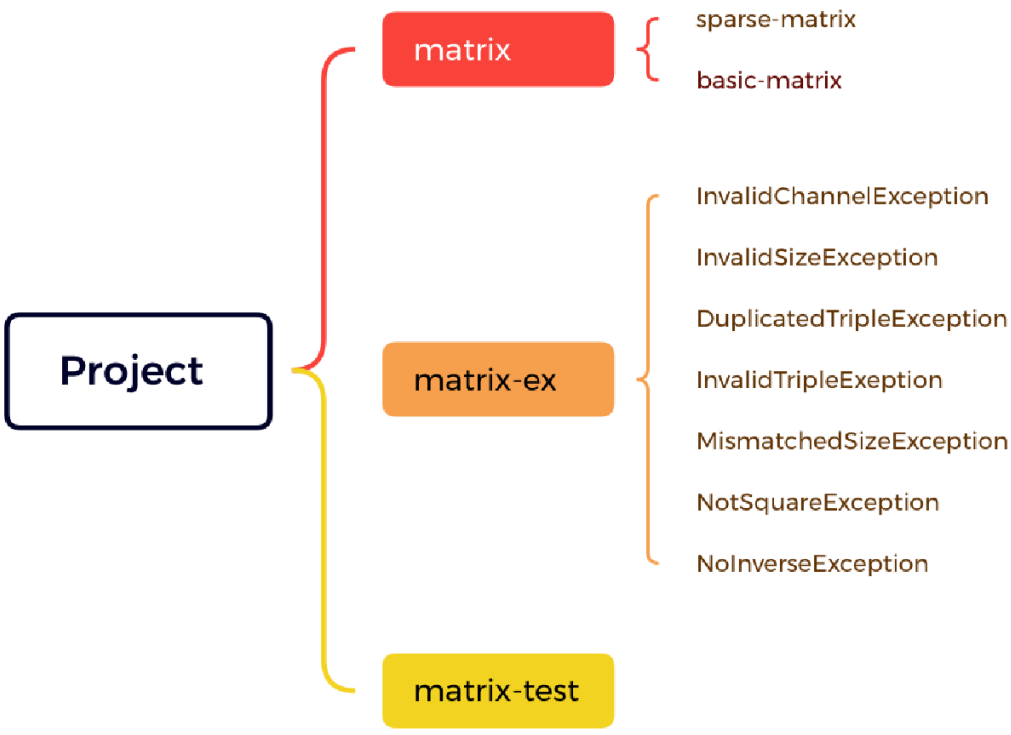
C and C++ Programming Final Project Report

1、项目结构



该项目我们创建了一个模板类Matrix作为整个项目的父类，以方便我们通过多态形式进行以Matrix类指针调用子类对象的方法，Matrix类中我们定义了三个变量，分别为矩阵的row行、col列和val值。在项目中我们设置了两个继承于Matrix父类的模板子类BasicMatrix和SparseMatrix。BasicMatrix是常规的以行列二维形式表示的矩阵，在这里我们通过类模板使用一个一维T*指针作为我们存储行和列的数据的容器。SparseMatrix是用来存储大数据的稀疏矩阵，我们使用c++ STL标准库中的map<int, Triple<T>*>作为容器存储稀疏矩阵中的值，Triple是我们自行定义的一个模板struct作为我们容器map的value值，容器中的key值则是由行和列计算出来的类hash值用于区分map中存储的不同Triple<T>*，Triple中存储了三个值：_row表示行，_col表示列，val表示当前行列对应的值。

Matrix:

Matrix类作为模板父类，我们通过纯虚函数将其设置为了一个抽象类，子类将重载所有设置为virtual的函数。在Matrix类中同样写了公有的setter和getter方法用来设置和获取行、列、尺寸的值。Matrix类的prototype如下所示：

```
template<class T>
    class Matrix {
    private:
        long size;
```

```
    protected:
        int row;
        int col;
    public:

        Matrix(int row, int col) {
            if (row <= 0 || col <= 0) {
                throw ex::InvalidSizeException("creating Matrix", 1, row, col);
            }
            this->row = row;
            this->col = col;
            this->size = this->row * this->col;
        }

        Matrix(const Mat &mat) {
            this->row = mat.rows;
            this->col = mat.cols;
            this->size = this->row * this->col;
        }

        void setSize(const long size) {
            this->size = size;
        }

        void setRow(const int row) {
            this->row = row;
        }

        void setCol(const int col) {
            this->col = col;
        }

        int getSize() const {
            return this->size;
        }

        int getRow() const {
            return this->row;
        }

        int getCol() const {
            return this->col;
        }

        virtual T getByIndex(int _row, int _col) const = 0;

        virtual void setByIndex(int _row, int _col, T val) = 0;

        virtual void scalarMultiply(T) = 0;

        virtual void scalarDivide(T) = 0;

        virtual void transpose() = 0;

        virtual void inverse() = 0;

        virtual void reverse() = 0;
```

```
        virtual void conjugate() = 0;

        virtual T getMax() = 0;

        virtual T getMin() = 0;

        virtual T getSum() = 0;

        virtual T getAvg() = 0;

        virtual T getRowMax(int) = 0;

        virtual T getColMax(int) = 0;

        virtual T getRowMin(int) = 0;

        virtual T getColMin(int) = 0;

        virtual T getRowSum(int) = 0;

        virtual T getColSum(int) = 0;

        virtual T getRowAvg(int) = 0;

        virtual T getColAvg(int) = 0;

        virtual T getTrace() = 0;

        virtual T getDeterminant() = 0;

        virtual void reshape(int row, int col) = 0;

        virtual void sliceRow(int row1, int row2) = 0;

        virtual void sliceCol(int col1, int col2) = 0;

        virtual void slice(int row1, int row2, int col1, int col2) = 0;

        virtual void exponent(int) = 0;

        virtual void show() {
            cout << "Base class Matrix" << endl;
        }

        virtual Mat *getCvMat() = 0;

        static Matrix<T> *eye(int row, int col, MATRIX_TYPE type) {
            if (row <= 0 || col <= 0) {
                throw ex::InvalidSizeException("creating Matrix", 1, row, col);
            }
            Matrix<T> *mat;
            if (type) {
                mat = new BasicMatrix<T>(row, col);
                for (size_t i = 0; i < row; i++)
                    mat->setByIndex(i, i, 1);
            } else {
                mat = new SparseMatrix<T>(row, col);
                for (size_t i = 0; i < row; i++)
```

```
                    mat->setByIndex(i, i, 1);
            }
            return mat;
        }

    };
```

BasicMatrix:

BasicMatrix的prototype如下所示:

```
template<class T>
    class BasicMatrix : public Matrix<T> {
    private:
        T *m_data;

    public:
        BasicMatrix(int, int, T val = 0);

        BasicMatrix(int, int, T *);

        BasicMatrix(const Mat &mat);

        BasicMatrix(vector<vector<T>>);

        BasicMatrix(const BasicMatrix<T> &);

        BasicMatrix<T> &operator=(const BasicMatrix<T> &);

        ~BasicMatrix<T>();

        void add(const BasicMatrix<T> &);

        void subtract(const BasicMatrix<T> &);

        void scalarMultiply(T);

        void scalarDivide(T);

        void dotProduct(const BasicMatrix<T> &);

        void crossProduct(const BasicMatrix<T> &);

        void transpose();

        void inverse();

        void reverse();

        void conjugate();

        T getByIndex(int, int) const;

        void setByIndex(int, int, T);

        T getMax();
```

```
        T getMin();

        T getSum();

        T getAvg();

        T getRowMax(int);

        T getColMax(int);

        T getRowMin(int);

        T getColMin(int);

        T getRowSum(int);

        T getColSum(int);

        T getRowAvg(int);

        T getColAvg(int);

        void Gaussian_Eliminate(BasicMatrix<T> &ans, BasicMatrix<T>
&eigenmatirx);

        bool getEigenvalue(int LoopNumber, BasicMatrix<T> &result);

        void QR(BasicMatrix<T> &Q, BasicMatrix<T> &R);

        BasicMatrix<T> &getEigenvector(BasicMatrix<T> &eigenvector, const T
lamda);

        T getTrace();

        T getDeterminant();

        bool getEigen(BasicMatrix<T> &eigenvector, BasicMatrix<T> &eigenvalue,
double error,
                      double iterator);

        void reshape(int row, int col);

        void loop(T[]);

        void sliceRow(int row1, int row2);

        void sliceCol(int col1, int col2);

        void slice(int row1, int row2, int col1, int col2);

        BasicMatrix<T> *convolve(BasicMatrix<T> &, int stride = 1, int padding =
0);

        void exponent(int exp);

        void show();

        BasicMatrix<T> operator+(const BasicMatrix<T> &);
```

```
        BasicMatrix<T> operator-(const BasicMatrix<T> &);

        template<typename P>
        BasicMatrix<T> operator*(P);

        Mat *getCvMat();

        template<typename P>
        friend BasicMatrix<T> operator*(P val, BasicMatrix<T> &right) {
            return right * val;
        }

        T *getData() {
            return this->m_data;
        }
    };
```

BasicMatrix初始化时都需要默认调用父类的构造器（行列构造器或OpenCV构造器），并且为容器m_data指针数组申请足够的内存空间。BasicMatrix重载了析构函数，在析构函数中释放了m_data指针数组申请的内存空间。BasicMatrix重载了拷贝构造器，通过传入的BasicMatrix为当前的BasicMatrix申请相等大小的内存空间并一一对应赋值。BasicMatrix重载了拷贝赋值操作符，先判断当前对象是否和等号右边的BasicMatrix是否指向同样的一段内存空间（即为相同的对象），若相同直接返回*this，若不相同则依据拷贝构造器的方式先申请对应的内存空间再一一赋值，最后释放原来的m_data的内存空间并指向新申请的内存空间。

SparseMatrix:

SparseMatrix的prototype如下所示：

```
template<class T>
    class SparseMatrix : public Matrix<T> {
    private:
        map<int, Triple<T> *> tri_map;
    public:
        SparseMatrix(int, int);

        SparseMatrix(const cv::Mat &mat);

        SparseMatrix(vector<vector<T>>);

        SparseMatrix(int, int, T *);

        SparseMatrix(int, int, vector<Triple<T>>);

        SparseMatrix(int, int, map<int, Triple<T> *>);

        SparseMatrix(const SparseMatrix<T> &);

        SparseMatrix<T> &operator=(const SparseMatrix<T> &);

        ~SparseMatrix<T>();

        void add(const SparseMatrix<T> &);

        void subtract(const SparseMatrix<T> &);
```

```
        void scalarMultiply(T);

        void scalarDivide(T);

        void dotProduct(const SparseMatrix<T> &);

        void crossProduct(const SparseMatrix<T> &);

        bool getEigen(SparseMatrix<T> &eigenvector, SparseMatrix<T> &eigenvalue,
double error,
                      double iterator);

        void transpose();

        void inverse();

        void reverse();

        void conjugate();

        T getMax();

        T getMin();

        T getSum();

        T getAvg();

        T getRowMax(int);

        T getColMax(int);

        T getRowMin(int);

        T getColMin(int);

        T getRowSum(int);

        T getColSum(int);

        T getRowAvg(int);

        T getColAvg(int);

        T getEigenvalue(int LoopNumber, SparseMatrix<T> &result);

        void Gaussian_Eliminate(SparseMatrix<T> &ans, SparseMatrix<T>
&eigenmatirx);

        SparseMatrix<T> &getEigenvector(SparseMatrix<T> &eigenvector, const T
lamda);

        T getByIndex(int _row, int _col) const;

        void setByIndex(int _row, int _col, T val);

        T getTrace();
```

```
        T getDeterminant();

        void reshape(int row, int col);

        void sliceRow(int row1, int row2);

        void sliceCol(int col1, int col2);

        void slice(int row1, int row2, int col1, int col2);

        SparseMatrix<T> *convolve(SparseMatrix<T> &, int stride = 1, int padding
= 0);

        void exponent(int exp);

        map<int, Triple<T> *> getTriples() const {
            return this->tri_map;
        }

        void QR(SparseMatrix<T> &Q, SparseMatrix<T> &R);

        void show();

        cv::Mat *getCvMat();
    };
```

SparseMatrix初始化时都需要默认调用父类的构造器（行列构造器或OpenCV构造器），若传入的值在相应的行列处有对应的数据值，则使用初始化对应的Triple<T> *赋值给对应int类型的key值对应的value值。SparseMatrix重载了拷贝构造器，只要根据传入的SparseMatrix的对象的tri_map通过iterator循环得到Triple<T> *，并为一个新的Triple<T> *申请对应的内存空间并赋值给对应int类型的key值对应的value值，key值通过Triple中存储的行值和列值计算所得。SparseMatrix重载了拷贝赋值运算符，先判断当前对象是否和等号右边的SparseMatrix是否指向同样的一段内存空间（即为相同的对象），若相同直接返回*this，若不相同则依据拷贝构造器的方式据传入的SparseMatrix的对象的tri_map通过iterator循环得到Triple<T> *，并为一个新的Triple<T> *申请对应的内存空间并赋值给对应int类型的key值对应的value值。

2、OpenCV操作

BasicMatrix和cv::Mat的相互转化如下图所示:

```
template<class T>
BasicMatrix<T>::BasicMatrix(const Mat &mat): Matrix<T>(mat) {
    if (mat.channels() != 1)
        throw ex::InvalidChannelDepth(mat.channels());
    this->m_data = new T[this->getRow() * this->getCol()];
    for (size_t i = 0; i < this->getRow(); i++) {
        for (size_t j = 0; j < this->getCol(); j++) {
            setByIndex(i, j, (T) mat.at<uchar>(i, j));
        }
    }
}
```

```
template<class T>
Mat *BasicMatrix<T>::getCvMat() {
    Mat *mat = new Mat(this->getRow(), this->getCol(), CV_8UC1);
    for (size_t i = 0; i < this->getRow(); i++) {
        for (size_t j = 0; j < this->getCol(); j++) {
            double re = real(getByIndex(i, j));
            mat->at<uchar>(i, j) = re;
        }
    }
    return mat;
}
```

SparseMatrix和cv::Mat的相互转化如下所示：

```
template<class T>
SparseMatrix<T>::SparseMatrix(const cv::Mat &mat): Matrix<T>(mat) {
    if (mat.channels() != 1)
        throw ex::InvalidChannelDepth(mat.channels());
    for (size_t i = 0; i < this->getRow(); i++) {
        for (size_t j = 0; j < this->getCol(); j++) {
            if ((T) mat.at<uchar>(i, j) != 0)
                setByIndex(i, j, (T) mat.at<uchar>(i, j));
        }
    }
}
```

```
template<class T>
Mat *SparseMatrix<T>::getCvMat() {
    bool flags[this->getRow()][this->getCol()];
    memset(flags, false, sizeof(flags));
    Mat *mat = new Mat(this->getRow(), this->getCol(), CV_8UC1);
    for (auto it = tri_map.begin(); it != tri_map.end(); it++) {
        auto tri = it->second;
        double re = real(getByIndex(tri->_row, tri->_col));
        mat->at<uchar>(tri->_row, tri->_col) = re;
        flags[tri->_row][tri->_col];
    }
    for (size_t i = 0; i < this->getRow(); i++) {
        for (size_t j = 0; j < this->getCol(); j++) {
            if (!flags[i][j]) mat->at<uchar>(i, j) = 0;
        }
    }
    return mat;
}
```

在测试代码中我们使用OpenCV读取本地的图片将其转化为灰度图并作为构造函数的输入参数传入BasicMatrix的构造器中（SparseMatrix同理），使用getCvMat则可以从BasicMatrix或SparseMatrix中返回cv::Mat的指针类型变量， 如下所示：

```
void mat_opencv_test() {
    cout << "Basic matrix opencv test begin: " << endl;
    Mat img = imread("./img/btn1.png");
    Mat grey;
    cvtColor(img, grey, CV_BGR2GRAY);
    BASIC_MATRIX_INT bm1(grey);
    cout << "(1) ";
    bm1.show();
    BASIC_MATRIX_INT bm2(50, 50, 240);
    Mat *mat1 = bm2.getCvMat();
    imwrite("./img/test1.png", *mat1);
    Matrix<int> *bm3 = Matrix<int>::eye(30, 30, true);
    for (size_t i = 0; i < bm3->getRow(); i++)
        bm3->setByIndex(4, i, 200);
    Mat *mat2 = bm3->getCvMat();
    imwrite("./img/test2.png", *mat2);
    imwrite("./img/test3.png", *bm1.getCvMat());
    SparseMatrix<int> bm4(100, 100);
    Mat *mat3 = bm4.getCvMat();
    imwrite("./img/test4.png", *mat3);

    cout << "test end" << endl << endl;
}
```

最终我们可以使用imwrite方法将我们得到的cv::Mat导出为灰度图片存储。

3、矩阵操作和运算

矩阵同矩阵点积

条件：矩阵行列均相同

$$C = A \cdot B$$

$$c_{ij} = a_{ij} \cdot b_{ij}$$

向量同矩阵点积

条件：向量长度与矩阵行数相同

$$C = x \cdot A$$

$$c_{ij} = x_i \cdot a_{ij}$$

矩阵同矩阵叉积

条件：$r_A = c_B$

本质为矩阵乘法，矩阵乘法公式如下：

$C = A \times B$

$$c_{ij} = \sum a_{ik} \times b_{kj}$$

矩阵同矩阵卷积

假定 A 为被卷积矩阵，K 为卷积核，B 为卷积结果。

条件：K 矩阵为正方形

公式：

$$B(i,j) = \sum_{m=0} \sum_{n=0} K(m,n) \times A(i-m, i-n)$$

代码中使用四层循环实现：

```
for (int i = 0; i < r; i++) {
    for (int j = 0; j < c; j++) {
        for (int k = 0; k < rev.row; k++) {
            for (int t = 0; t < rev.col; t++) {
                mat.setByIndex(i, j, mat.getByIndex(i, j) + rev.getByIndex(k, t)
* ext.getByIndex(k+i, t+j));
            }
        }
    }
}
```

最大值最小值、和与平均值

1、初始化最小值为矩阵的第一个元素，遍历矩阵的所有元素，当遇到更小的数值的时候进行交换，最大值同理。需要注意的是，我们的稀疏矩阵采用map进行储存，在map内的元素都是非零的，因此我们要将最大值或者最小值初始为0。

2、我们在计算和的时候只需将矩阵内所有的元素进行累加，将和除以矩阵的大小即可得到平均值。同样的，我们可以通过类似的方法求出每行或者每列的和与平均值，与求总体的和与平均值不同的则是要在参数当中加上对应的行数或者列数。

特征值与特征向量

1、通过QR分解法计算一般矩阵的特征值，基本的思路是不断的迭代，迭代格式为A（k）=Q（k）R（k），A（k+1）=R（k）Q（k）。k为循环的次数，随着k的增大，Ak会越来越近似与A（k+1），并且Ak会逐渐收敛于上三角矩阵，因此对角线上的元素就是Ak上的元素。在QR分解的过程中，始终保持如下关系——A（k+1）=QTAkQ，Ak始终与A相似，因此他们的特征值相同，综上，我们只需要将目标矩阵进行多次QR分解就可以得到上三角矩阵Ak，取Ak的对角线元素即可得到矩阵AK也就是矩阵A的特征值。

2、高斯消元处理得到特征向量。从第一列开始，找出每列的最大数值及其行数，将最大的值所在的一行与第一行交换，接着开始按列消元，这样的话这一列下方的所有元素都可以消为0.最后从上三角矩阵的最底端开始逐步向上带入求解，换句话说，就是将矩阵转化为对角矩阵。这样求解特征向量x的过程就变得十分简单。

3、对于非对称矩阵，QR分解虽然能得到所有的特征值，但是不能通过这些特征值得到全部正确的特征向量，而实对称矩阵是线性代数的重要研究对象，我们可以通过雅克比迭代法求得实对称矩阵的特征值和特征向量。基本思路为

(1) 初始化特征向量为对角阵V。

(2) 在非对角线元素中找到绝对值最大元素S_{ij}。

(3) 通过公式计算旋转角度。并得到旋转矩阵P。

$$\tan(2\theta) = \frac{2S_{ij}}{S_{jj} - S_{ii}}$$

（4）用以下公式计算S'，用当前特征向量矩阵V乘以矩阵P得到当前的特征向量V。

$$\begin{aligned}
S'_{ii} &= c^2\, S_{ii} - 2\, sc\, S_{ij} + s^2\, S_{jj} \\
S'_{jj} &= s^2\, S_{ii} + 2sc\, S_{ij} + c^2\, S_{jj} \\
S'_{ij} &= S'_{ji} = (c^2 - s^2)\, S_{ij} + sc\, (S_{ii} - S_{jj}) \\
S'_{ik} &= S'_{ki} = c\, S_{ik} - s\, S_{jk} && k \neq i, j \\
S'_{jk} &= S'_{kj} = s\, S_{ik} + c\, S_{jk} && k \neq i, j \\
S'_{kl} &= S_{kl} && k, l \neq i, j
\end{aligned}$$

where $s = \sin(\theta)$ and $c = \cos(\theta)$.

（5）判断此时非主对角线元素中最大值是否小于给定的允差范围，如果是，停止计算，返回特征向量V，如果不是，继续计算。

矩阵行列式计算

采用递归的方式计算矩阵的行列式，递归的截止条件是矩阵的尺寸为2*2.（一阶矩阵和二阶矩阵可以用简单的数学表达式计算）从第一行开始展开，在递归的过程中，首先为代数余子式申请内存，接着遍历每一个元素，为代数余子式赋值，特别需要注意的是，如果该元素为零，则代数余子式也一定为零，因此不需要计算。

矩阵的求逆

矩阵求逆的逻辑比较清晰，首先通过递归的方法，从第一行开始展开。为每一个元素的代数余子式进行赋值，如果该元素为零，我们选择直接跳过，因为在稀疏矩阵当中，有大量含有零的元素，这种情况下代数余子式一定为零，可以很大程度上减少我们递归的次数。

4、问题与解决方法

在测试过程中，我们发现`std::complex<int>`无法和`std::complex<int>`类型进行大小比较。编译报错，错误出现在
`getMax()，getMin()`等方法中，在一系列尝试后，我们发现可以通过重载运算符来解决。

```
    template<typename P>
    bool operator<(complex<P> c1, complex<P> c2) {
        return c1.imag() * c1.imag() + c1.real() * c1.real() < c2.imag() *
c2.imag() + c2.real() * c2.real();
    }

    template<typename P>
    bool operator>(complex<P> c1, complex<P> c2) {
        return c2 < c1 || c2 == c1;
    }
```

5、小组分工

邱逸伦：项目架构、reshape和slice、矩阵库和OpenCV相互转化、异常处理、代码测试

张钧翔：基础的算术约简运算、特征值和特征向量的计算、行列式和迹的计算、异常处理、代码测试

李开：基础的矩阵运算操作、矩阵卷积操作、异常处理、代码测试框架

项目GitHub链接：https://github.com/Olin66/2022-Spring-C-and-Cpp-Project-Matrix