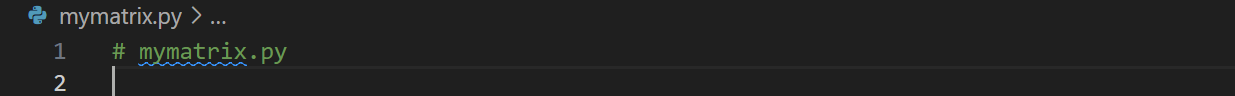
实验三

王为 2311605

1. 矩阵运算函数库



矩阵运算函数库，保存在mymatrix.py中。

**1、检测函数**

（1）简介

该函数库提供矩阵运算服务，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，可以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，该函数库无法继续提供服务，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：

def check\_matrix(matrix):

    """检查矩阵是否合法"""

    # 检查是否为 None

    if matrix is None:

        raise ValueError("输入矩阵不能为空（None）")

    # 检查是否为二维列表

    if not isinstance(matrix, list) or not all(isinstance(row, list) for row in matrix):

        raise ValueError("输入必须是一个二维列表")

    # 检查矩阵是否为空

    if len(matrix) == 0:

        raise ValueError("输入矩阵不能为空")

    # 检查第一行是否为空

    if len(matrix[0]) == 0:

        raise ValueError("输入矩阵的第一行不能为空")

    # 检查每一行的列数是否相同

    num\_cols = len(matrix[0])

    for row in matrix:

        if len(row) != num\_cols:

            raise ValueError("输入矩阵的每一行必须具有相同的列数")

    # 检查每个元素是否为数字

    for i, row in enumerate(matrix):

        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}")

    return True

（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(matrix):

    """判断矩阵是否为方阵"""

    check\_matrix(matrix)

    return len(matrix) > 0 and all(len(row) == len(matrix) for row in matrix)

（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 are\_SameShape(A,B):

    """判断矩阵是否形状相同"""

    check\_matrix(A)

    check\_matrix(B)

    return len(A) == len(B) and len(A[0]) == len(B[0])

**2、矩阵构造函数**

这里提前实现一些生成矩阵的函数，测试程序和函数库中的后续操作都可以调用它们生成所需的矩阵，而不需要人工输入。

代码：

def  generate\_zero\_matrix(n, m):

    """创建一个n\*m的零初始化矩阵"""

    return [[0 for \_ in range(m)] for \_ in range(n)]

def  generate\_random\_matrix(n, m, lower\_bound=0, upper\_bound=10):

    """创建一个n\*m的随机初始化矩阵"""

return [[random.randint(lower\_bound, upper\_bound) for \_ in range(m)] for \_ in range(n)]

def generate\_identity\_matrix(n):

    """创建一个n\*n的单位矩阵"""

    return [[1 if i == j else 0 for j in range(n)] for i in range(n)]

**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(A,B)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。
  + 每个元素和 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(A, B):

    """计算哈达玛积"""

    if not are\_SameShape(A,B):

        raise ValueError("哈达玛积要求两个矩阵形状相同")

    return [[A[i][j] \* B[i][j] for j in range(len(A[0]))] for i in range(len(A))]

**4、矩阵的转置运算函数**

（1）矩阵的合法性检测（所有函数均会先进行相应检测，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，构造出新矩阵的第行第列元素为原矩阵的第行第列，实现矩阵的转置。

代码：

def transpose(matrix):

    """计算矩阵的转置"""

    check\_matrix(matrix)

    return [[matrix[i][j]  for i in range(len(matrix))] for j in range(len(matrix[0]))]

**5、矩阵的乘积运算函数**

（1）传统算法（暴力法）

·数学公式：

·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 行和 B 的第 列对应元素相乘并累加到 result 矩阵的  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,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 matrix\_multiply(A, B):

    """计算矩阵乘积"""

    check\_matrix(A)

    check\_matrix(B)

    if len(A[0]) != len(B):

        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")

    # 初始化结果矩阵

    result = generate\_zero\_matrix(len(A),len(B[0]))

for i in range(len(A)):

# 遍历矩阵 A 的每一行

        for j in range(len(B[0])):

# 遍历矩阵 B 的每一列。

            for k in range(len(B)):

# 遍历矩阵 B 的每一行。

                result[i][j] += A[i][k] \* B[k][j]

    return 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 在访问矩阵A和B时导致不良的内存访问模式，尤其是 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(A)):

for k in range(len(B)):

        for j in range(len(B[0])):

                result[i][j] += A[i][k] \* B[k][j]

# 其余部分同（2）

1. **矩阵的行列式运算函数**

(1)递归算法

* + 处理特殊情况（设置递归边界）。如果矩阵是，直接返回该元素的值作为行列式;如果矩阵是，使用标准的行列式公式直接计算并返回结果。
  + 对于大于的方阵，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：

其中是当前列的索引，是第一行第  列的元素，是对应的余子矩阵。

代码：

def determinant(matrix):

"""计算矩阵的行列式"""

    # 检查输入矩阵是否为方阵，如果不是，则抛出异常

    if not is\_square(matrix):

        raise ValueError("矩阵必须为方阵")

    n = len(matrix)  # 获取矩阵的行数（列数）

    # 处理特殊情况：如果矩阵是 1x1，直接返回该元素

    if n == 1:

        return matrix[0][0]

    # 处理特殊情况：如果矩阵是 2x2，使用直接计算的公式

    if n == 2:

        return matrix[0][0] \* matrix[1][1] - matrix[0][1] \* matrix[1][0]

    det = 0  # 初始化行列式的值

    # 使用递归计算行列式，遍历第一行的每个元素

    for c in range(n):

        # 生成当前元素的余子矩阵（去掉当前行和当前列）

        minor = [[matrix[i][j] for j in range(n) if j != c] for i in range(1, n)]

        # 递归调用，计算余子矩阵的行列式

        # 根据当前列的索引 c 计算行列式的值

        det += ((-1) \*\* c) \* matrix[0][c] \* determinant(minor)

    return det  # 返回最终计算的行列式值

(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。

代码：

def gaussian\_determinant(matrix):

    n = len(matrix)

    if not is\_square(matrix):

        raise ValueError("矩阵必须为方阵")

    A = [row[:] for row in matrix]

    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 的余子矩阵。对于一个矩阵，递归会调用 次，每次调用需要的时间来生成余子矩阵。因此，时间复杂度为：

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：

适合处理较大规模的矩阵，效率显著更高。

1. **计算矩阵的伴随矩阵函数**

(1)数学公式：

其中 是矩阵去掉第 行和第列后的子矩阵。  
(2)算法逻辑：

·获取矩阵的维度 n并创建一个的零矩阵 用于存储伴随矩阵的结果。

·通过从原矩阵中删除第 行和第 列来生成（minor）

·计算伴随矩阵中位置的元素，其值为minor的行列式乘以

代码：

def get\_adjugate(matrix):

"""计算伴随矩阵"""

if not is\_square(matrix):

        raise ValueError("矩阵必须为方阵")

    n = len(matrix)

    adjugate = generate\_zero\_matrix(n, n)

    for i in range(n):

        for j in range(n):

            # 获得

            minor = [r[:j] + r[j+1:] for r in (matrix[:i] + matrix[i+1:])]

            # 获得伴随矩阵的元素

            adjugate[j][i]=((-1) \*\* (i + j)) \* determinant(minor)

    return adjugate

1. **计算矩阵的逆矩阵函数**
2. 伴随矩阵法

·数学公式：

·算法逻辑：

调用determinant函数计算，调用get\_adjugate函数计算，然后按照公式将矩阵逐元素除以即可（详见代码）。

代码：

def adjugate\_matrix\_inverse(matrix):

    """使用伴随矩阵法计算逆矩阵"""

    det = determinant(matrix)

    if det == 0:

        raise ValueError("该矩阵不可逆。")

    adjugate = get\_adjugate(matrix)

    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\_matrix\_inverse(matrix):

    """计算矩阵的逆"""

    # 检查输入矩阵是否为方阵，如果不是，则抛出异常

    if not is\_square(matrix):

        raise ValueError("矩阵必须为方阵")

    n = len(matrix)  # 获取矩阵的行数（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(matrix)]

    # 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 = []

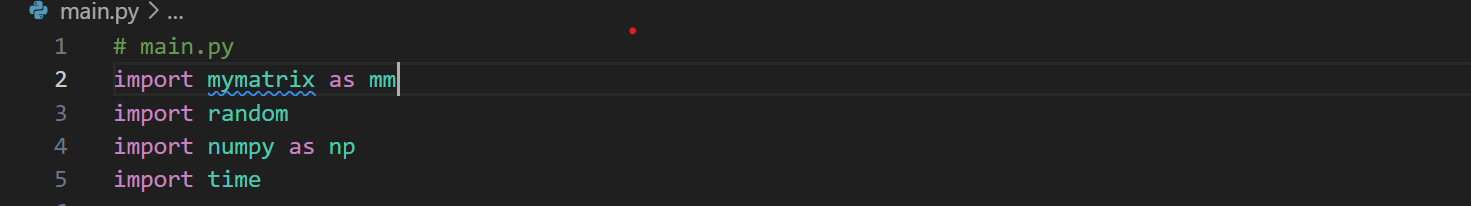
    for i in range(n):

        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



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:

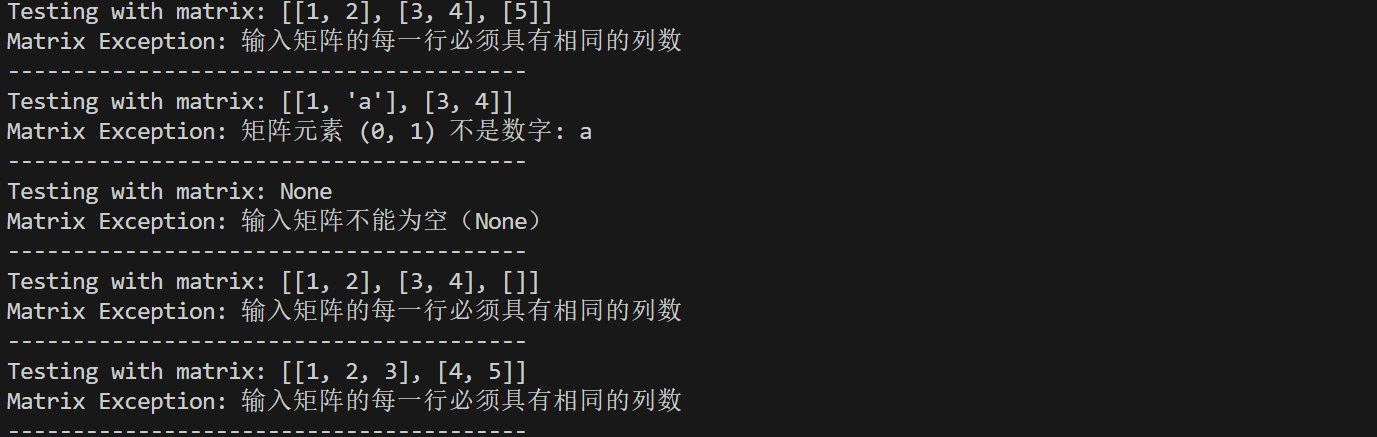
            mm.check\_matrix(matrix)

        except Exception as e:

            print("Matrix Multiply Exception:", e)

        print("-" \* 40)

测试结果：



结果分析：函数按照预期识别并抛出异常。

（2）使用合法矩阵测试是否每个算法都调用了check\_matrix函数。

部分代码：

def test\_all\_algorithms():

    """测试所有矩阵算法是否调用了 check\_matrix 函数"""

    valid\_matrix\_A = [[1, 2], [3, 4]]

    valid\_matrix\_B = [[5, 6], [7, 8]]

    # 测试矩阵乘法

    print("矩阵乘法:")

    result = mm.matrix\_multiply(valid\_matrix\_A, valid\_matrix\_B)

    # 测试哈达玛积

    print("哈达玛积:")

result = mm.hadamard\_product(valid\_matrix\_A, valid\_matrix\_B)

# 对剩余算法重复以上操作

测试结果：

结果分析：所有算法调用函数对矩阵进行检测。

（3）测试方阵检测函数和形状相同检测函数能否正确的被调用和运行。

部分代码：

def test\_exceptions():

    # 测试非方阵

    matrix = mm.generate\_random\_matrix(2, 3)  # 2行3列的矩阵

    print("Testing non-square matrix:")

    try:

        mm.matrix\_inverse(matrix)

    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.generate\_random\_matrix(3, 3)  # 3x3 矩阵

    matrix\_b = mm.generate\_random\_matrix(3, 2)  # 3x2 矩阵

    print("Testing matrices of different shapes:")

    try:

        if not mm.hadamard\_product(matrix\_a, matrix\_b):

            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：



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，测试函数库矩阵操作（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**矩阵分别作为本函数库各函数的输入和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=1000):

    """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.generate\_random\_matrix(size1, size2)

        B = mm.generate\_random\_matrix(size2, size3)

        try:

            custom\_result = mm.matrix\_multiply(A, 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

    print(f"Matrix Multiply Accuracy: {correct\_count / num\_tests \* 100:.2f}%")

    # 测试 matrix\_inverse

    correct\_count = 0

    for \_ in range(num\_tests):

        size1=random.randint(1, 100)

        A = mm.generate\_random\_matrix(size1, size1)

        try:

            # 确保矩阵可逆

                custom\_result = mm.matrix\_inverse(A)

                numpy\_result = np.linalg.inv(A).tolist()

                if np.allclose(custom\_result, numpy\_result):

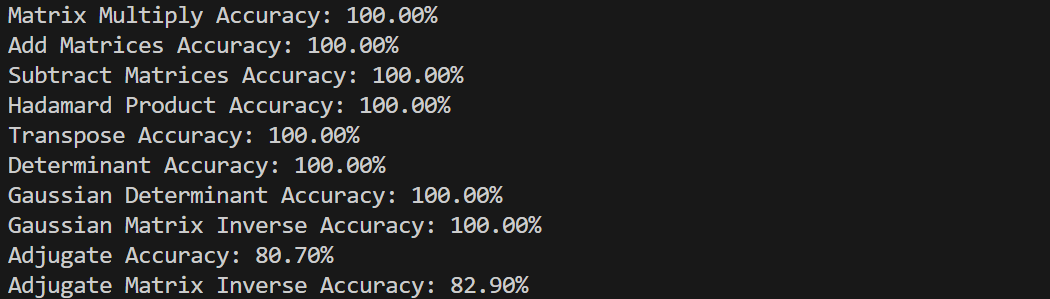
                    correct\_count += 1

        except ValueError:

            continue

    print(f"Matrix Inverse Accuracy: {correct\_count / num\_tests \* 100:.2f}%")

（3）测试结果



（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

为评估函数库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.generate\_random\_matrix(10,10)

    print(f"Testing performance with a {10}x{10} matrix:")

    print(f"Determinant: {time\_function(mm.determinant, matrix):.6f} seconds")

    print("-" \* 40)

    matrix = mm.generate\_random\_matrix(100,100)

    print(f"Testing performance with a {100}x{100} matrix:")

    print(f"Adjugate: {time\_function(mm.get\_adjugate, matrix):.6f} seconds")

    print(f"adjugate\_matrix\_inverse: {time\_function(mm.adjugate\_matrix\_inverse, matrix):.6f} seconds")

    print("-" \* 40)

    matrix = mm.generate\_random\_matrix(1000,1000)

    print(f"Testing performance with a {1000}x{1000} matrix:")

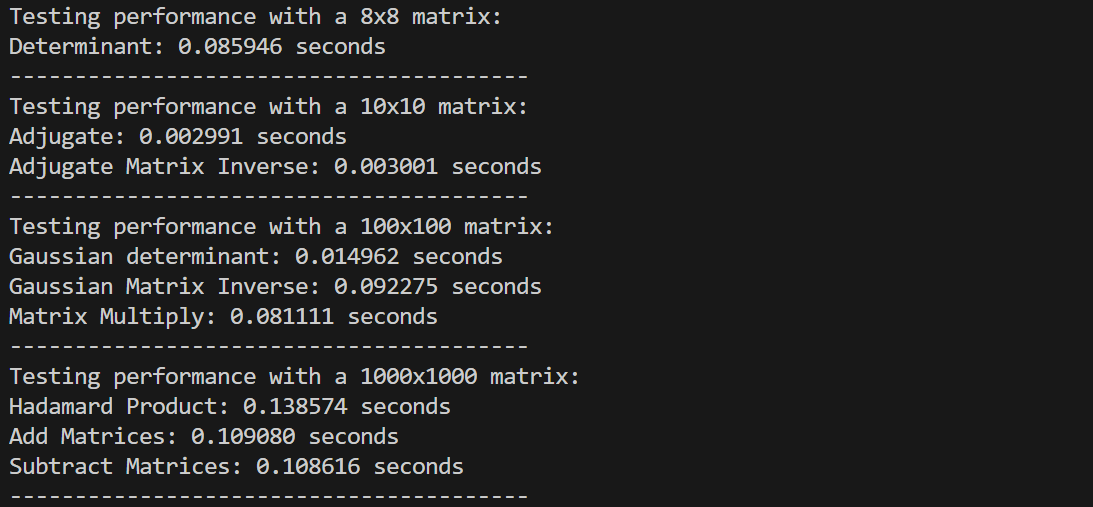
    print(f"Gaussian determinant: {time\_function(mm.gaussian\_determinant, matrix):.6f} seconds")

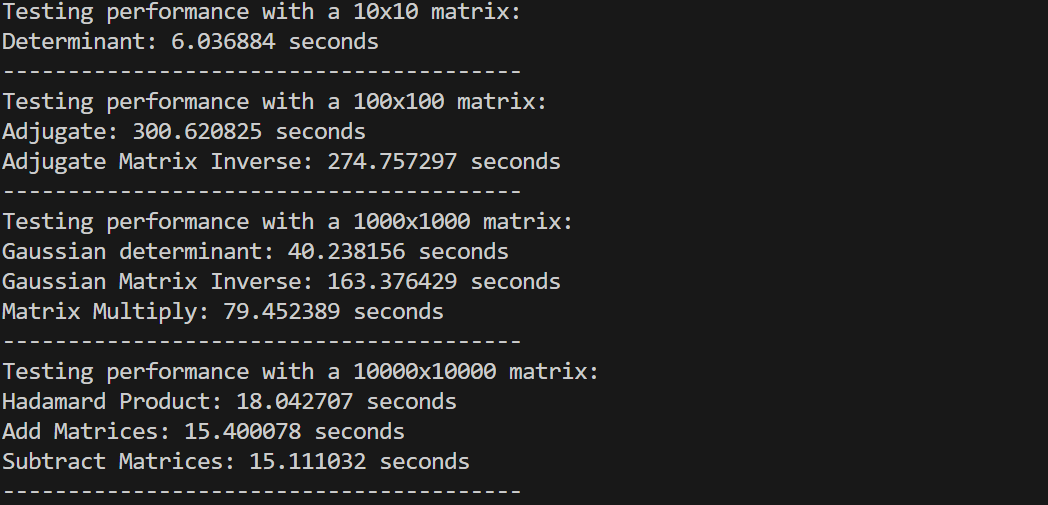
    print(f"gaussian\_matrix\_inverse: {time\_function(mm.gaussian\_matrix\_inverse, matrix):.6f} seconds")

    print(f"matrix\_multiply: {time\_function(mm.matrix\_multiply, matrix,matrix):.6f} seconds")

    print("-" \* 40)

1. 测试结果





（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. **矩阵存储**：

·问题：矩阵是二维数据，无法直接存储。

·解决方案：使用 list[list[int]] 的形式存储矩阵。

1. **嵌套列表推导式**：

·问题：传统方法先创建矩阵，然后使用二层循环进行赋值，操作较为复杂。

·解决方案：使用嵌套列表推导式快速生成所需的矩阵。

1. **高斯消元法**：

·问题：在进行高斯消元时，因舍入误差导致计算结果不准确。

·解决方案：采用偏序寻找主元法，选择当前列中绝对值最大的元素作为主元。

1. **递归计算行列式**：

·问题：计算行列式时，递归的逻辑较为复杂。

·解决方案：编写清晰的辅助函数，简化递归调用过程。

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。

1. **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。

1. **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。

1. **高阶函数**

·问题：测试函数库的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。