矩阵类

在Eigen 中，所有矩阵和向量都是Matrix模板类的对象。向量只是矩阵的一种特殊情况，具有 1 行或 1 列。

注：系数：就是矩阵中的值。

Matrix 的前三个模板参数

该矩阵类需要六个模板参数，但现在了解前三个参数已经足够，因为剩下的三个参数有默认值，现在我们将保持不变，我们将在下面讨论。

Matrix的三个强制性模板参数是：

```
Matrix<typename Scalar, int RowsAtCompileTime, int ColsAtCompileTime>
```

- `Scalar`是标量类型，即系数的类型。也就是说，如果您想要一个浮点矩阵，请选择`float`此处。有关所有支持的标量类型的列表以及如何扩展对新类型的支持，请参阅标量类型。
- `RowsAtCompileTime` 和 `ColsAtCompileTime`是在编译时已知的矩阵的行数和列数（如果在编译时不知道该数字，请参见下文了解该怎么做）。

我们提供了很多方便的 typedef 来涵盖通常的情况。例如，`Matrix4f`是一个 4x4 的浮点矩阵。这是Eigen定义的方式：

```
typedef Matrix<float, 4, 4> Matrix4f;
```

我们在下面讨论这些方便的 typedef。

Vectors

如上所述，在Eigen 中，向量只是矩阵的一种特殊情况，具有 1 行或 1 列。他们有 1 列的情况是最常见的；这样的向量称为列向量，通常简称为向量。在它们有 1 行的另一种情况下，它们被称为行向量。

例如，方便的 typedef `Vector3f`是 3 个浮点数的（列）向量。它由Eigen定义如下：

```
typedef Matrix<float, 3, 1> Vector3f;
```

我们还为行向量提供方便的 typedef，例如：

```
typedef Matrix<int, 1, 2> RowVector2i;
```

特殊值 Dynamic

当然，Eigen不限于其维度在编译时已知的矩阵。在`RowsAtCompileTime`和`ColsAtCompileTime`模板参数可以采取特殊值`Dynamic`这表明大小在编译时是未知的，所以必须作为运行时变量来处理。在Eigen术语中，这样的大小称为*动态大小*；而在编译时已知的大小称为*固定大小*。例如，方便的 typedef `MatrixXd`表示具有动态大小的双精度矩阵，定义如下：

```
typedef Matrix<double, Dynamic, Dynamic> MatrixXd;
```

同样，我们定义了一个不言自明的 typedef `VectorXi` 如下：

```
typedef Matrix<int, Dynamic, 1> VectorXi;
```

您可以完美地拥有例如固定数量的行和动态数量的列，如下所示：

```
Matrix<float, 3, Dynamic>
```

构造函数

默认构造函数始终可用，从不执行任何动态内存分配，也从不初始化矩阵系数。你可以做：

```
Matrix3f a;
MatrixXf b;
```

这里，

- `a` 是一个 3×3 矩阵，带有未初始化系数的普通 float[9] 数组，
- `b` 是一个动态大小的矩阵，其大小当前为 0×0，并且其系数数组尚未分配。

构造函数也可以定义大小。对于矩阵，总是首先传递行数。对于向量，只需传递向量大小。他们分配给定大小的系数数组，但不初始化矩阵内具体的值：

```
MatrixXf a(10,15);
VectorXf b(30);
```

这里，

- `a` 是一个 10x15 动态大小的矩阵，具有已分配但当前未初始化的值。
- `b` 是大小为 30 的动态大小向量，具有已分配但当前未初始化的值。

为了在固定大小和动态大小的矩阵之间提供统一的 API，在固定大小的矩阵上使用这些构造函数是合法的，即使在这种情况下传递大小是无用的。所以这是合法的：

```
Matrix3f a(3,3);
```

矩阵和向量也可以从系数列表中初始化。在 C++11 之前，此功能仅限于固定大小的列或最大大小为 4 的向量：

```
Vector2d a(5.0, 6.0);

Vector3d b(5.0, 6.0, 7.0);

Vector4d c(5.0, 6.0, 7.0, 8.0);
```

如果启用 C++11，则可以通过传递任意数量的系数来初始化任意大小的固定大小的列或行向量：

```
Vector2i a(1, 2);                       // 包含元素 {1, 2} 的列向量
Matrix<int, 5, 1> b {1, 2, 3, 4, 5};    // 包含元素 {1, 2, 3, 4, 5} 的行向量
Matrix<int, 1, 5> c = {1, 2, 3, 4, 5}; // 包含元素 {1, 2, 3, 4, 5} 的列向量
```

在具有固定或运行时大小的矩阵和向量的一般情况下，值必须按行分组并作为初始化列表（详细信息）的初始化列表传递：

```
MatrixXi a {      // 构造一个 2x2 的矩阵
      {1, 2},     // 第一行
      {3, 4}      // 第二行
};
Matrix<double, 2, 3> b {
      {2, 3, 4},
      {5, 6, 7},
};
```

对于列或行向量，允许隐式转置。这意味着可以从单行初始化列向量：

```
VectorXd a {{1.5, 2.5, 3.5}};             // 具有 3 个系数的列向量

RowVectorXd b {{1.0, 2.0, 3.0, 4.0}};     // 一个有 4 个系数的行向量
```

系数存取器

Eigen中的主要系数访问器和修改器是重载的括号运算符。对于矩阵，始终首先传递行索引。对于向量，只需传递一个索引。编号从 0 开始。这个例子是不言自明的：

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

using namespace Eigen;

int main()
{
    MatrixXd m(2,2);
    m(0,0) = 3;
    m(1,0) = 2.5;
    m(0,1) = -1;
    m(1,1) = m(1,0) + m(0,1);
    std::cout << "Here is the matrix m:\n" << m << std::endl;
    VectorXd v(2);
    v(0) = 4;
    v(1) = v(0) - 1;
    std::cout << "Here is the vector v:\n" << v << std::endl;
}

// 输出:
Here is the matrix m:
  3  -1
2.5 1.5
Here is the vector v:
4
3
```

请注意，语法`m(index)`不限于向量，它也可用于一般矩阵，这意味着在系数数组中进行基于索引的访问。然而，这取决于矩阵的存储顺序。**所有特征矩阵默认为列优先存储顺序**，但这可以更改为行优先，请参阅存储顺序。

operator[] 也被重载用于向量中基于索引的访问，但请记住，C++ 不允许 operator[] 接受多个参数。我们将 operator[] 限制为向量，因为 C++ 语言中的笨拙会使 matrix[i,j] 编译为与 matrix[j] 相同的东西！

逗号初始化

矩阵和向量系数可以使用所谓的 “逗号” 初始化语法方便地设置。现在，了解这个例子就足够了：

```
Matrix3f m;
m << 1, 2, 3,
     4, 5, 6,
     7, 8, 9;
std::cout << m;

// 输出:
1 2 3
4 5 6
7 8 9
```

调整大小

矩阵的当前大小可以通过rows()、cols()和size()检索。这些方法分别返回行数、列数和系数个数。调整动态大小矩阵的大小是通过resize()方法完成的。

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

using namespace Eigen;

int main()
{
    MatrixXd m(2,5);
    m.resize(4,3);
    std::cout << "The matrix m is of size "
        << m.rows() << "x" << m.cols() << std::endl;
    std::cout << "It has " << m.size() << " coefficients" << std::endl;
    VectorXd v(2);
    v.resize(5);
    std::cout << "The vector v is of size " << v.size() << std::endl;
    std::cout << "As a matrix, v is of size "
        << v.rows() << "x" << v.cols() << std::endl;
}

// 输出:
The matrix m is of size 4x3
It has 12 coefficients
The vector v is of size 5
As a matrix, v is of size 5x1
```

如果实际矩阵大小是固定的，则 resize() 方法是无操作的；否则它是破坏性的：系数的值可能会改变。如果你想要一个不改变系数的 resize() 的保守变体，请使用conservativeResize()，有关更多详细信息，请参阅此页面。

为了 API 的一致性，所有这些方法在固定大小的矩阵上仍然可用。当然，您实际上无法调整固定大小的矩阵的大小。尝试将固定大小更改为实际不同的值将触发断言失败；但以下代码是合法的：

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

using namespace Eigen;

int main()
{
    Matrix4d m;
    m.resize(4,4); // no operation
    std::cout << "The matrix m is of size " << m.rows() << "x" << m.cols()
<< std::endl;
}

// 输出:
The matrix m is of size 4x4
```

赋值和调整大小

赋值是使用将矩阵复制到另一个矩阵的操作 `operator=`。Eigen 自动调整左侧矩阵的大小，使其与右侧矩阵的大小匹配。例如：

```cpp
MatrixXf a(2,2);
std::cout << "a is of size " << a.rows() << "x" << a.cols() << std::endl;
MatrixXf b(3,3);
a = b;
std::cout << "a is now of size " << a.rows() << "x" << a.cols() << std::endl;

// 输出:
a is of size 2x2
a is now of size 3x3
```

当然，如果左侧是固定大小，则不允许调整大小。

如果您不希望发生这种自动调整大小（例如出于调试目的），您可以禁用它，请参阅此页面。

固定大小与动态大小

什么时候应该使用固定大小（例如 `Matrix4f`），什么时候应该使用动态大小（例如 `MatrixXf`）？简单的答案是：**尽可能对非常小的尺寸使用固定尺寸，对较大尺寸或必须使用的尺寸使用动态尺寸。**对于小尺寸，尤其是小于（大约）16 的尺寸，使用固定尺寸对性能非常有益，因为它允许Eigen避免动态内存分配并展开循环。在内部，固定大小的特征矩阵只是一个普通数组，即做

```cpp
Matrix4f mymatrix;
```

真的等于只是做

```cpp
float mymatrix[16];
```

所以这真的是零运行成本。相比之下，动态大小矩阵的数组总是分配在堆上，所以这样做

```cpp
MatrixXf mymatrix(rows, columns);
```

等于做

```
float *mymatrix = new float[rows *columns];
```

除此之外，MatrixXf 对象将其行数和列数存储为成员变量。

当然，使用固定大小的限制是只有当您在编译时知道大小时才可能这样做。此外，**对于足够大的尺寸，比如大于（大约）32 的尺寸，使用固定尺寸的性能优势变得可以忽略不计。更糟糕的是，尝试在函数内部使用固定大小创建一个非常大的矩阵可能会导致堆栈溢出，因为Eigen会尝试自动分配数组作为局部变量，而这通常是在堆栈上完成的。**最后，根据情况，当使用动态大小时，Eigen也可以更积极地尝试矢量化（使用 SIMD 指令），请参阅Vectorization。

可选模板参数

我们在本页开头提到Matrix类需要六个模板参数，但到目前为止我们只讨论了前三个。其余三个参数是可选的。这是模板参数的完整列表：

```
Matrix<typename Scalar,
       int RowsAtCompileTime,
       int ColsAtCompileTime,
       int Options = 0,
       int MaxRowsAtCompileTime = RowsAtCompileTime,
       int MaxColsAtCompileTime = ColsAtCompileTime>
```

- Options 是一个位域。在这里，我们只讨论一点：RowMajor， 它指定这种类型的矩阵使用行优先存储顺序；默认情况下，存储顺序为列优先。例如，这种类型表示行主 3x3 矩阵：

```
Matrix<float, 3, 3, RowMajor>
```

- `MaxRowsAtCompileTime` 和 `MaxColsAtCompileTime` 在您想要指定时很有用，**即使在编译时不知道矩阵的确切大小，但在编译时知道固定的上限。您可能想要这样做的最大原因是避免动态内存分配。**例如，以下矩阵类型使用 12 个浮点数的普通数组，没有动态内存分配：

```
Matrix<float, Dynamic, Dynamic, 0, 3, 4>
```

方便的类型定义

Eigen定义了以下矩阵类型定义：

- MatrixNt 用于 Matrix<type, N, N>。例如，MatrixXi 表示 Matrix<int, Dynamic, Dynamic>。
- Matrix<type, N, 1> 的 VectorNt。例如，用于 Matrix<float, 2, 1> 的 Vector2f。
- Matrix<type, 1, N> 的 RowVectorNt。例如，用于 Matrix<double, 1, 3> 的 RowVector3d。

注：

- N 可以是 `2`，`3`，`4`，或 `X`（意思是 `Dynamic`）中的任何一个。
- t 可以是 `i`（表示 int）、`f`（表示 float）、`d`（表示双精度）、`cf`（表示 complex<float>）或 `cd`（表示 complex<double>）中的任何一种。typedef 仅针对这五种类型定义的事实并不意味着它们是唯一受支持的标量类型。例如，支持所有标准整数类型，请参阅标量类型。