实验三 Eigen 的基本使用

2151094 宋正非

1. 实验目标

（1）熟悉 Eigen 的基本运算

（2）熟悉基本的线性代数运算

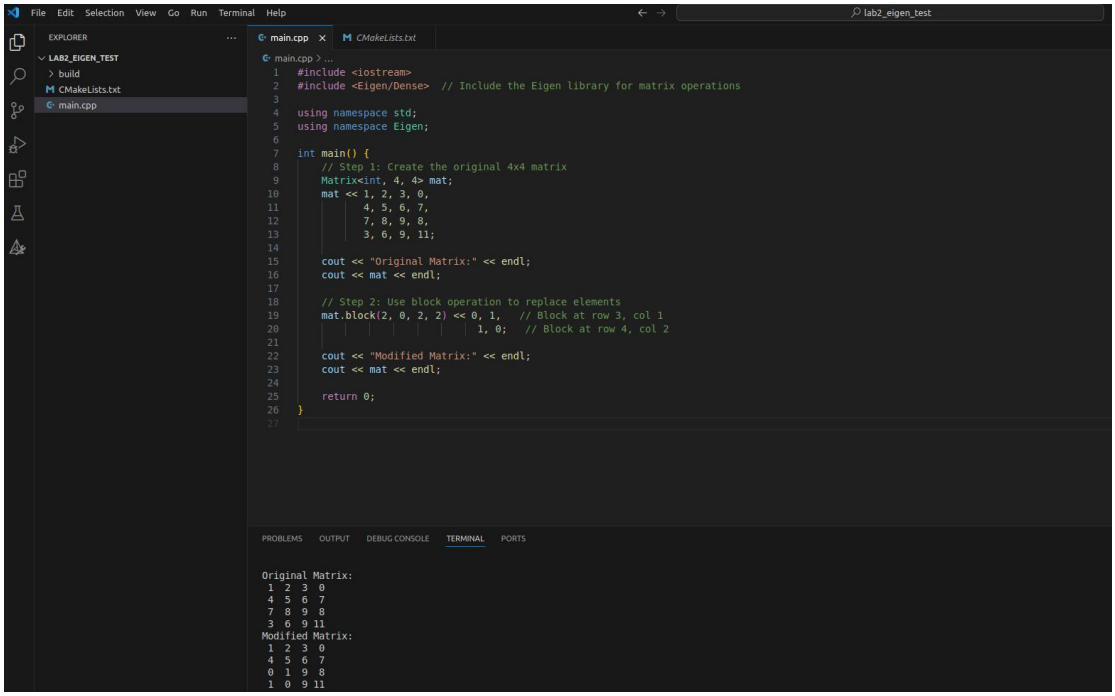
（3）熟悉 OpenCV 和 Eigen 之间的联动操作

2. Eigen 的基本运算

首先创建一个矩阵，使用 block 操作来修改矩阵的元素值，替换第三四行、第一二列的元素，得到新的矩阵。

```
#include <iostream>
#include <Eigen/Dense>  // Include the Eigen library for matrix operations

using namespace std;
using namespace Eigen;

int main() {
    // Step 1: Create the original 4x4 matrix
    Matrix<int, 4, 4> mat;
    mat << 1, 2, 3, 0,
           4, 5, 6, 7,
           7, 8, 9, 8,
           3, 6, 9, 11;

    cout << "Original Matrix:" << endl;
    cout << mat << endl;

    // Step 2: Use block operation to replace elements
    mat.block(2, 0, 2, 2) << 0, 1,   // Block at row 3, col 1
                             1, 0;   // Block at row 4, col 2

    cout << "Modified Matrix:" << endl;
    cout << mat << endl;

    return 0;
}
```

其中，block(start_row, start_col, num_rows, num_cols)的规则如下：start_row 是选定子矩阵开始的行索引，start_col 是选定子矩阵开始的列索引，eigen 的行列都是从 0 开始索引的；num_rows 是要选定的子矩阵的行数，num_cols 是要选定的子矩阵的列数。mat.block(2, 0, 2, 2)表示选择原始矩阵 mat 中从第 3 行第 1 列开始的一个 2×2 的子矩阵。“<<”是 eigen 用于批量赋值的语法，可以快速地将一系列值赋给选定的矩阵块。

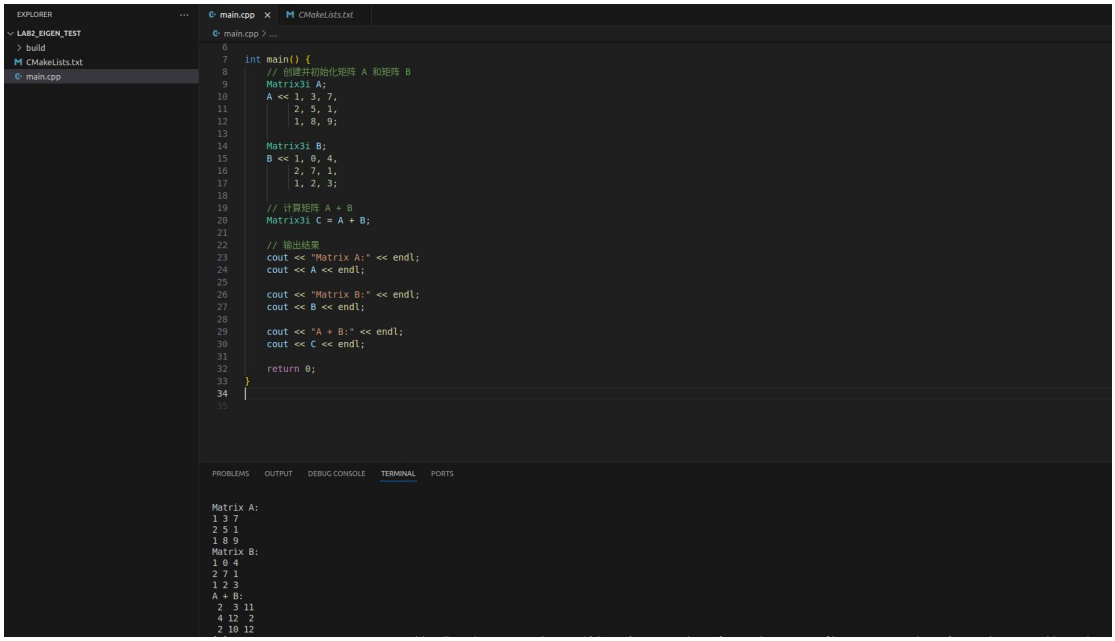


3. 基本线性代数运算

（1）矩阵加减

```
#include <iostream>
#include <Eigen/Dense>

using namespace std;
using namespace Eigen;

int main() {
    Matrix3i A;
    A << 1, 3, 7,
         2, 5, 1,
         1, 8, 9;

    Matrix3i B;
    B << 1, 0, 4,
         2, 7, 1,
         1, 2, 3;

    Matrix3i C = A + B;

    cout << "Matrix A:" << endl;
    cout << A << endl;

    cout << "Matrix B:" << endl;
    cout << B << endl;

    cout << "A + B:" << endl;
    cout << C << endl;

    return 0;
}
```

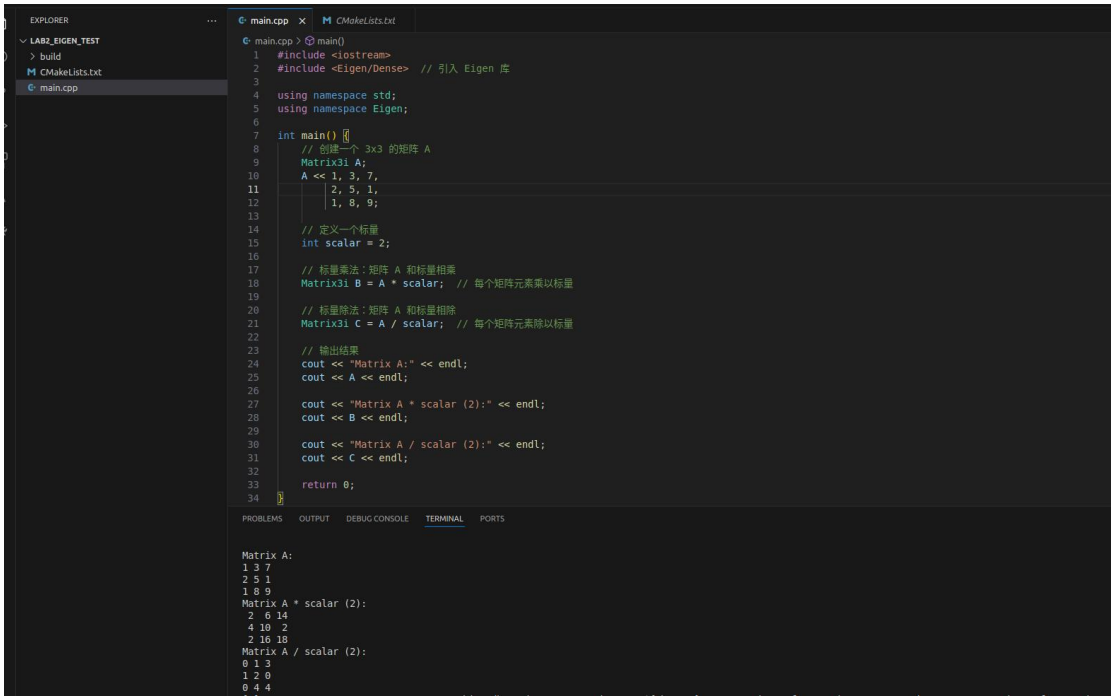
其中，Matrix3i C = A + B 这一行是 Eigen 中的矩阵加法运算。



（2）标量乘除

```
#include <iostream>
#include <Eigen/Dense>
using namespace std;
using namespace Eigen;

int main() {
    Matrix3i A;
    A << 1, 3, 7,
         2, 5, 1,
         1, 8, 9;

    int scalar = 2;

    Matrix3i B = A * scalar;
    Matrix3i C = A / scalar;

    cout << "Matrix A:" << endl;
    cout << A << endl;

    cout << "Matrix A * scalar (2):" << endl;
    cout << B << endl;

    cout << "Matrix A / scalar (2):" << endl;
    cout << C << endl;

    return 0;
}
```

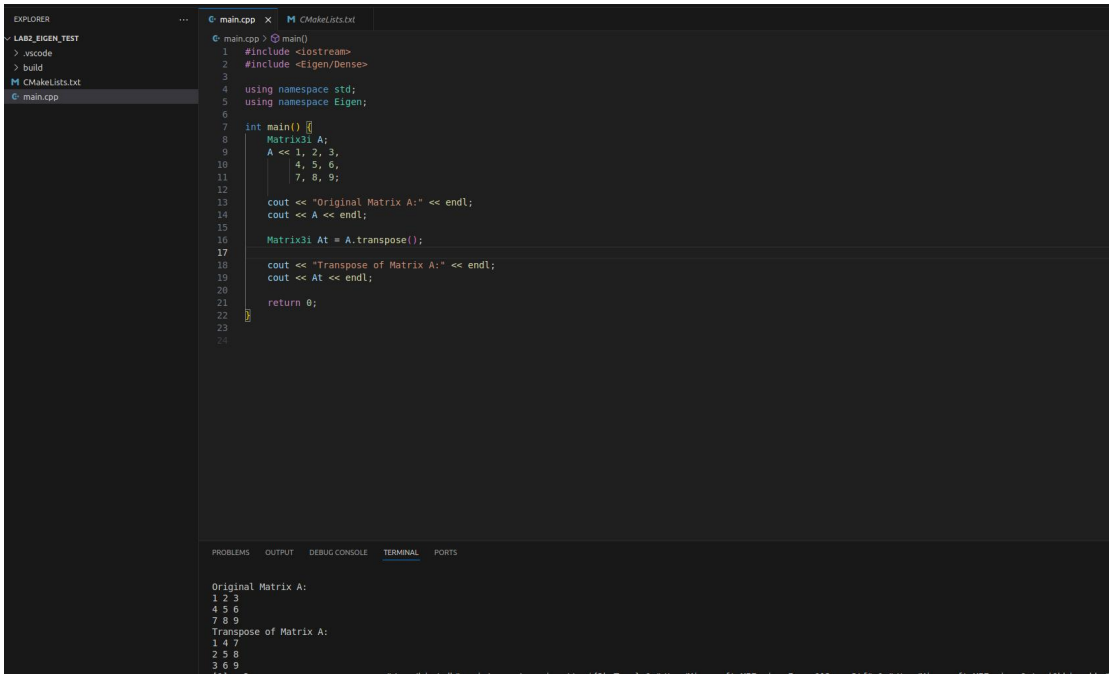
通过这里的标量除法运算，我发现当矩阵 A 和标量 scalar 都是整数类型时，Eigen 会执行整数除法，除法结果会被取整，即小数部分被丢弃。



（3）转置

```
#include <iostream>
#include <Eigen/Dense>

using namespace std;
using namespace Eigen;

int main() {
    Matrix3i A;
    A << 1, 2, 3,
         4, 5, 6,
         7, 8, 9;

    cout << "Original Matrix A:" << endl;
    cout << A << endl;

    Matrix3i At = A.transpose();

    cout << "Transpose of Matrix A:" << endl;
    cout << At << endl;

    return 0;
}
```

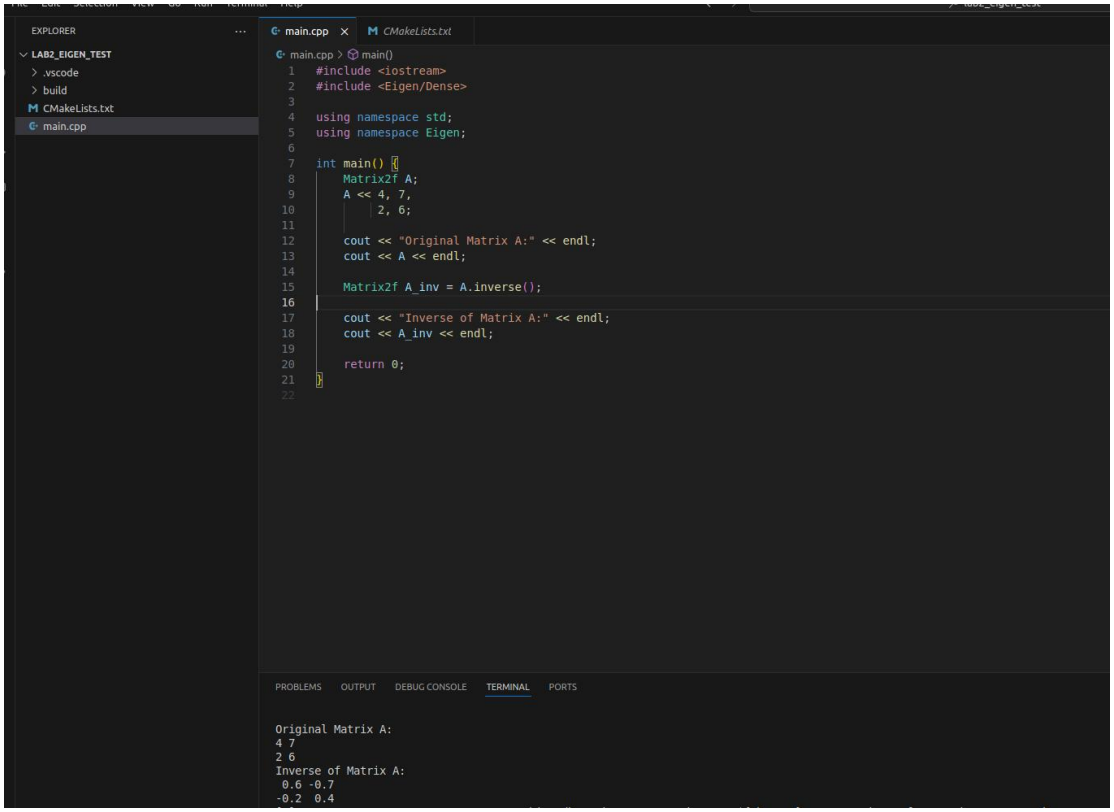
Eigen::Matrix<>.transpose() 是 Eigen 库中用于执行矩阵转置操作的一个函数。transpose() 用于返回矩阵的转置。它返回一个新的矩阵对象，该矩阵是原矩阵的转置，而不会修改原始矩阵。



（4）求逆

```
#include <iostream>
#include <Eigen/Dense>

using namespace std;
using namespace Eigen;

int main() {
    Matrix2f A;
    A << 4, 7,
         2, 6;

    cout << "Original Matrix A:" << endl;
    cout << A << endl;

    Matrix2f A_inv = A.inverse();

    cout << "Inverse of Matrix A:" << endl;
    cout << A_inv << endl;

    return 0;
}
```

Eigen::Matrix<>.inverse() 作用：inverse() 是 Matrix 类的成员函数，用于计算矩阵的逆。如果矩阵是非奇异的（行列式不为零），则它有一个逆矩阵，inverse() 将返回这个逆矩阵。如果矩阵是奇异的（即行列式为零），则无法计算逆矩阵，inverse() 会抛出一个错误。

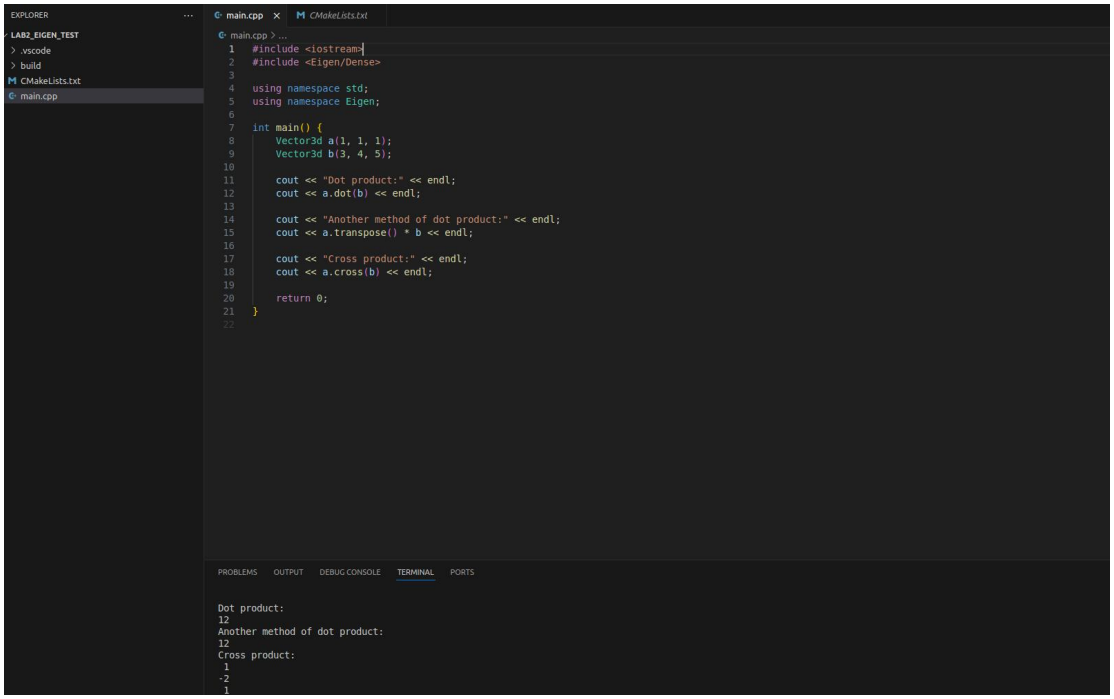


（5）矩阵相乘

```
#include <iostream>
#include <Eigen/Dense>

using namespace std;
using namespace Eigen;

int main() {
    Vector3d a(1, 1, 1);
    Vector3d b(3, 4, 5);

    cout << "Dot product:" << endl;
    cout << a.dot(b) << endl;

    cout << "Another method of dot product:" << endl;
    cout << a.transpose() * b << endl;

    cout << "Cross product:" << endl;
    cout << a.cross(b) << endl;

    return 0;
}
```

点积：a.dot(b)， 直接调用 dot 方法计算两个向量的点积; a.transpose() * b：另一种计算点积的方式，先将 a 转置，然后与 b 做矩阵乘法，得到点积的结果。

叉积：a.cross(b)，直接调用 cross 方法来计算两个三维向量的叉积。



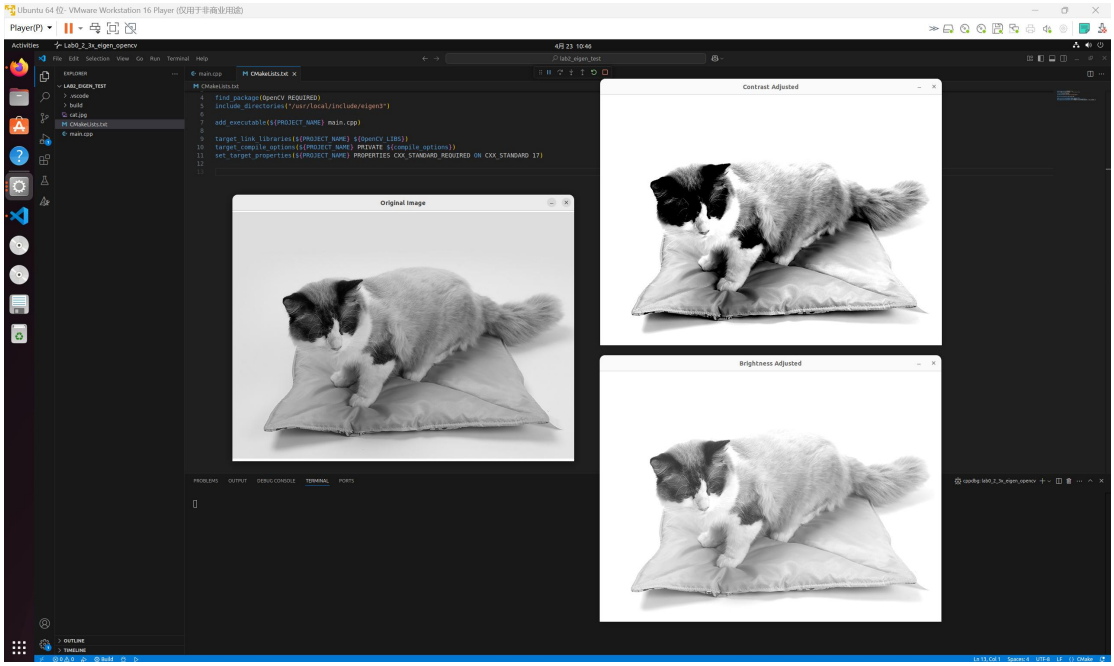
4. OpenCV 和 Eigen 联动

（1）将 OpenCV Mat 转化为 Eigen 矩阵

```
#include <Eigen/Dense>
#include <Eigen/Core>
#include <opencv2/core/eigen.hpp>
```

（2）使用 Eigen 函数处理图像

使用 OpenCV 读取图像文件并创建数组，将其转换为 eigen 矩阵，并对图像进行操作。得到对比度调整、亮度调整后的图像如下图所示，效果较为明显。

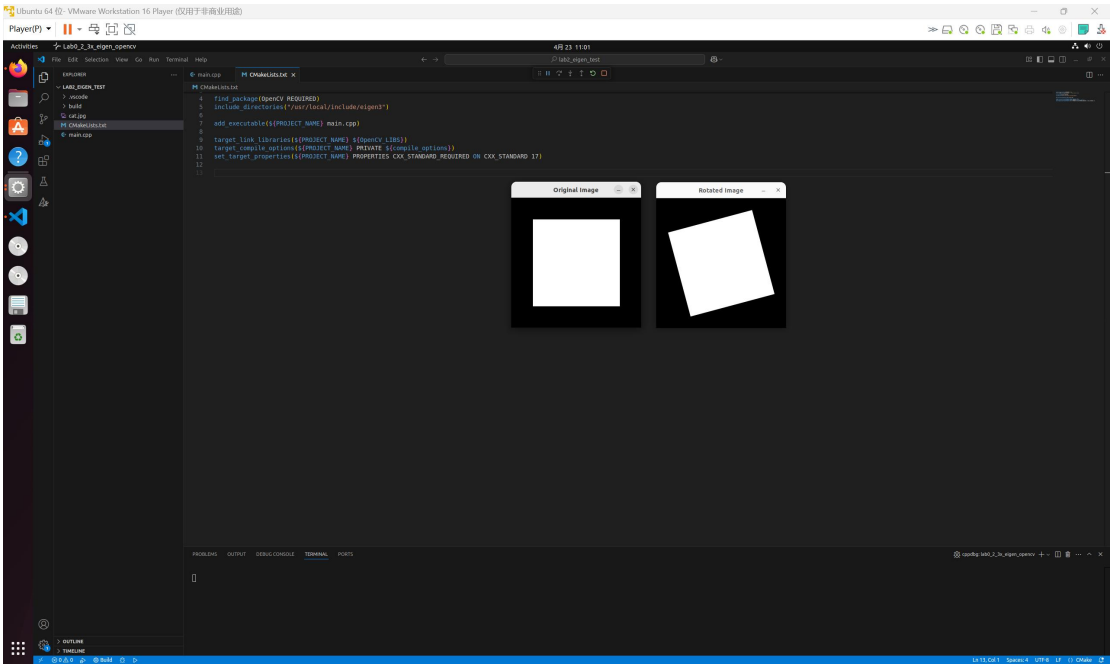


```
#include <Eigen/Dense>
#include <Eigen/Core>
#include <opencv2/core/eigen.hpp>
#include <opencv2/opencv.hpp>

// Function to adjust brightness
Eigen::MatrixXd adjustBrightness(const Eigen::MatrixXd& image, double alpha) {
    Eigen::MatrixXd result = image.array() + alpha;  // Add alpha to each pixel to adjust brightness
    result = result.cwiseMax(0.0);  // Ensure the pixel values are non-negative
    result = result.cwiseMin(255.0);  // Ensure the pixel values do not exceed 255
    return result;
}

// Function to adjust contrast
Eigen::MatrixXd adjustContrast(const Eigen::MatrixXd& image, double contrastFactor) {
    Eigen::MatrixXd result = (image.array() - 128.0) * contrastFactor + 128.0;  // Adjust contrast
    result = result.cwiseMax(0.0);  // Ensure the pixel values are non-negative
    result = result.cwiseMin(255.0);  // Ensure the pixel values do not exceed 255
    return result;
}

int main() {
    cv::Mat img = cv::imread("/home/jeff/work/lab2_eigen_test/cat.jpg", cv::IMREAD_GRAYSCALE);
// Read in grayscale

    if (img.empty()) {
        std::cerr << "Could not open or find the image!" << std::endl;
        return -1;
    }

    // Convert cv::Mat to Eigen matrix
    Eigen::MatrixXd eigenImg(img.rows, img.cols);
    for (int i = 0; i < img.rows; ++i) {
        for (int j = 0; j < img.cols; ++j) {
            eigenImg(i, j) = img.at<uchar>(i, j);
        }
    }

    // Adjust brightness and contrast
    double brightnessAlpha = 50.0;
    double contrastFactor = 2;

    Eigen::MatrixXd brightImg = adjustBrightness(eigenImg, brightnessAlpha);
    Eigen::MatrixXd contrastImg = adjustContrast(eigenImg, contrastFactor);

    // Convert Eigen matrix back to cv::Mat for displaying
    cv::Mat brightMat(brightImg.rows(), brightImg.cols(), CV_8UC1);
    cv::Mat contrastMat(contrastImg.rows(), contrastImg.cols(), CV_8UC1);

    for (int i = 0; i < brightImg.rows(); ++i) {
        for (int j = 0; j < brightImg.cols(); ++j) {
            brightMat.at<uchar>(i, j) = static_cast<uchar>(brightImg(i, j));
            contrastMat.at<uchar>(i, j) = static_cast<uchar>(contrastImg(i, j));
        }
    }

    // Display
    cv::imshow("Original Image", img);
    cv::imshow("Brightness Adjusted", brightMat);
    cv::imshow("Contrast Adjusted", contrastMat);
```

```
    cv::waitKey(0);

    return 0;
}
```

（3）创建图像并使用 Eigen 函数逆时针旋转 15 度

使用 OpenCV 绘制矩形，将其转换为 eigen 矩阵，并应用旋转函数生成旋转后的图像，图像逆时针旋转 15 度。




```
#include <Eigen/Dense>
#include <Eigen/Core>
#include <opencv2/core/eigen.hpp>
#include <opencv2/opencv.hpp>

Eigen::MatrixXd rotateImage(const Eigen::MatrixXd& original, double angle) {
    double angle_rad = angle * M_PI / 180.0;

    Eigen::Matrix2d rotationMatrix;
    rotationMatrix << cos(angle_rad), -sin(angle_rad),
                      sin(angle_rad), cos(angle_rad);

    int rows = original.rows();
    int cols = original.cols();

    int newRows = std::round(abs(cos(angle_rad) * rows) + abs(sin(angle_rad) * cols));
    int newCols = std::round(abs(sin(angle_rad) * rows) + abs(cos(angle_rad) * cols));

    Eigen::MatrixXd rotatedImage(newRows, newCols);
    rotatedImage.setZero();

    int centerX = newCols / 2;
    int centerY = newRows / 2;

    for (int i = 0; i < rows; ++i) {
```

```
        for (int j = 0; j < cols; ++j) {
            int newI = centerY + std::round(rotationMatrix(0, 0) * (i - rows / 2) + rotationMatrix(0, 1) * (j - cols / 2));
            int newJ = centerX + std::round(rotationMatrix(1, 0) * (i - rows / 2) + rotationMatrix(1, 1) * (j - cols / 2));

            if (newI >= 0 && newI < newRows && newJ >= 0 && newJ < newCols) {
                rotatedImage(newI, newJ) = original(i, j);
            }
        }
    }

    return rotatedImage;
}

int main() {
    cv::Mat img(300, 300, CV_8UC1, cv::Scalar(0));
    cv::rectangle(img, cv::Point(50, 50), cv::Point(250, 250), cv::Scalar(255), -1);

    cv::Point2f center(img.cols / 2.0, img.rows / 2.0);
    cv::Mat rotationMatrix = cv::getRotationMatrix2D(center, 15, 1.0);

    cv::Mat rotatedImg;
    cv::warpAffine(img, rotatedImg, rotationMatrix, img.size(), cv::INTER_LINEAR, cv::BORDER_CONSTANT, cv::Scalar(0));

    cv::imshow("Original Image", img);
    cv::imshow("Rotated Image", rotatedImg);
    cv::waitKey(0);

    return 0;
}
```

（4）（延伸任务）用 OpenCV 读取图像并使用 Eigen 应用高斯模糊算法

使用 Eigen 函数创建一个高斯模糊算子，其矩阵维度为 5×5，sigma=10.0。使用 OpenCV 导入一张 512×512 大小的图像，转换为 Eigen 矩阵，使用 Eigen 函数应用高斯算子。然后将其反向转换为 OpenCV 图像并显示。

高斯模糊通过高斯函数对每一个像素点进行加权平均，将其与周围的像素混合，使图像显得更加平滑柔和。高斯函数可表示为：

$$G(x) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2}{2\sigma^2}}$$

扩展到二维空间可用于处理图像，二维高斯函数可表示为：

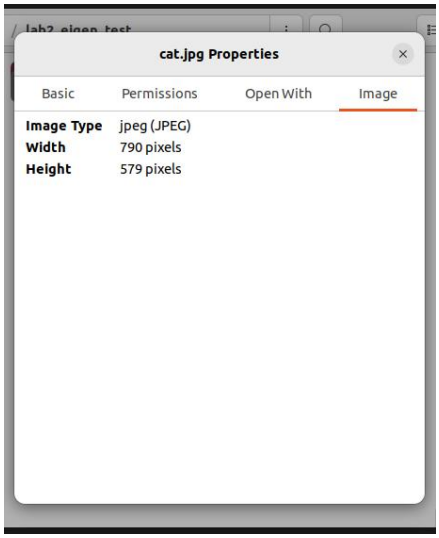
$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

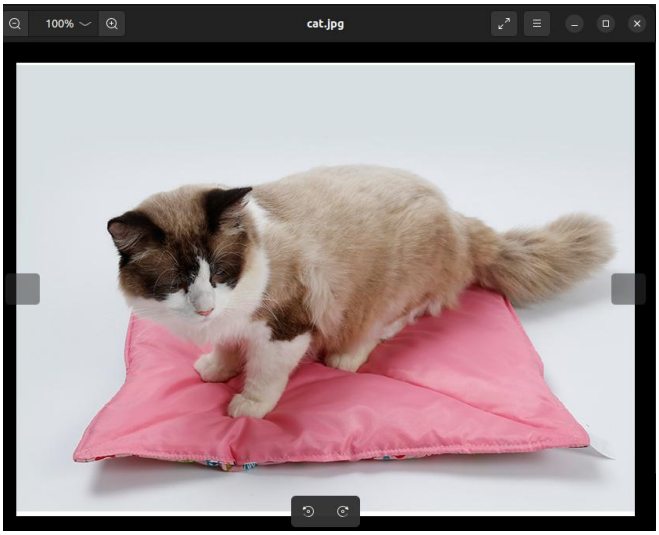
其中，x 和 y 是相对中心点的位置坐标。σ是标准差，决定模糊的程度；σ越大，模糊效果越强。

高斯模糊的实现是用高斯核的小窗口对图像进行逐点扫描，可表示为：

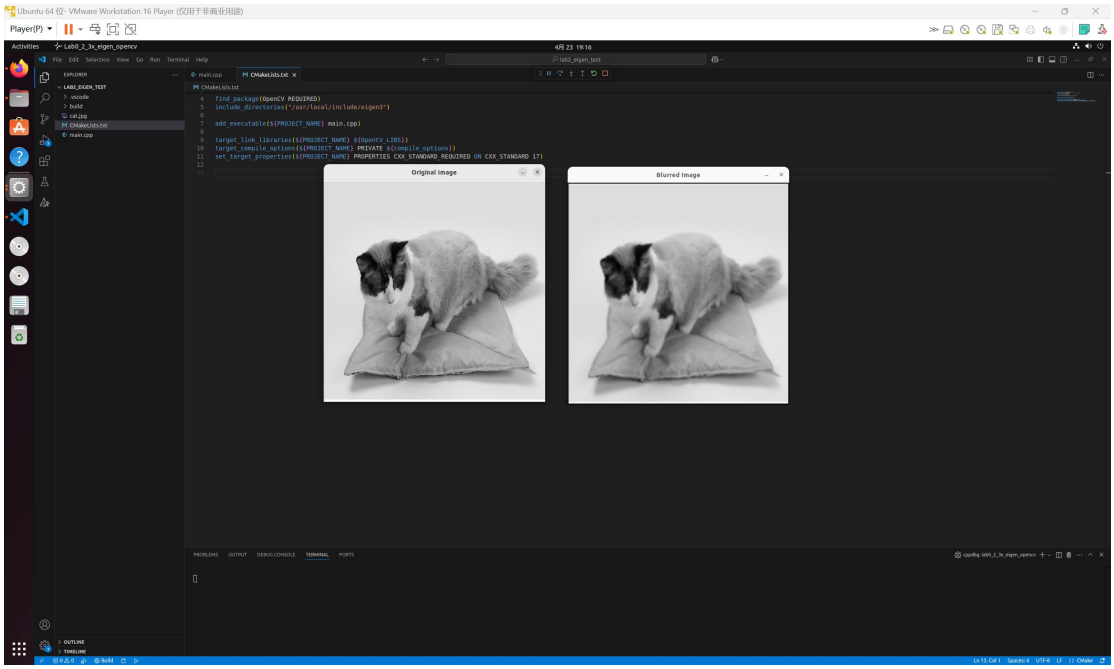
$$I_{blurred} = \sum_{i=-k}^{k} \sum_{i=-k}^{k} I(x+i, y+j) \cdot K(i,j)$$

其中 I 为原图像，K 为核函数。





由于我的图像的原始尺寸为 790px×579px，因此在导入后先修改尺寸为要求的 512×512 尺寸。实现效果与代码如下。




```
#include <Eigen/Dense>
#include <Eigen/Core>
#include <opencv2/core/eigen.hpp>
#include <opencv2/opencv.hpp>
#include <iostream>
#include <cmath>

// Create Gaussian Kernel
Eigen::MatrixXd createGaussianKernel(int size, double sigma) {
    Eigen::MatrixXd kernel(size, size);
    int halfSize = size / 2;
```

```
    double sum = 0.0;

    for (int i = -halfSize; i <= halfSize; ++i) {
        for (int j = -halfSize; j <= halfSize; ++j) {
            double exponent = -(i * i + j * j) / (2 * sigma * sigma);
            kernel(i + halfSize, j + halfSize) = exp(exponent) / (2 * M_PI * sigma * sigma);
            sum += kernel(i + halfSize, j + halfSize);
        }
    }

    kernel /= sum;

    return kernel;
}

//Gaussian Blur
Eigen::MatrixXd applyGaussianBlur(const Eigen::MatrixXd& image, const Eigen::MatrixXd& kernel)
{
    int kernelSize = kernel.rows();
    int halfSize = kernelSize / 2;
    int rows = image.rows();
    int cols = image.cols();
    Eigen::MatrixXd result(rows, cols);

    for (int i = halfSize; i < rows - halfSize; ++i) {
        for (int j = halfSize; j < cols - halfSize; ++j) {
            double value = 0.0;
            for (int ki = -halfSize; ki <= halfSize; ++ki) {
                for (int kj = -halfSize; kj <= halfSize; ++kj) {
                    value += image(i + ki, j + kj) * kernel(ki + halfSize, kj + halfSize);
                }
            }
            result(i, j) = value;
        }
    }

    return result;
}

int main() {
    cv::Mat img = cv::imread("/home/jeff/work/lab2_eigen_test/cat.jpg", cv::IMREAD_GRAYSCALE);
// Load image in grayscale
    if (img.empty()) {
        std::cerr << "Could not open or find the image!" << std::endl;
        return -1;
    }
    cv::resize(img, img, cv::Size(512, 512));

    // Convert to Eigen matrix
    Eigen::MatrixXd eigenImg(img.rows, img.cols);
    for (int i = 0; i < img.rows; ++i) {
        for (int j = 0; j < img.cols; ++j) {
            eigenImg(i, j) = static_cast<double>(img.at<uchar>(i, j));
        }
    }

    // Gaussian KerneL
    Eigen::MatrixXd gaussianKernel = createGaussianKernel(5, 10.0);  // 5x5 kernel, sigma=10.0

    // Gaussian Blur
```

```
    Eigen::MatrixXd blurredImg = applyGaussianBlur(eigenImg, gaussianKernel);

    // Back to OpenCV Mat
    cv::Mat blurredMat(img.rows, img.cols, CV_8UC1);
    for (int i = 0; i < img.rows; ++i) {
        for (int j = 0; j < img.cols; ++j) {
            blurredMat.at<uchar>(i, j) = static_cast<uchar>(std::min(255.0, std::max(0.0, blurredImg(i,
j))));
        }
    }

    cv::imshow("Original Image", img);
    cv::imshow("Blurred Image", blurredMat);

    cv::waitKey(0);

    return 0;
}
```

在此基础上，需要进行一些额外的工作来评估高斯模糊算法的效果。使用 DFT 离散傅里叶变换分析并比较原始图像和处理后图像的频域特性。DFT 用于将图像从空间域转换到频域，便于观察低频、高频成分的分布。DFT 可用公式表示为：

$$F(u,v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-j2\pi(\frac{u_x}{M}+\frac{v_y}{N})}$$

其中，f(x,y)为原始图像在空间域中的像素值；F(u,v)为图像在频域中的频率分量；M 和 N 为图像的宽度和高度。

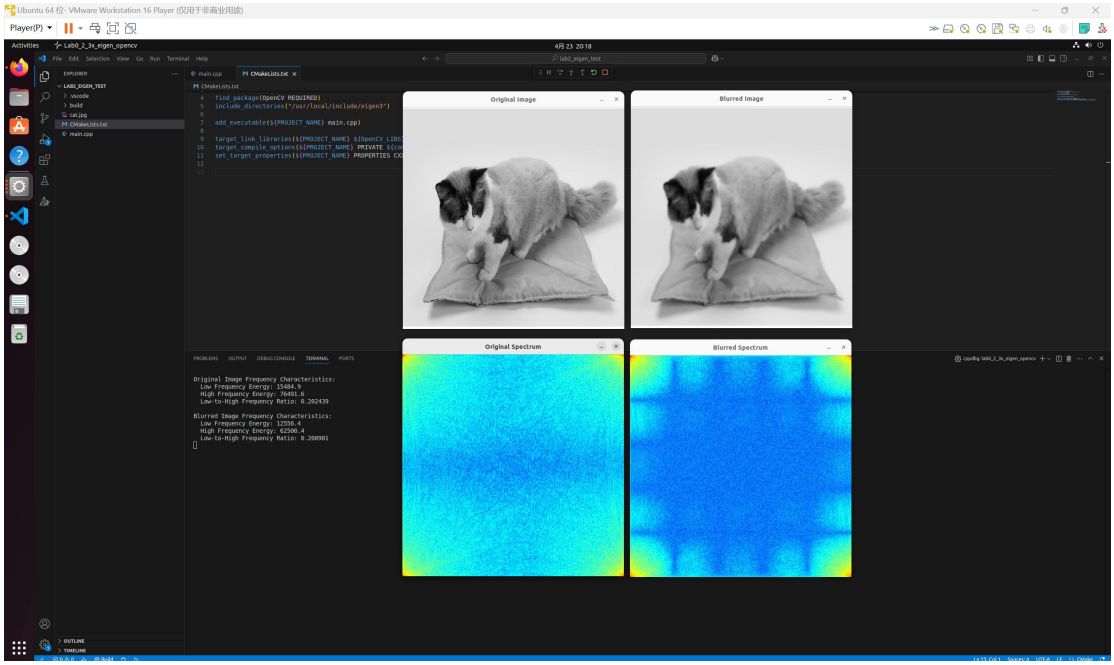
增加可视化函数来计算和显示频域特性。通过傅里叶变化从时域转换到频域，解释图像中不同频率的成分。通过对数变化和颜色映射，频谱中的高低频部分都可以被清晰地可视化，便于观察分析。同时，计算高频能量、低频能量、能量均值与方差进行频域特征计算。[1]

```
cv::Mat visualizeDFT(const cv::Mat& img, cv::Mat& magnitudeFloat) {
    // Convert input image to floating-point matrix
    cv::Mat planes[] = {cv::Mat_<float>(img), cv::Mat::zeros(img.size(), CV_32F)};
    cv::Mat complexImg;
    cv::merge(planes, 2, complexImg);

    // Perform DFT
    cv::dft(complexImg, complexImg);

    // Split into real and imaginary parts
    cv::split(complexImg, planes);
    cv::magnitude(planes[0], planes[1], magnitudeFloat);

    // Apply logarithmic scaling and normalize the magnitude
```

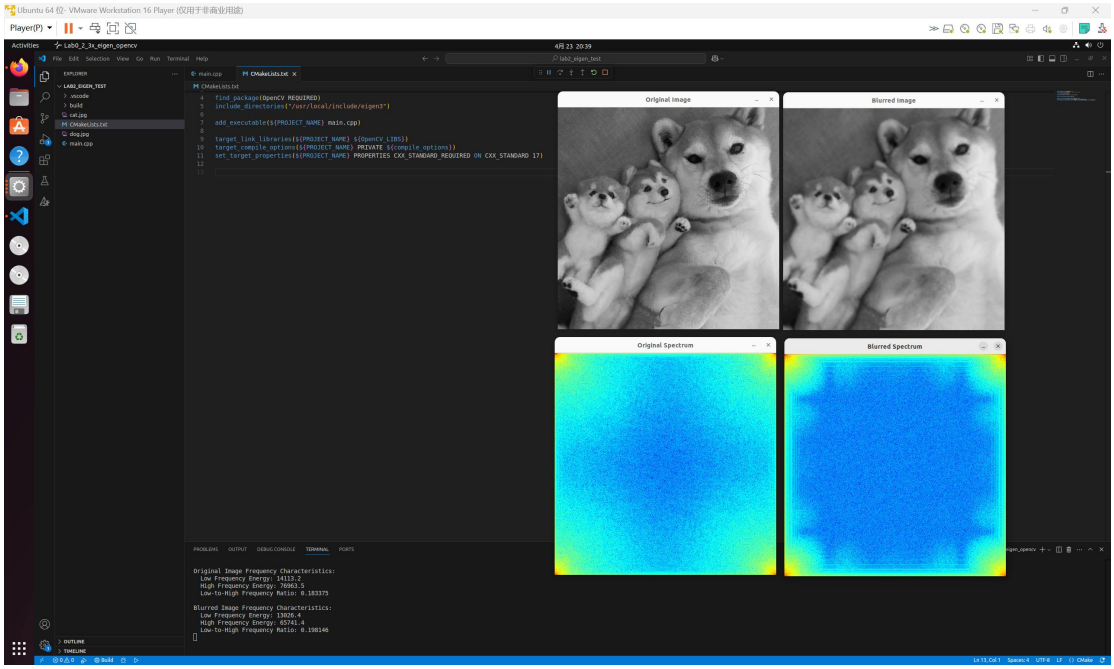
[1] 参考网页 https://blog.csdn.net/BIYing_Aurora/article/details/146398642?spm=1001.2014.3001.5502

```
    cv::log(magnitudeFloat + 1, magnitudeFloat);
    cv::normalize(magnitudeFloat, magnitudeFloat, 0, 1, cv::NORM_MINMAX);

    // Convert magnitude to 8-bit and apply color map
    cv::Mat magnitude8U;
    magnitudeFloat.convertTo(magnitude8U, CV_8U, 255);
    cv::Mat magnitudeColorMap;
    cv::applyColorMap(magnitude8U, magnitudeColorMap, cv::COLORMAP_JET);

    return magnitudeColorMap;  // Return the color-mapped DFT magnitude image
}
```

```
// Calculate frequency domain features
void calculateFrequencyFeatures(const cv::Mat& magnitudeImg, float& lowFreqEnergy, float&
highFreqEnergy) {
    int cx = magnitudeImg.cols / 2;
    int cy = magnitudeImg.rows / 2;
    int radius = std::min(cx, cy) / 2;

    lowFreqEnergy = 0.0f;
    highFreqEnergy = 0.0f;

    CV_Assert(magnitudeImg.type() == CV_32F);

    for (int i = 0; i < magnitudeImg.rows; ++i) {
        for (int j = 0; j < magnitudeImg.cols; ++j) {
            float value = magnitudeImg.at<float>(i, j); // Get magnitude data
            int dist = std::sqrt((i - cy) * (i - cy) + (j - cx) * (j - cx));
            if (dist < radius)
                lowFreqEnergy += value;  // Add to low frequency energy
            else
                highFreqEnergy += value;  // Add to high frequency energy
        }
    }
}
```



根据以上方式进行评估，可得如上图所示结果。高斯模糊作为一个低通滤波器，过滤掉了高频部分，对应图像中快速变化的细节和边缘，减少了高频成分。得到结果高频能量由 76491.6 减少至了高斯模糊后的 62500.4，与理论相符。



更换输入图片，结果如上图所示，高频能量由 76963.5 减少至了高斯模糊后的 65741.4，与理论相符。

（5）（延伸任务）评估 Eigen lib 性能并比较

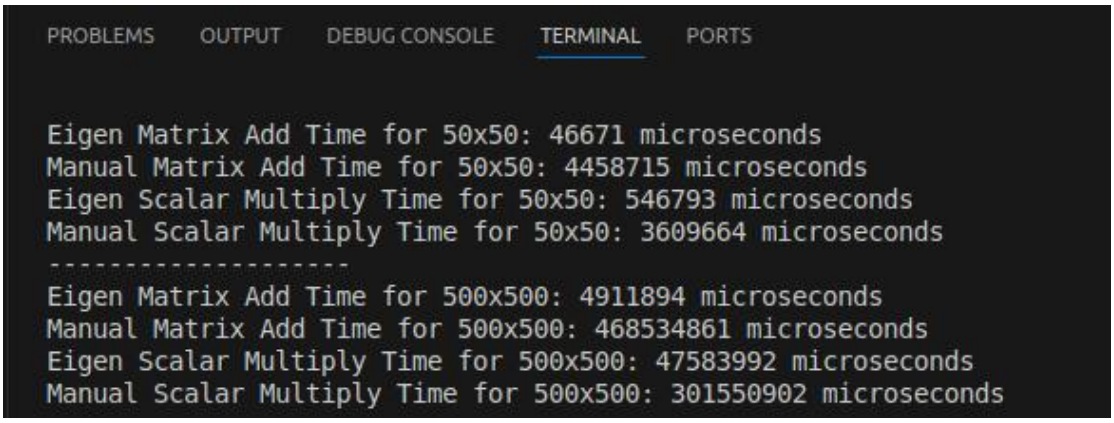
对于任意维度的随机矩阵使用自己的 C++代码实现矩阵加法和标量乘法，并与基于 Eigen 库的操作进行比较。比较两种实现的运行时间时，为排除操作系统或其他进程的影响，统一操作运行多轮计时。

```
#include <iostream>
#include <Eigen/Dense>
#include <cstdlib>
#include <chrono>

using namespace Eigen;
using namespace std;
using namespace std::chrono;

// matrix addition
MatrixXd matrixAddManual(const MatrixXd& A, const MatrixXd& B) {
    int rows = A.rows();
    int cols = A.cols();
    MatrixXd C(rows, cols);

    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            C(i, j) = A(i, j) + B(i, j);
        }
    }
```

```
    return C;
}

// scalar multiplication
MatrixXd scalarMultiplyManual(const MatrixXd& A, double scalar) {
    int rows = A.rows();
    int cols = A.cols();
    MatrixXd C(rows, cols);

    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            C(i, j) = A(i, j) * scalar;
        }
    }
    return C;
}

int main() {
    int dims[] = {50, 500};

    for (int dim : dims) {
        MatrixXd A = MatrixXd::Random(dim, dim);
        MatrixXd B = MatrixXd::Random(dim, dim);
        double scalar = rand() % 100;

        auto start = high_resolution_clock::now();
        for (int i = 0; i < 10000; ++i) {
            MatrixXd C_eigen = A + B;
        }
        auto end = high_resolution_clock::now();
        auto duration_eigen_add = duration_cast<microseconds>(end - start);
        cout << "Eigen Matrix Add Time for " << dim << "x" << dim << ": " <<
duration_eigen_add.count() << " microseconds" << endl;

        start = high_resolution_clock::now();
        for (int i = 0; i < 1000; ++i) {
            MatrixXd C_manual = matrixAddManual(A, B);
        }
        end = high_resolution_clock::now();
        auto duration_manual_add = duration_cast<microseconds>(end - start);
        cout << "Manual Matrix Add Time for " << dim << "x" << dim << ": " <<
duration_manual_add.count() << " microseconds" << endl;

        start = high_resolution_clock::now();
        for (int i = 0; i < 10000; ++i) {
            MatrixXd C_eigen = A * scalar;
        }
        end = high_resolution_clock::now();
        auto duration_eigen_scalar = duration_cast<microseconds>(end - start);
        cout << "Eigen Scalar Multiply Time for " << dim << "x" << dim << ": " <<
duration_eigen_scalar.count() << " microseconds" << endl;

        start = high_resolution_clock::now();
        for (int i = 0; i < 10000; ++i) {
            MatrixXd C_manual = scalarMultiplyManual(A, scalar);
        }
        end = high_resolution_clock::now();
        auto duration_manual_scalar = duration_cast<microseconds>(end - start);
        cout << "Manual Scalar Multiply Time for " << dim << "x" << dim << ": " <<
duration_manual_scalar.count() << " microseconds" << endl;
```

```
    cout << "--------------------" << endl;
  }

  return 0;
}
```

每种操作分别执行 1000 次，以减少系统抖动或其他进程干扰的影响。得到下图所示结果。



对于 50×50 矩阵，基于 eigen 库实现 1000 次矩阵相加总共需要 0.046671 秒，而手动实现矩阵相加需要 4.458715 秒；基于 eigen 库实现 1000 次标量乘法需要 0.546793 秒，而手动实现需要 3.609664 秒。

对于 500×500 矩阵，基于 eigen 库实现 1000 次矩阵相加总共需要 4.911894 秒，而手动实现需要 468.534861 秒；基于 eigen 库实现 1000 次标量乘法需要 47.583992 秒，而手动实现需要 301.550902 秒。

由上对比可知，基于 eigen 库开发可以很大程度缩短矩阵运算的时间。