

Matrix Class

December 18, 2022

目录

1 功能展示	2
2 分析与实现	3
2.1 一个草案	3
2.1.1 “地基”	3
2.1.2 共享内存	4
2.1.3 ROI 支持	5
2.1.4 多通道支持	6
2.2 常用操作	6
2.2.1 矩阵初始化	6
2.2.2 元素访问	7
2.2.3 其它运算	8
2.2.4 矩阵乘法	9
2.3 抽象	10
2.3.1 开始之前	10
2.3.2 基类与实现	11
2.3.3 channel 提取	12
3 源码	13
4 总结	13

1 功能展示

1. 基本功能

```
// 定义并初始化一个 2x2 的矩阵
Mat<double> A({{3., 4.},
              {8., 7.}});

// 再声明一个
Mat<double> B({{7., 2.},
              {4., 0.}});

B(1, 1) = 9.; // 访问/修改元素

// 乘一下
Mat<double> C = A * B;
cout << C << endl;

// 输出最小值
cout << C.min() << endl;
```

输出：

```
[37, 42,
 84, 79]

37
```

2. 使用共享内存避免硬拷贝

```
Mat<double> A(2, 2);

Mat<double> B = A; // no hard copy

// 当然，对 B 的修改也会同步到 A
B(1, 0) = 2.;
cout << A << endl;

auto T = A.transpose(); // 转置，还是没有 hard copy
T(1, 0) = 3.;
cout << A << endl;
```

```
[0, 0,
 2, 0]

[0, 3,
 2, 0]
```

3. ^{channel}多通道支持

```
// 一个 2x2 的 3 channels 矩阵
Mat<double, 3> A(2, 2);

// 初始化第 0, 1 个 channel
A.channel(0).set({{3., 4.},
                 {8., 7.}});
A.channel(1).set({{7., 2.},
                 {4., 9.}});
```

```
// 把前两个 channel 的乘积放到第三个 channel
A.channel(2) = A.channel(0) * A.channel(1);

cout << A << endl;
```

输出：

```
[(3, 7, 37), (4, 2, 42),
 (8, 4, 84), (7, 9, 79)]
```

4. 高效 ROI 支持

支持选择矩阵中的 ROI，并且没有内存硬拷贝。

```
Mat<double> A(3, 3);

// 从左上角 (1, 1) 开始，大小为 (2, 2) 的 ROI
Mat<double> roi = A.block(Rect({1, 1}, {2, 2}));

cout << roi << endl;

// 对 ROI 做的修改会同步到原矩阵
roi(1, 0) = 5.;
cout << A << endl;
```

输出：

```
[0, 0,
 0, 0]

[0, 0, 0,
 0, 0, 0,
 0, 5, 0]
```

5. 乘法加速支持

```
Mat<double> A(3, 3);

// 从左上角 (1, 1) 开始，大小为 (2, 2) 的 ROI
Mat<double> roi = A.block(Rect({1, 1}, {2, 2}));

Mat<double> B = A * A; // 此处会调用 openblas 加速矩阵乘法
Mat<double> C = roi * roi; // ROI 也可以
```

2 分析与实现

2.1 一个草案

2.1.1 “地基”

首先，一个很显然的想法是，把矩阵定义成这样：

```
template<typename T>
class Mat {
    size_t rows_, cols_;
    T* p_data_;
};
```

这样做，算是“能用”级别，但是不算特别好。

2.1.2 共享内存

但是题目要求，矩阵赋值的时候不能用硬拷贝，所以我们需要一个共享内存。

这里就直接用 C++ 标准库提供的 `shared_ptr` 来管理矩阵申请的内存。`shared_ptr` 会对使用这块内存的 `shared_ptr` 实例进行计数。如果有复制，它就会把计数加一，如果有对象销毁了，就减一。当它发现计数到 0 的时候，它就会把这块内存给释放掉，就不会产生内存泄漏和 double free。

```
template<typename T>
class Mat {
    size_t rows_, cols_;
    T* p_data_;

    std::shared_ptr<T> data_holder_;
};
```

至于不选择手动管理内存的原因，是因为在 C++ 的新标准下，除了 copy constructor 和 copy assignment，还有 move constructor 和 move assignment，然后处理这些东西很头大，稍微不注意就有可能出错，所以就直接用了标准库的 `shared_ptr` 来管理内存。

当然，这里还有一个点，就是虽然我们使用了 `shared_ptr`，但是我们只用它来管理内存申请和释放，实际上访问数据的时候，我们还是使用 `p_data_` 指针。首先这样是安全的，因为 `p_data_` 就是指向 `shared_ptr` 拥有的那块内存，只要 `shared_ptr` 还在，`p_data_` 就不会失效。然后至于为什么要这样多此一举，我只能说，「这是等一下会用到的神奇妙妙工具」。

然后对 copy constructor 的实现也很简单：

```
class Mat {
    // ...

    Mat(const Mat&) = default;
};
```

对，让它维持默认即可，C++ 默认会帮我们每个成员变量都拷贝一份，这正是我们需要的。

但是这里不对 copy assignment 实现内存软拷贝，因为当 $A = B$ 的时候，用户可能确实需要把 B 中的内容拷贝到 A 去（特别是 ROI）。如果我们只是把 B 的指针拷到 A 去，用户可能会迷惑为什么 A 变了，但是 A 对应的原矩阵没变（例如 A 是某个矩阵的 ROI 的时候）。所以这里就只对 copy constructor 做软拷贝，对 copy assignment 还是做硬拷贝。

shared_ptr 的一些细节 因为我们申请和释放一个数组的时候，是使用 `p = new T[cnt]` 和 `delete []p` 的。

但是在 `shared_ptr` 中，如果我们定义 `shared_ptr<T>` 的话，`shared_ptr` 释放内存的时候有可能调用的是 `delete p`，虽然内存也能释放，但是 `T` 的构造函数就不一定都会被调用。（虽然在矩阵中 `T` 都是简单类型，但是看着就很不爽。）

要解决这个问题，我们一是可以使用 C++17 标准中加入的 `shared_ptr<T[]>`，它就可以正确地释放内存。另一个途径是我们可以将 `shared_ptr` 的构造函数中，传入自定义的释放内存的函数，即 `shared_ptr<T>(new T[cnt], [](T *p) { delete []p; })`，这样也可以。因为我电脑上的 g++ 对 C++17 支持貌似不是完全，所以这里就使用了第二个方法。

eval 如果用户确实需要把某个矩阵硬拷贝一份，我们也提供了一个叫做 `eval` 的函数：

```
class Mat {
    // ...

    Mat eval() const {
```

```

        Mat mat(size());
        return mat = *this;
    }
};

```

就是先创建一个大小一模一样的矩阵，然后再把元素硬拷贝过去。

至于为什么选择 `eval` 这个名字，因为后面也可以对一些中间代理类，“计算”出一份新的矩阵。

2.1.3 ROI 支持

在开始之前，我们先回顾一下，如果只用 `rows_` 和 `cols_`，那么元素 (r, c) 在一维数组的下标就是 $r * \text{cols_} + c$ 。

在一般情形，这样子是没什么问题的。但是我们不妨把路走宽一点，为什么 `cols_` 就一定是矩阵中行与行的“间隔”呢？

换句话说，我们可以加一个 `steps_`，表示矩阵中行与行的“间隔”，然后原来的 `cols_` 就只用来表示矩阵的大小。

举个例子，对于一个 `rows_ = 3`, `cols_ = 3`, `steps_ = 5` 的矩阵：

	0	1	2		
0					
1					
2					

其中橙色部分还是矩阵的有效范围，但是灰色部分不属于矩阵的有效数据（可以看作是 padding 等）。

这样的话，我们就可以很方便地定义一个矩阵的 ROI。对于一个 5×5 的矩阵，如果我们想要取出橙色部分的 ROI，我们可以把 `p_data_` 指向橙色部分的左上角（即第一个元素），然后把长宽（`rows_` 和 `cols_`）设为 2 和 2，`steps_` 就跟随原矩阵即可。

	0	1	2	3	4
0					
1					
2					
3					
4					

这样我们就可以不使用 memory hard copy 来取出一个矩阵的 ROI。

当然为了管理内存，`data_holder_` 还是要从原矩阵中拷贝一份过来，这就是上文中把 `data_holder_` 和 `p_data_` 分开的原因（`p_data_` 可以指向这块内存中的任何位置，但是 `data_holder_` 不能动）。

写成代码的话，大概就是（这里在构造函数中实现了）：

```

class Mat {
    // ...

```

```

    Mat(Mat &mat, Rect roi)
        : rows_(roi.rows), cols_(roi.cols), steps_(mat.steps_), data_holder_(mat.data_holder_)
    {
        if (roi.r + roi.rows > mat.rows_ || roi.c + roi.cols > mat.cols_)
            throw std::out_of_range("ROI region out of range");
        p_data_ = &mat(roi.r, roi.c, 0);
    }
};

```

2.1.4 多通道支持

只是支持多通道储存的话，其实挺简单的。把原来每一个 cell 存储 1 个数改成每个 cell 存储“通道”个数即可。

但是相比与单通道的矩阵，多通道矩阵的处理会比较地麻烦，因为从单通道矩阵变成多通道矩阵，本质上就是从二维矩阵变成了三维矩阵。然后访问接口和矩阵乘法的实现都要进行修改，想要一组代码同时支持两种矩阵，会比较地麻烦。

然后为了一定程度上方便一点，这里把通道数定义成了模板参数。虽然这样一来，通道数就不太能动态变化了（虽然也可以预定义几个常用的通道），但考虑到 opencv 也只是提供了有限种 channel 可供选择¹，也尚且可以接受。

所以现在的矩阵定义就是

```

template<typename T, size_t CHANNELS>
class Mat {
    size_t rows_, cols_;
    size_t steps_;
    T* p_data_;

    std::shared_ptr<T> data_holder_;
};

```

当访问某个元素的时候，就用 $(r * steps_ + c) * CHANNELS + channel$ 计算其下标，并且把 channel 添加默认参数 0，以兼容单通道矩阵。

2.2 常用操作

2.2.1 矩阵初始化

对于矩阵初始化，常用的方法要不就是声明了之后手动一个个赋值，要不就是把一个数组传进去让它复制。

我就想着，在 C++ 里面，二维数组的初始化可以用 $\{\{1, 2\}, \{3, 4\}\}$ 这种初始化列表，那么这种初始化方式能不能用到矩阵里面呢？

答案是可以的，在 C++ 新标准里面，有一个叫做 `std::initializer_list<T>` 的东西（其中 T 表示里面元素的类型），它可以接收像上面那样大括号的数据。然后上面的初始化列表是二维的，我们可以把 `initializer_list` 稍微地嵌套一下，即把 `initializer_list<initializer_list<T>>` 拿来用。

```

class Mat {
    // ...

    explicit Mat(initializer_list<initializer_list<T>> list) {
        static_assert(CHANNELS == 1);

        rows_ = list.size();
        cols_ = list.begin()->size();
        data_holder_ = std::shared_ptr<T>(new T[rows_ * cols_], [](T *p) { delete []p; });
    }
};

```

¹<https://github.com/opencv/opencv/blob/4.x/modules/core/include/opencv2/core/hal/interface.h#L69>

```

        p_data_ = data_holder_.get();

        size_t p = 0;
        for (auto row : list) {
            assert(row.size() == cols); // 确保每一行都是等长的
            for (T val : row)
                buffer[p++] = val;
        }
    }
};

```

虽然能用是能用了，但是还是感觉很不优雅。特别是 for 循环中要判断每一个 `initializer_list` 是否等长。这个检查是运行期的，但是接收的参数往往是编译器决定的（就像一开始的功能展示那样，是已经写好的），我们能不能在编译器就限制每一行都是等长的呢？

这里有一个好消息和一个坏消息。坏消息就是 `initializer_list` 不支持编译器决定 `size`，更不要说编译器决定其内容的 `size` 是否正确了。

但是好消息是，我们可以不用 `initializer_list` 来达成这个目的：

```

template<size_t M, size_t N>
class Mat {
    // ...

    explicit Mat(const T(&list)[M][N]) : Mat(M, N) {
        static_assert(CHANNELS == 1);

        for (size_t r = 0; r < rows(); r++)
            for (size_t c = 0; c < cols(); c++)
                operator()(r, c) = list[r][c];
    }
};

```

如果传入不等长的初始化列表，编译器就会报错。而且这个代码也比上面的要短，还是蛮优雅的。

2.2.2 元素访问

一般来说，用下标访问元素，一般是要重载 `[]` 运算符的。但是矩阵是二维的，如果用 `[r][c]` 访问，就要另外加一个中间层，首先一个担心就是会影响性能，另外这样做也非常不优雅。然后如果用 `[r, c]` 来访问，emmm，只能说想法很美好，但是 `[]` 运算符接收多个参数只在 C++23 之后被支持，所以也不太能用。然后如果用 `[{r, c}]` 呢？中间是一个 `initializer_list`，貌似也不是不行，但是稍微有一点点丑。

综合利弊以及参考了一些矩阵库的设计之后，这里决定用 `(r, c)`（即 `operator()`）来实现元素访问。

同时，参考标准库，对于常用的下标访问 `(r, c)` 不检查参数合法性，以保证性能。如果需要检查参数合法性，则增加一个 `at` 函数作为带有参数检查的 `(r, c)`。

代码就是：

```

class Mat {
    // ...

    T operator()(size_t r, size_t c, size_t channel = 0) const {
        return p_data_[(r * steps_ + c) * CHANNELS + channel];
    }

    T& operator()(size_t r, size_t c, size_t channel = 0) {
        return p_data_[(r * steps_ + c) * CHANNELS + channel];
    }
}

```

```

    T at(size_t r, size_t c, size_t channel = 0) const {
        if (r >= rows_ || c >= cols_ || channel >= CHANNELS)
            throw std::out_of_range("index out of range");
        return p_data_[(r * steps_ + c) * CHANNELS + channel];
    }

    T& at(size_t r, size_t c, size_t channel = 0) {
        if (r >= rows_ || c >= cols_ || channel >= CHANNELS)
            throw std::out_of_range("index out of range");
        return p_data_[(r * steps_ + c) * CHANNELS + channel];
    }
};

```

使用的时候，就是：

```

Mat<double> A(2, 2);
A(0, 0) = 2.;
A(1, 1) = A(0, 0);

```

虽然没 `[r, c]` 那么直观，但是还是很不错的。

2.2.3 其它运算

对于矩阵的运算，一般可以分为四种类型：

类型	定义	解释	举例
Piecewise	$A, B \rightarrow A$	两个矩阵对应位置的元素做二元运算	加减、硬赋值
Self Piecewise	$A \rightarrow A$	一个矩阵的每个元素都做一元运算	取反、取绝对值
Scalar	$A, x \rightarrow A$	一个矩阵的每个元素和另一个标量做二元运算	加减乘除一个标量
Reduce	$A \rightarrow y$	一个矩阵的每个元素 reduce 成一个标量	取最大值、最小值

对于每种类型，除了具体的二元或一元函数不确定之外，主要逻辑都是相同的。所以我们可以把他们都抽象出来，接受对应的参数，以及一个二元或一元函数，然后做运算。

具体代码如下：

```

template<typename T, size_t CHANNELS, typename BinaryOp>
void call_piecewise_op(Mat<T, CHANNELS> &lhs, const Mat<T, CHANNELS> &rhs, BinaryOp op) {
    if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols())
        throw std::invalid_argument("the matrices should have the same size");

    // 由于内存可能不连续，所以不能直接遍历 p_data_
    for (size_t r = 0; r < lhs.rows(); r++)
        for (size_t c = 0; c < lhs.cols(); c++)
            for (size_t l = 0; l < CHANNELS; l++)
                lhs(r, c, l) = op(lhs(r, c, l), rhs(r, c, l));
}

template<typename T, size_t CHANNELS, typename UnaryOp>
void call_self_piecewise_op(Mat<T, CHANNELS> &mat, UnaryOp op) {
    for (size_t r = 0; r < mat.rows(); r++)
        for (size_t c = 0; c < mat.cols(); c++)
            for (size_t l = 0; l < CHANNELS; l++)
                mat(r, c, l) = op(mat(r, c, l));
}

template<typename T, size_t CHANNELS, typename BinaryOp>
void call_scalar_op(Mat<T, CHANNELS> &lhs, T rhs, BinaryOp op) {
    for (size_t r = 0; r < lhs.rows(); r++)
        for (size_t c = 0; c < lhs.cols(); c++)

```



```

        for (size_t l = 0; l < CHANNELS; l++)
            lhs(r, c, l) = op(lhs(r, c, l), rhs);
    }

template<typename T, typename U, size_t CHANNELS, typename BinaryOp>
U call_reduce(const Mat<T, CHANNELS> &mat, U initial, BinaryOp op) {
    U result = initial;
    for (size_t r = 0; r < mat.rows(); r++)
        for (size_t c = 0; c < mat.cols(); c++)
            for (size_t l = 0; l < CHANNELS; l++)
                result = op(result, mat(r, c, l));
    return result;
}

```

然后对于一些常用运算，我们就只需要用一两行调用这些函数即可：

```

class Mat {
    // ...

    Mat& operator+=(const Mat &rhs) {
        call_pieewise_op(*this, rhs, std::plus<T>());
        return *this;
    }

    Mat& operator+=(const T rhs) {
        call_scalar_op(*this, rhs, std::plus<T>());
        return *this;
    }

    Mat& operator-=(const Mat &rhs) {
        call_pieewise_op(*this, rhs, std::minus<T>());
        return *this;
    }

    Mat& operator-=(const T rhs) {
        call_scalar_op(*this, rhs, std::minus<T>());
        return *this;
    }

    Mat abs() const {
        Mat result = eval(); // 用于复制一个新矩阵 (with hard copy)
        call_self_pieewise_op(result, [](T a) { return a < 0 ? -a : a; });
        return result;
    }

    inline T min() const {
        return call_reduce(*this, std::numeric_limits<T>::max(), [](T a, T b) { return std::min(a, b); });
    }
};

```

2.2.4 矩阵乘法

对于矩阵乘法，因为不同于上述的 4 种模型，并且对于部分类型，我们也可以进行优化，所以就单独开一小节。

```

template<typename T>
void call_openblas_gemm(size_t M, size_t N, size_t K, T alpha, T *A, size_t lda, T *B, size_t ldb,
                       T beta, T *C, size_t ldc) {

```

```

// 根据不同类型调用不同的函数
if constexpr (std::is_same_v<T, double>)
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                M, N, K, alpha, A, lda, B, ldb, beta, C, ldc);
else if constexpr (std::is_same_v<T, float>)
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                M, N, K, alpha, A, lda, B, ldb, beta, C, ldc);
else
    static_assert(always_false_v<T>, "only double and float are supported");
}

template<typename T, size_t CHANNELS>
Mat<T, CHANNELS> operator*(const Mat<T, CHANNELS> &lhs, const Mat<T, CHANNELS> &rhs) {
    // 检查参数
    if (lhs.cols() != rhs.rows())
        throw std::invalid_argument("matrices size mismatched in multiplication");
    Mat<T, CHANNELS> result(lhs.rows(), rhs.cols());

    // 对于 double 和 float, 我们可以用 OpenBLAS 加速
    if constexpr (std::is_same_v<T, double> || std::is_same_v<T, float>) {
        if constexpr (CHANNELS == 1) {
            call_openblas_gemm<T>(lhs.rows(), lhs.cols(), rhs.cols(), 1., lhs.data(), lhs.steps(),
                                   rhs.data(), rhs.steps(), 0., result.data(), result.steps());
        } else {
            // 对于多通道, 因为同一通道的元素不连续, 而且 OpenBLAS 也没有原生支持多通道的矩阵乘法,
            // 所以这里我们先把每个通道单独提取出来, 然后做乘法, 再放回去
            Mat<T> A(lhs.size()), B(rhs.size()), C(result.size());
            for (size_t l = 0; l < CHANNELS; l++) {
                A = lhs.channel_const(l);
                B = rhs.channel_const(l);
                call_openblas_gemm<T>(A.rows(), A.cols(), B.cols(), 1., A.data(), A.steps(),
                                       B.data(), B.steps(), 0., C.data(), C.steps());
                result.channel(l) = C;
            }
        }
    } else {
        // 否则就朴素矩阵乘法
        for (size_t l = 0; l < CHANNELS; l++)
            for (size_t r = 0; r < lhs.rows(); r++)
                for (size_t k = 0; k < lhs.cols(); k++)
                    for (size_t c = 0; c < rhs.cols(); c++)
                        result(r, c, l) += lhs(r, k, l) * rhs(k, c, l);
    }

    return result;
}

```

2.3 抽象

2.3.1 开始之前

因为一些需要, 比如说, 用户可能需要把矩阵转置之后做一些操作, 然后再转置回去。如果我们转置的时候不使用 hard copy, 我们就需要一层中间层 `MatTransposeProxy` 来给原矩阵“套层壳”, 操作的时候还是操作转置的矩阵, 但实际上修改的还是原来的矩阵。(如果不套壳的话, 各种情况的处理会让 `Mat` 的逻辑越来越复杂, 也是不可取的。)

这样听起来挺正常, 但是, 当实际操作起来, 你会发现一个很严重的问题: 对 `MatTransposeProxy` 这个类, 我们又需要重新写一遍加减乘除、最大值、绝对值的那一堆函数, 如果有很多个这样的“代理类”的话, 每

一个类都要写一遍。而且更要命的是，如果要支持不同的类交叉加减乘除的话，就要写 n^2 级别的函数。这样一来，可维护性就飞速下降，我们需要一些更好的办法来解决这个问题。

2.3.2 基类与实现

在实际开始之前，我们先来思考一下，什么东西是一个“矩阵”。在面向对象编程中，有一句著名的话，「如果它看起来像鸭子、游泳像鸭子、叫声像鸭子，那么它就是只鸭子」。所以思考一个矩阵有什么特征，是非常重要的。

矩阵最基本的特征，就是行数、列数，以及数据。对于行数和列数，就用数字来代表就可以；对于数据，我们可以这么定义，给出一个坐标 (r, c) ，可以得到矩阵这个位置的数据 $A_{r,c}$ 。

对应到代码中，行数和列数就是两个参数，数据就是 `operator()` 下标访问。那么就是说，实现了这三个东西的类，就是矩阵。

所以我们可以写出基类的大致框架：

```
template<typename T, size_t CHANNELS>
class MatBase {
    size_t rows() const;
    size_t cols() const;

    T operator()(size_t r, size_t c, size_t channel = 0);
    T& operator()(size_t r, size_t c, size_t channel = 0);
};
```

那么所有子类都只需要实现这几个函数就行了。

但是这里又有个问题，如果子类要 override 这些函数，而且又需要多态，这就需要把这些函数都声明成 `virtual`。但是 `virtual` 会稍微增大类的大小，而且调用的时候查函数表也会稍微消耗一点性能。

为了解决这个问题，我们可以改成静态 dispatch。我们可以给 `MatBase` 加一个模板参数 `Derived`，表示最终的字类是什么。

这样的话，我们就可以用 `static_cast` 把 `this` 转成子类，然后再调用其 `rows()` 等函数。因为 `static_cast` 是编译期完成的，并且之后的调用，因为没有 `virtual`，所以也是编译时完成绑定的。所以整个过程是编译期完成的，不会影响运行时性能。

所以代码就是

```
template<typename T, size_t CHANNELS, typename Derived>
class MatBase {
public:
    // 用于直接获取子类对象（编译期）
    inline const Derived& derived() const { return *static_cast<const Derived*>(this); }
    inline Derived& derived() { return *static_cast<Derived*>(this); }

    inline size_t rows() const { return derived().rows(); }
    inline size_t cols() const { return derived().cols(); }

    inline T operator()(size_t r, size_t c, size_t channel = 0) const {
        return derived()(r, c, channel);
    }

    inline T& operator()(size_t r, size_t c, size_t channel = 0) {
        return derived()(r, c, channel);
    }
};
```

然后像什么加减乘除、绝对值、最大值这些函数，都可以在这个类中实现。

```

template<typename T, size_t CHANNELS, typename Derived>
class MatBase {
    // ...

    template<typename OtherDerived>
    Derived& operator+=(const MatBase<T, CHANNELS, OtherDerived> &rhs) {
        call_pieewise_op(*this, rhs, std::plus<T>()); // 前面的 call_pieewise_op 也要改一下签名
        return derived();
    }

    Derived& operator+=(const T rhs) {
        call_scalar_op(*this, rhs, std::plus<T>());
        return derived();
    }

    template<typename U>
    Mat<T, CHANNELS> operator+(const U &rhs) const {
        return eval() += rhs;
    }

    // ...
};

```

然后令 Mat、MatTransposeProxy 都继承这个类，就可以继承这些函数，就不用我们写那么多遍了。

例如：

```

template<typename T, size_t CHANNELS>
class Mat : public MatBase<T, CHANNELS, Mat<T, CHANNELS>> {
    inline size_t rows() const { return rows_; }
    inline size_t cols() const { return cols_; }

    T operator()(size_t r, size_t c, size_t channel = 0) const {
        return p_data_[(r * steps_ + c) * CHANNELS + channel];
    }

    T& operator()(size_t r, size_t c, size_t channel = 0) {
        return p_data_[(r * steps_ + c) * CHANNELS + channel];
    }
}

```

2.3.3 channel 提取

有时候我们需要对矩阵的某一个 channel 取出来，当成一个单通道矩阵来做一点运算。然后为了避免内存硬拷贝，这里就用一个代理类来解决。

```

template<typename T, size_t CHANNELS>
class Mat : public MatBase<T, CHANNELS, Mat<T, CHANNELS>> {
    // ...

    MatChannelProxy<T, CHANNELS> channel(const size_t channel) {
        if (channel >= CHANNELS)
            throw std::out_of_range("channel out of range");
        return MatChannelProxy(*this, channel);
    }
};

template<typename T, size_t CHANNELS>

```

```

class MatChannelProxy : public MatBase<T, 1, MatChannelProxy<T, CHANNELS>> {
//                                     ^ 单个通道当然是单通道矩阵
protected:
    Mat<T, CHANNELS> mat_;
    size_t channel_;

public:
    MatChannelProxyConst(Mat<T, CHANNELS> mat, size_t channel) : mat_(mat), channel_(channel) {}

    inline size_t rows() const { return mat_.rows(); }
    inline size_t cols() const { return mat_.cols(); }
    inline Size size() const { return mat_.size(); }

    template<typename OtherDerived>
    MatChannelProxy& operator=(const MatBase<T, 1, OtherDerived> &rhs) {
        // binary_assignment: (a, b) => b
        call_pieewise_op(*this, rhs, binary_assignment<T>());
        return *this;
    }

    T operator()(size_t r, size_t c, size_t channel = 0) const {
        return mat_(r, c, channel_);
    }

    T& operator()(size_t r, size_t c, size_t channel = 0) {
        return mat_(r, c, channel_);
    }
};

```

然后得益于 MatBase, MatChannelProxy 瞬间就继承了加减乘除那一堆函数, 不用另外写一遍了, 十分方便。

3 源码

在 C++ 中, 如果把模板的声明和实现分别放在 .h 和 .cpp 中, 编译器在链接的时候就会往往会出现 can not find symbol 的错误。要解决这个错误, 要么在 .cpp 中用到的模板参数进行显式实例化 (即 `template class Mat<double>;`), 但是这样很不优雅, 也限制了拓展性。

所以在本次 Project 中, 就把全部声明和实现都放在了头文件中, 并且把后缀名改成 .hpp 提示头文件中包含实现。

然后按照传统, 加上 -Wall -Wextra 编译 (除此之外, 实际上还加了很多), 本代码不会产生任何编译警告。

4 总结

虽然这次 Project 开始的时候, 也没想着会如此大量地使用模板。但是写着写着, 就陷入了“写重复代码”——“用模板优化”——“好炫酷, 继续写新功能”的循环中。

而且写模板的时候真的感觉自己在操纵魔法, 当它编译错误的时候, 看编译错误往往看不出什么东西 (特别是它找不到 overload 函数的时候), 基本只能靠肉眼看。然后有时候整一些非常炫酷的操作的时候, 本来以为编译器多多少少会迷惑一下, 结果一编译, 通过了……。心理过程就像是: 「*It doesn't work....why? It works.....but why?*」。

不过话说回来, 还是挺佩服那帮写编译器的同志。C++ 标准一年又一年 (虽然准确来说是 3 年) 地提出新的 feature, 然后写编译器的就要在兼顾旧屎山标准的情况下, 支持新的标准。然后 (我) 看到新的炫酷东西被支持并且能编译成功跑起来的时候, 我真的会感叹, 这编译器真的牛逼。

然后继续话说回来, 这次 Project 虽然尝试了不少新的东西, 也写出了矩阵的一些操作, 并且我也自认为

这个库的可维护性还是挺不错的。但是受时间限制，还有很多功能没有实现，例如转置后的矩阵（代理类），也可以调用 OpenBLAS 进行矩阵乘法；也可以实现其它矩阵操作等等。

虽然，但是看着 eigen 矩阵库²，它可以同时支持编译期确定的矩阵大小（模板参数），也有支持运行时确定矩阵大小；而且它的不少 API 看着就非常地“人体工程学”，看着就很舒服；再者，对于一些运算，例如文档中写到的 `c.noalias() -= 2 * a.adjoint() * b;`，是“fully optimized and trigger a single gemm-like function call”。然后在看看自己的玩具，确实就只是一个玩具。虽然这次 Project 是最后一个了，但是对于如何设计出一个对用户足够友善的 API，以及如何写出高优化、高可维护性的代码，我确实还要继续学习。

然后按照传统（虽然 Project 4 忘记写了，但是也是放了），这个项目也会放到 GitHub 开源，会放在 https://github.com/YanWQ-momad/SUSTech_CS205_Projects 仓库的 Project5 的子目录下。欢迎大家 star，谢谢！

EOF.

²<https://eigen.tuxfamily.org/dox/index.html>