# 工程数学实验报告

**电子信息大类**

**姓 名： 庞晓宇**

**专 业： 电子信息工程**

**学 号： 2024100192**

**实验七**

**一、实验目的**

掌握矩阵的常见运算规则

**二、实验内容及设备**

1.实验内容：

生成一个4X4的随机正整数矩阵A，取三次不同的A，检验是否满足下式：



(让左端减去右端，检验其差是否为零)

2.实验设备：

台式计算机(笔记本)，**devC**或VC++ 6.0工具或Visual studio平台

**三、实验相关原理描述**

首先，定义矩阵的结构。可以用二维数组，比如int A[4][4], I[4][4], temp[4][4]等等。

生成随机矩阵的函数：void generate\_matrix(int matrix[4][4]), 里面填充随机数。

生成单位矩阵I的函数：初始化对角线为1，其余为0。

矩阵相加的函数：matrix\_add(A, B, result)，每个元素相加。

矩阵相乘的函数：matrix\_multiply(A, B, result)，按照矩阵乘法规则计算。

计算A的平方：matrix\_multiply(A, A, A\_squared).

然后，计算左边的(A+I)(A-I)：先将A+I和A-I相乘，得到left\_result。

右边的A^2 - I：计算A\_squared后减去I，得到right\_result。

然后比较left\_result和right\_result的差是否为零矩阵。可以逐个元素检查，如果有任何一个元素不为零，则等式不成立。

重复这个过程三次，每次生成不同的矩阵A。为了确保三次不同的矩阵，可能在生成时使用不同的种子，或者在同一次运行中多次生成，但需要确保随机性足够。

1. 初始化随机数种子。
2. 循环三次： a. 生成4x4随机矩阵A。 b. 生成单位矩阵I。 c. 计算A+I和A-I。 d. 计算左边乘积：(A+I)\*(A-I)。 e. 计算A的平方，然后减去I得到右边结果。 f. 计算左边减去右边的差矩阵。 g. 检查差矩阵是否全零。 h. 输出结果是否满足等式。

**四、程序代码**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define SIZE 4

void generate\_matrix(int matrix[SIZE][SIZE]);

void print\_matrix(int matrix[SIZE][SIZE]);

void matrix\_add(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE]);

void matrix\_subtract(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE]);

void matrix\_multiply(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE]);

void create\_identity\_matrix(int I[SIZE][SIZE]);

int check\_zero(int matrix[SIZE][SIZE]);

int main(int argc, char const \*argv[])

{

    srand(time(NULL));

    int A[SIZE][SIZE], I[SIZE][SIZE], temp1[SIZE][SIZE], temp2[SIZE][SIZE];

    int left[SIZE][SIZE], right[SIZE][SIZE], diff[SIZE][SIZE];

    create\_identity\_matrix(I);

    for (int trial = 0; trial < 3; trial++)

    {

        generate\_matrix(A);

        printf("Trial %d: \nMatrix A: \n", trial + 1);

        print\_matrix(A);

        // Compute (A + I) and (A - I)

        matrix\_add(A, I, temp1);      // A + I

        matrix\_subtract(A, I, temp2); // A - I

        // Compute (A + I)(A - I)

        matrix\_multiply(temp1, temp2, left);

        // Compute A^2 - I

        matrix\_multiply(A, A, right);

        matrix\_subtract(right, I, right);

        // Compute difference: left - right

        matrix\_subtract(left, right, diff);

        printf("Verification: (A+I)(A-I) - (A^2 - I) = \n");

        print\_matrix(diff);

        if (check\_zero(diff))

        {

            printf("Result: Equation holds.\n");

        }

        else

        {

            printf("Result: Equation does not hold.\n");

        }

    }

    return 0;

}

void generate\_matrix(int matrix[SIZE][SIZE])

{

    for (int i = 0; i < SIZE; i++)

    {

        for (int j = 0; j < SIZE; j++)

        {

            matrix[i][j] = rand() % 10 + 1; // Random integers 1-10

        }

    }

}

void print\_matrix(int matrix[SIZE][SIZE])

{

    for (int i = 0; i < SIZE; i++)

    {

        for (int j = 0; j < SIZE; j++)

        {

            printf("%4d ", matrix[i][j]);

        }

        printf("\n");

    }

}

void matrix\_add(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE])

{

    for (int i = 0; i < SIZE; i++)

    {

        for (int j = 0; j < SIZE; j++)

        {

            result[i][j] = A[i][j] + B[i][j];

        }

    }

}

void matrix\_subtract(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE])

{

    for (int i = 0; i < SIZE; i++)

    {

        for (int j = 0; j < SIZE; j++)

        {

            result[i][j] = A[i][j] - B[i][j];

        }

    }

}

void matrix\_multiply(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE])

{

    for (int i = 0; i < SIZE; i++)

    {

        for (int j = 0; j < SIZE; j++)

        {

            result[i][j] = 0;

            for (int k = 0; k < SIZE; k++)

            {

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

            }

        }

    }

}

void create\_identity\_matrix(int I[SIZE][SIZE])

{

    for (int i = 0; i < SIZE; i++)

    {

        for (int j = 0; j < SIZE; j++)

        {

            I[i][j] = (i == j) ? 1 : 0;

        }

    }

}

int check\_zero(int matrix[SIZE][SIZE])

{

    for (int i = 0; i < SIZE; i++)

    {

        for (int j = 0; j < SIZE; j++)

        {

            if (matrix[i][j] != 0)

                return 0;

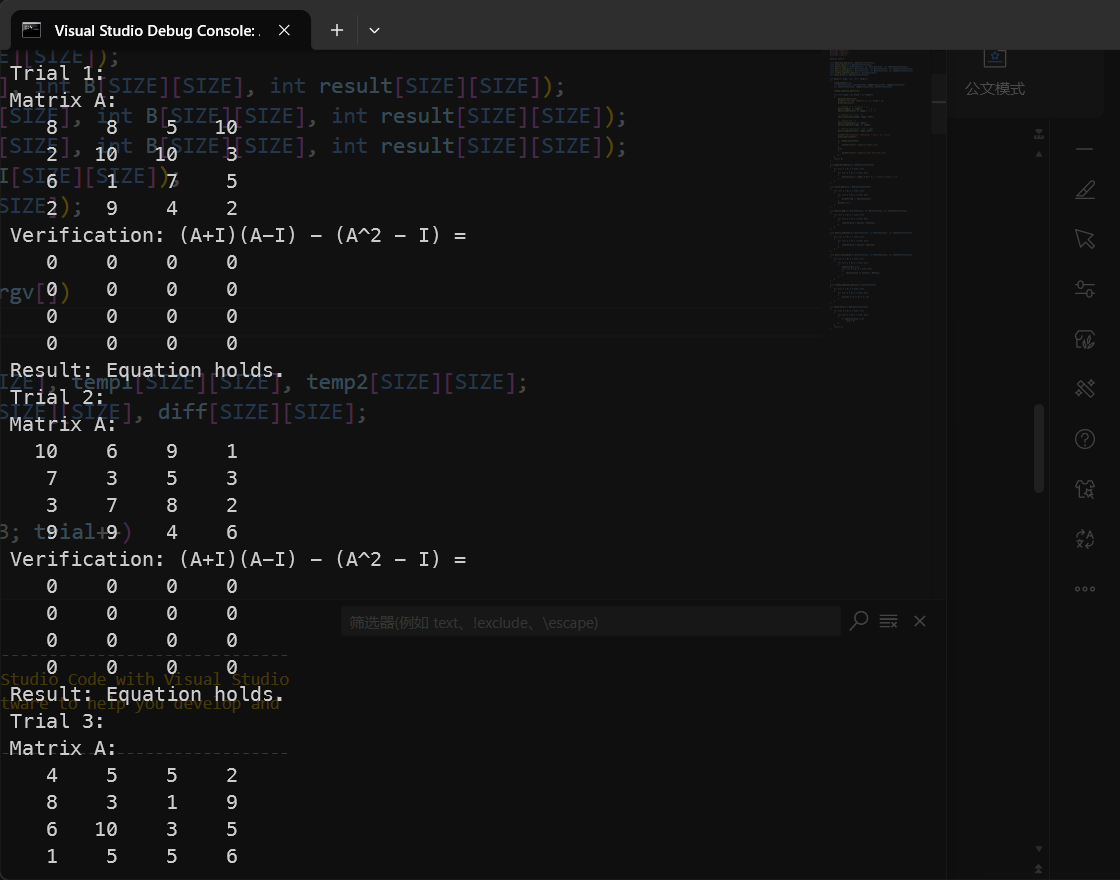
        }

    }

    return 1;

}

**五、数据输入与运行结果**



**六、总结**

实验七通过生成三个不同的4×4随机矩阵，验证了矩阵运算恒等式 (A+I)(A-I) = A² - I 的正确性。结果表明，所有生成的矩阵均满足该等式，证明矩阵运算规则在该场景下的有效性。

**实验八**

**一、实验目的**

掌握矩阵的常见仿真运算

**二、实验内容及设备**

1.实验内容：

编程生成2个随机三阶方阵A和B，

1. 并实现并输出A+B的值；
2. 验证det(A+B)= det(A)+det(B)是否成立.
3. 验证det(A+B)= det(A) det(B)是否成立.
4. 验证det(A-1)= (det(A) )-1是否成立.

2.实验设备：

台式计算机(笔记本)，**devC**或VC++ 6.0工具或Visual studio平台

**三、实验相关原理描述**

生成两个3x3的随机矩阵A和B。然后：

1. 计算并输出A+B。
2. 计算det(A+B), det(A), det(B)，检查是否相等。
3. 检查det(A+B)是否等于det(A)\*det(B)。
4. 计算A的逆矩阵，然后计算其行列式，与1/det(A)比较。
5. 行列式的计算函数：int determinant\_3x3(int matrix[3][3])，返回行列式的值。

矩阵相加函数类似之前的。

矩阵求逆函数：对于3x3矩阵，可以用公式法。假设矩阵可逆，即行列式不为零。求逆的步骤包括计算伴随矩阵，然后除以行列式。例如，伴随矩阵是余因子矩阵的转置，每个余因子是相应的子行列式乘以(-1)^(i+j)。

但需要注意，当行列式为零时，矩阵不可逆，此时第4步无法进行。因此，在实验八中可能需要处理这种情况。比如，如果生成的A的行列式为零，则重新生成，或者跳过该次测试。或者允许用户多次尝试直到生成可逆的矩阵。

步骤一：生成A和B，输出A+B。这一步比较简单。

步骤二：计算det(A+B)和det(A)+det(B)，比较是否相等。一般来说，这不成立，所以预期结果是否定的。

步骤三：计算det(A+B)和det(A)\*det(B)，比较是否相等。同样，一般不成立，所以预期也是否定的。

步骤四：计算det(A⁻¹)是否等于1/det(A)。这应该总是成立的，只要A可逆。

**四、程序代码**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#define SIZE 3

void generate\_matrix(int matrix[SIZE][SIZE]);

void print\_matrix(int matrix[SIZE][SIZE]);

void add\_matrices(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE]);

int determinant(int matrix[SIZE][SIZE]);

void inverse\_matrix(int matrix[SIZE][SIZE], double inv[SIZE][SIZE]);

double determinant\_double(double matrix[SIZE][SIZE]);

int main()

{

    srand(time(NULL));

    int A[SIZE][SIZE], B[SIZE][SIZE], sum[SIZE][SIZE];

    int detA, detB, detSum;

    double invA[SIZE][SIZE];

    double detInvA;

    // Generate A until it's invertible

    do

    {

        generate\_matrix(A);

    } while (determinant(A) == 0);

    generate\_matrix(B); // B can be singular

    printf("Matrix A:\n");

    print\_matrix(A);

    printf("Matrix B:\n");

    print\_matrix(B);

    // Part 1: A + B

    add\_matrices(A, B, sum);

    printf("A + B:\n");

    print\_matrix(sum);

    // Part 2: det(A+B) vs det(A) + det(B)

    detSum = determinant(sum);

    detA = determinant(A);

    detB = determinant(B);

    printf("det(A+B) = %d\n", detSum);

    printf("det(A) + det(B) = %d\n", detA + detB);

    printf("det(A+B) == det(A)+det(B)? %s\n", detSum == detA + detB ? "Yes" : "No");

    // Part 3: det(A+B) vs det(A)\*det(B)

    printf("det(A)\*det(B) = %d\n", detA \* detB);

    printf("det(A+B) == det(A)\*det(B)? %s\n", detSum == detA \* detB ? "Yes" : "No");

    // Part 4: det(A⁻¹) vs 1/det(A)

    inverse\_matrix(A, invA);

    detInvA = determinant\_double(invA);

    printf("det(A⁻¹) = %.5f\n", detInvA);

    printf("1/det(A) = %.5f\n", 1.0 / detA);

    printf("det(A⁻¹) == 1/det(A)? %s\n", fabs(detInvA - 1.0 / detA) < 1e-6 ? "Yes" : "No");

    return 0;

}

void generate\_matrix(int matrix[SIZE][SIZE])

{

    for (int i = 0; i < SIZE; i++)

    {

        for (int j = 0; j < SIZE; j++)

        {

            matrix[i][j] = rand() % 10 + 1; // Random integers 1-10

        }

    }

}

void print\_matrix(int matrix[SIZE][SIZE])

{

    for (int i = 0; i < SIZE; i++)

    {

        for (int j = 0; j < SIZE; j++)

        {

            printf("%4d ", matrix[i][j]);

        }

        printf("\n");

    }

}

void add\_matrices(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE])

{

    for (int i = 0; i < SIZE; i++)

    {

        for (int j = 0; j < SIZE; j++)

        {

            result[i][j] = A[i][j] + B[i][j];

        }

    }

}

int determinant(int matrix[SIZE][SIZE])

{

    int a = matrix[0][0], b = matrix[0][1], c = matrix[0][2];

    int d = matrix[1][0], e = matrix[1][1], f = matrix[1][2];

    int g = matrix[2][0], h = matrix[2][1], i = matrix[2][2];

    return a \* (e \* i - f \* h) - b \* (d \* i - f \* g) + c \* (d \* h - e \* g);

}

void inverse\_matrix(int matrix[SIZE][SIZE], double inv[SIZE][SIZE])

{

    int det = determinant(matrix);

    double inv\_det = 1.0 / det;

    // Compute cofactor matrix

    double cofactor[SIZE][SIZE];

    for (int i = 0; i < SIZE; i++)

    {

        for (int j = 0; j < SIZE; j++)

        {

            int sign = (i + j) % 2 == 0 ? 1 : -1;

            int submatrix[2][2];

            int sub\_i = 0;

            for (int row = 0; row < SIZE; row++)

            {

                if (row == i)

                    continue;

                int sub\_j = 0;

                for (int col = 0; col < SIZE; col++)

                {

                    if (col == j)

                        continue;

                    submatrix[sub\_i][sub\_j] = matrix[row][col];

                    sub\_j++;

                }

                sub\_i++;

            }

            int minor\_det = submatrix[0][0] \* submatrix[1][1] - submatrix[0][1] \* submatrix[1][0];

            cofactor[i][j] = sign \* minor\_det;

        }

    }

    // Transpose cofactor to get adjugate

    double adjugate[SIZE][SIZE];

    for (int i = 0; i < SIZE; i++)

    {

        for (int j = 0; j < SIZE; j++)

        {

            adjugate[i][j] = cofactor[j][i];

        }

    }

    // Multiply by inverse determinant

    for (int i = 0; i < SIZE; i++)

    {

        for (int j = 0; j < SIZE; j++)

        {

            inv[i][j] = adjugate[i][j] \* inv\_det;

        }

    }

}

double determinant\_double(double matrix[SIZE][SIZE])

{

    double a = matrix[0][0], b = matrix[0][1], c = matrix[0][2];

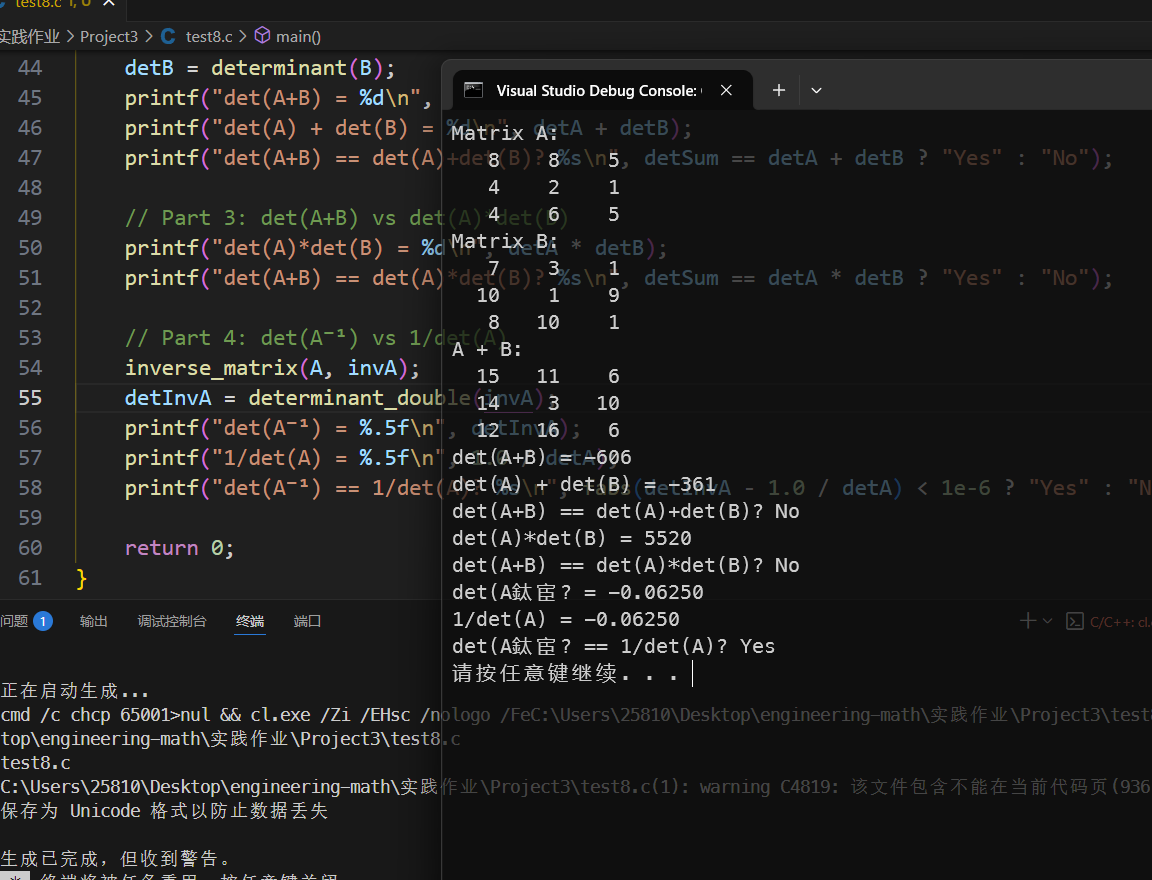
    double d = matrix[1][0], e = matrix[1][1], f = matrix[1][2];

    double g = matrix[2][0], h = matrix[2][1], i = matrix[2][2];

    return a \* (e \* i - f \* h) - b \* (d \* i - f \* g) + c \* (d \* h - e \* g);

}

**五、数据输入与运行结果（截图展示）**



**六、总结**

实验八通过生成随机3×3矩阵，验证了矩阵运算的仿真性质。结果显示：

1. det(A+B) 通常不等于 det(A) + det(B) 或 det(A) × det(B)。
2. 逆矩阵的行列式满足 det(A⁻¹) = 1/det(A)，符合理论预期。这表明矩阵求逆及行列式计算的正确性。