        inverse_matrix = []
        for i in range(n):
```

```
            row = aug_matrix[i][n:]  # 取右半部分
            inverse_matrix.append(row)  # 将计算得到的行添加到逆矩阵中

        return inverse_matrix  # 返回计算得到的逆矩阵
```

二、测试程序

```
main.py > ...
1   # main.py
2   import mymatrix as mm
3   import random
4   import numpy as np
5   import time
```

引入 mymatrix.py 脚本文件，在 main.py 中进行测试。

1、安全性测试

（1）生成非法矩阵，测试尝试使用非法矩阵构造类对象时 check_matrix 函数能否正确识别并抛出异常。

测试代码：

```
def generate_illegal_matrices():
    """生成非法矩阵"""
    illegal_matrices = [
        [[1, 2], [3, 4], [5]],      # 不规则矩阵（行长度不同）
        [[1, 'a'], [3, 4]],         # 包含非数字元素
        None,                       # None 类型
        [[1, 2], [3, 4], []],       # 一行为空
        [[1, 2, 3], [4, 5]]         # 非方阵用于行列式和逆矩阵测试
    ]
    return illegal_matrices
def test_matrix_operations():
    illegal_matrices = generate_illegal_matrices()

    for matrix in illegal_matrices:
        print(f"Testing with matrix: {matrix}")
        try:
            mm.check_matrix(matrix)
        except Exception as e:
            print("Matrix Exception:", e)
        print("-" * 40)
```

测试结果：

```
Testing with matrix: [[1, 2], [3, 4], [5]]
Matrix Exception: 输入矩阵的每一行必须具有相同的列数
----------------------------------------
Testing with matrix: [[1, 'a'], [3, 4]]
Matrix Exception: 矩阵元素 (0, 1) 不是数字: a
----------------------------------------
Testing with matrix: None
Matrix Exception: 输入矩阵不能为空（None）
----------------------------------------
Testing with matrix: [[1, 2], [3, 4], []]
Matrix Exception: 输入矩阵的每一行必须具有相同的列数
----------------------------------------
Testing with matrix: [[1, 2, 3], [4, 5]]
Matrix Exception: 输入矩阵的每一行必须具有相同的列数
----------------------------------------
```

结果分析：函数按照预期识别并抛出异常。

（2）测试使用非方阵构造方阵类对象和使用不同形状矩阵相乘时相应的检测函数能否正确的被调用和运行。

部分代码：

```python
def test_exceptions():
    # 测试非方阵
    matrix = mm.Matrix.generate_random_matrix(2, 3)  # 2 行 3 列的矩阵
    print("Testing non-square matrix:")
    try:
        mm.SquareMatrix(matrix.data)
    except ValueError as e:
        print(f"成功捕获异常: {e}")

    # 测试形状不同的矩阵
    matrix_a = mm.Matrix.generate_random_matrix(3, 3)  # 3x3 矩阵
    matrix_b = mm.Matrix.generate_random_matrix(3, 2)  # 3x2 矩阵
    print("Testing matrices of different shapes:")
    try:
        if not matrix_b.hadamard_product(matrix_a):
            raise ValueError("矩阵形状不同")
    except ValueError as e:
        print(f"成功捕获异常: {e}")
```

测试结果：

```
Testing non-square matrix:
成功捕获异常: 矩阵必须为方阵
Testing matrices of different shapes:
成功捕获异常: 哈达玛积要求两个矩阵形状相同
```

结果分析：检测函数正确的被调用和运行

2、准确性检测

（1）测试内容

NumPy 是广泛使用的数值计算库，其实现经过严格验证。通过与 NumPy 库的结果进行比较，测试矩阵类各算法（如矩阵乘法、加法、减法、Hadamard 乘积、转置、行列式、逆矩阵和伴随矩阵）的准确性。

将**随机生成**的 1000 组**随机大小**矩阵分别作为本矩阵类（及其子类）各函数的输入和 Numpy 库中相应功能的函数的输入，接收它们的返回值，然后使用 Numpy 库提供的 `allclose` 函数进行比较并计算出各函数返回值的准确率。（为缩短测试消耗的时间，随机生成的矩阵维度不超过 100，矩阵元素大小不超过 1000）。

（2）部分测试代码（其余部分逻辑相同）

```
def test_matrix_functions(num_tests=100):
    """测试所有矩阵函数的准确率"""

    # 测试 matrix_multiply
    correct_count = 0
    for _ in range(num_tests):
        size1=random.randint(1, 100)
        size2=random.randint(1, 100)
        size3=random.randint(1, 100)
        A = mm.Matrix.generate_random_matrix(size1, size2)
        B = mm.Matrix.generate_random_matrix(size2, size3)
        try:
            custom_result = A.multiply(B)
            numpy_result = np.matmul(A, B)
            if np.allclose(custom_result, numpy_result.tolist()):
                correct_count += 1
        except ValueError:
            continue  # 如果矩阵不符合乘法条件，跳过
```

（3）测试结果

```
Matrix Multiply Accuracy: 100.00%
Add Matrices Accuracy: 100.00%
Subtract Matrices Accuracy: 100.00%
Hadamard Product Accuracy: 100.00%
Transpose Accuracy: 100.00%
Determinant Accuracy: 100.00%
Gaussian Determinant Accuracy: 100.00%
Gaussian Matrix Inverse Accuracy: 100.00%
Adjugate Accuracy: 80.70%
Adjugate Matrix Inverse Accuracy: 82.90%
```

（4）结果分析

• 矩阵乘法、加法、减法、Hadamard 乘积、转置、行列式（递归法和高斯消元法）和高斯消元法计算的矩阵逆的准确率均为 100%。这表明这些操作的实现可靠且与预期结果一致。

• 伴随矩阵和伴随矩阵法计算矩阵逆的准确率只有 80%左右，可能是计算伴随矩阵时浮点数运算的精度问题，计算过程中会出现数值误差。在实际操作中不建议使用这两个函数，建议使用其他算法实现的相同功能函数。

3、性能测试

（1）测试内容

为评估矩阵类各算法在处理矩阵时的效率，设置函数测试它们的时间开销。函数随机生成矩阵，根据算法时间复杂度选取不同规模的矩阵测试性能。

（2） 测试代码

```
def time_function(func, *args):
    """计算函数执行时间"""
    start_time = time.time()
    func(*args)
    end_time = time.time()
    return (end_time - start_time)  # 返回每次调用的平均时间

# 测试每个函数的时间开销
def test_performance():
    """测试各个矩阵操作函数的性能"""

    matrix = mm.SquareMatrix.generate_random_matrix(8)
    print(f"Testing performance with a {8}x{8} matrix:")
    print(f"Determinant: {time_function(matrix.determinant):.6f}
seconds")
    print("-" * 40)
```

```
    matrix = mm.SquareMatrix.generate_random_matrix(10)
    print(f"Testing performance with a {10}x{10} matrix:")
    print(f"Adjugate: {time_function(matrix.get_adjugate):.6f} seconds")
    print(f"Adjugate Matrix Inverse:
{time_function(matrix.adjugate_inverse):.6f} seconds")
    print("-" * 40)

    matrix = mm.SquareMatrix.generate_random_matrix(100)
    print(f"Testing performance with a {100}x{100} matrix:")
    print(f"Gaussian determinant:
{time_function(matrix.gaussian_determinant):.6f} seconds")
    print(f"Gaussian Matrix Inverse:
{time_function(matrix.gaussian_inverse):.6f} seconds")
    print(f"Matrix Multiply: {time_function(matrix.multiply,matrix):.6f}
seconds")
    print("-" * 40)

    matrix = mm.SquareMatrix.generate_random_matrix(1000)
    print(f"Testing performance with a {1000}x{1000} matrix:")
    print(f"Hadamard Product:
{time_function(matrix.hadamard_product,matrix):.6f} seconds")
    print(f"Add Matrices: {time_function(matrix.add, matrix):.6f}
seconds")
    print(f"Subtract Matrices:
{time_function(matrix.subtract,matrix):.6f} seconds")
    print("-" * 40)
```

（3） 测试结果

```
Testing performance with a 8x8 matrix:
Determinant: 0.085946 seconds
----------------------------------------
Testing performance with a 10x10 matrix:
Adjugate: 0.002991 seconds
Adjugate Matrix Inverse: 0.003001 seconds
----------------------------------------
Testing performance with a 100x100 matrix:
Gaussian determinant: 0.014962 seconds
Gaussian Matrix Inverse: 0.092275 seconds
Matrix Multiply: 0.081111 seconds
----------------------------------------
Testing performance with a 1000x1000 matrix:
Hadamard Product: 0.138574 seconds
Add Matrices: 0.109080 seconds
Subtract Matrices: 0.108616 seconds
----------------------------------------
```

```
Testing performance with a 10x10 matrix:
Determinant: 6.036884 seconds
----------------------------------------
Testing performance with a 100x100 matrix:
Adjugate: 300.620825 seconds
Adjugate Matrix Inverse: 274.757297 seconds
----------------------------------------
Testing performance with a 1000x1000 matrix:
Gaussian determinant: 40.238156 seconds
Gaussian Matrix Inverse: 163.376429 seconds
Matrix Multiply: 79.452389 seconds
----------------------------------------
Testing performance with a 10000x10000 matrix:
Hadamard Product: 18.042707 seconds
Add Matrices: 15.400078 seconds
Subtract Matrices: 15.111032 seconds
----------------------------------------
```

（4）结果分析

• 递归算法求矩阵的行列式时间开销随矩阵维度快速增长，运算 8*8 矩阵耗时不足 0.1s,运算 10*10 矩阵耗时超过 6s，不适用于中大规模矩阵运算。

• 计算伴随矩阵的时间开销也较大，但运算 10*10 和 100*100 矩阵耗时可以接受，适用于中小规模矩阵，同时不建议使用该算法计算矩阵的逆。

• 使用高斯消元法计算矩阵的行列式和矩阵的逆表现出较好的性能，可用于较大规模的矩阵运算。

• 矩阵的基本运算（加、减、哈达玛积）时间开销较小，即使对于超大规模矩阵也能在 10s 左右完成。

三、遇到的困难

1. **异常处理：**
   • 问题：对矩阵的各种条件检查和异常处理要求细致。
   • 解决方案：实现全面的异常处理机制，确保对不同类型的错误进行适当处理。
2. **数学理论：**
   • 问题：逆矩阵和伴随矩阵的计算依赖于数学知识。
   • 解决方案：深入学习和理解相关数学公式。
3. **性能优化：**
   • 问题：Python 的性能不如 C++，在处理大数据集时，效率较低。
   • 解决方案：优化算法，寻找并使用时间复杂度更低的算法。

4. **高阶函数**

- 问题：测试算法时重复代码过多。
- 解决方案：使用接受函数和可变数量参数的高阶函数。

5. **实例属性和类属性**

- 问题：可能会误用实例属性和类属性，导致数据不按预期存储和访问。
- 解决方案：使用清晰的命名来区分实例属性（self 前缀）和类属性（直接通过类名访问）。

6. **实例方法与静态方法**

- 问题：不能将自动生成矩阵算法封装成该类的实例方法，因为无法在类外不依赖对象直接访问实例方法。
- 解决方案：将自动生成矩阵算法封装成静态构造方法。