从代码到AI

Eigen 新手入门教程

作者：pengliustd@163.com
简介：电子科技大学硕士学位，曾就职于华为 OS 内核实验室，已发表多篇 SCI

目 录

第一章 了解 Eigen

Eigen 是使用 C++模板元编程技术构建的高效矩阵计算库，在人工智能、自动驾驶、机器人、工程数值模拟等领域应用广泛。例如，人工智能领域 TensorFlow，自动驾驶领域 AirSim 等都把 Eigen 作为基础的矩阵计算库。

截止 2022.09，Eigen 社区依然非常活跃。全世界的开发者都在为 Eigen 支持新的功能而贡献代码，以不断适应人工智能、自动驾驶等领域的新需求，社区贡献网址 https://gitlab.com/libeigen/eigen/-/merge_requests。

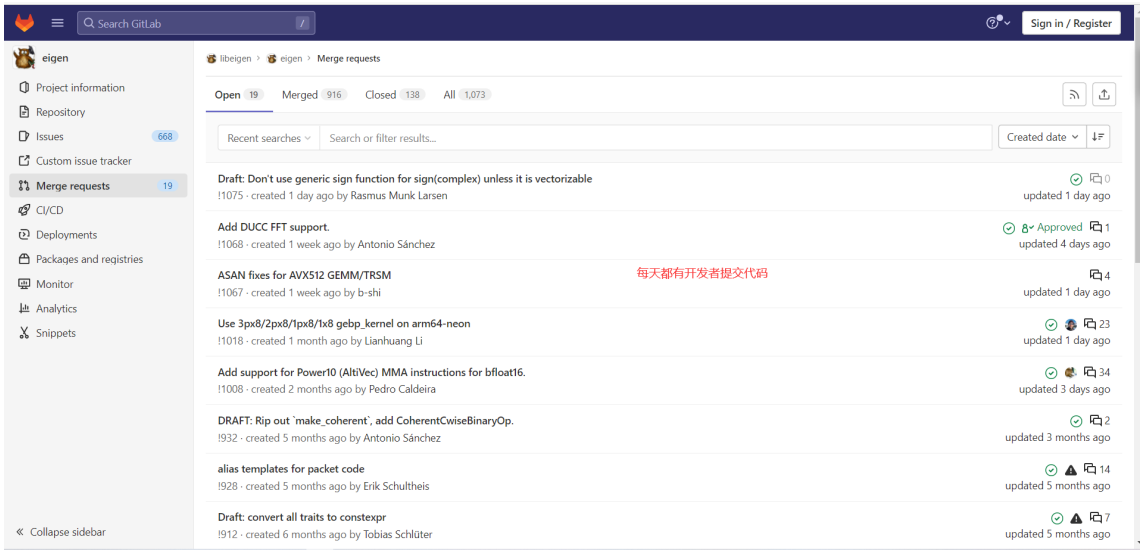

图 1-1 Eigen 社区充满活力。

Eigen 因为使用模板元编程技术，所以使用该库只需要包含相应头文件即可，非常方便。Eigen 使用 CPU 计算时效率非常高，但是 Eigen 对 GPU 的支持却很差。如果需要使用 GPU 进行矩阵计算，那么推荐使用 CUDA。

第二章 安装 Eigen

2.1 源码下载

建议下载最新稳定版，Windows 环境下可下载“zip”格式的文件，Linux 环境可下载“tar.gz”格式的文件，当前最新 Eigen 已经更新到 3.4.0。Eigen 源码下载：http://eigen.tuxfamily.org/index.php?title=Main_Page。

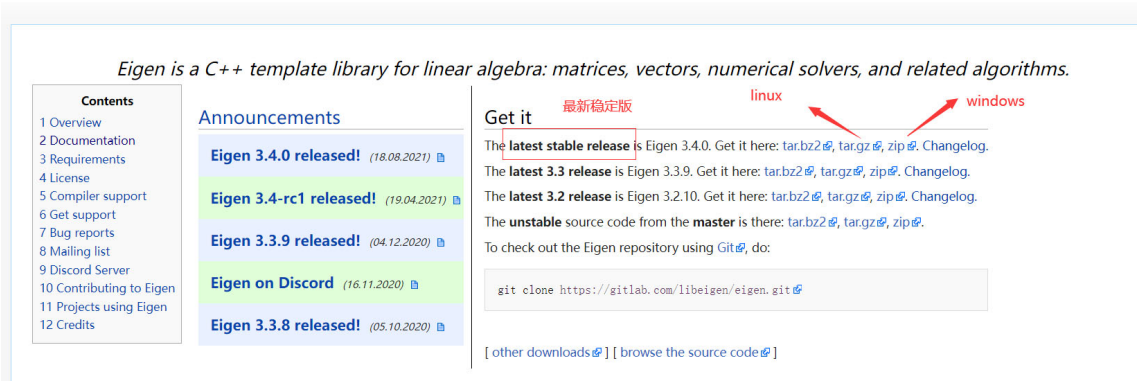

图 2-1 Eigen 官网截图，推荐最新稳定版

2.2 Win10+VS2019

解压源码，进入该文件夹，即可看到如下文件目录。在图 2-2 中，Eigen 文件夹是我们需要使用的源文件目录，通常只要将该文件复制到对应的工程即可使用 Eigen 库。进入 Eigen 文件夹，可以看到有一个 Dense 文件，这是一个头文件接口，在使用 Eigen 库的源码中需要包含该头文件。

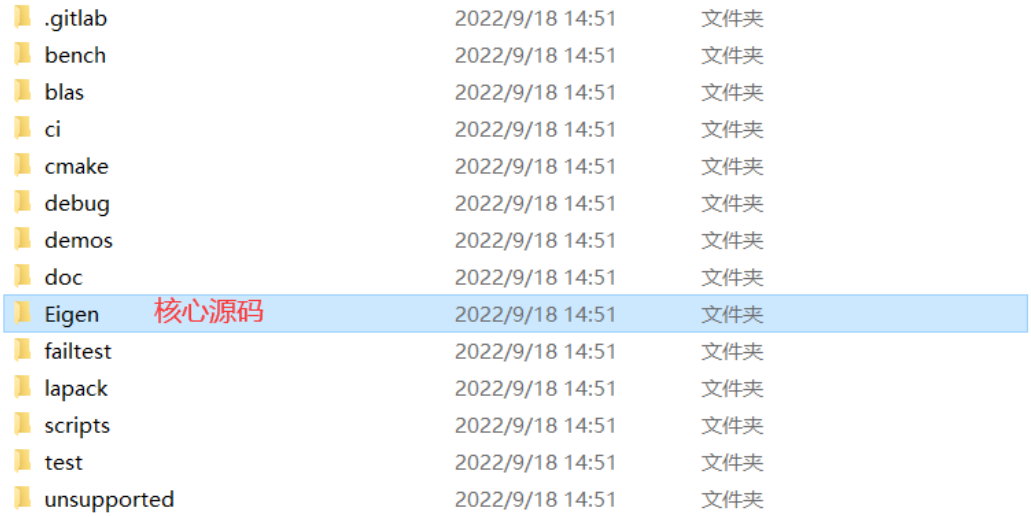

图 2-2 Eigen3.4.0 源码目录

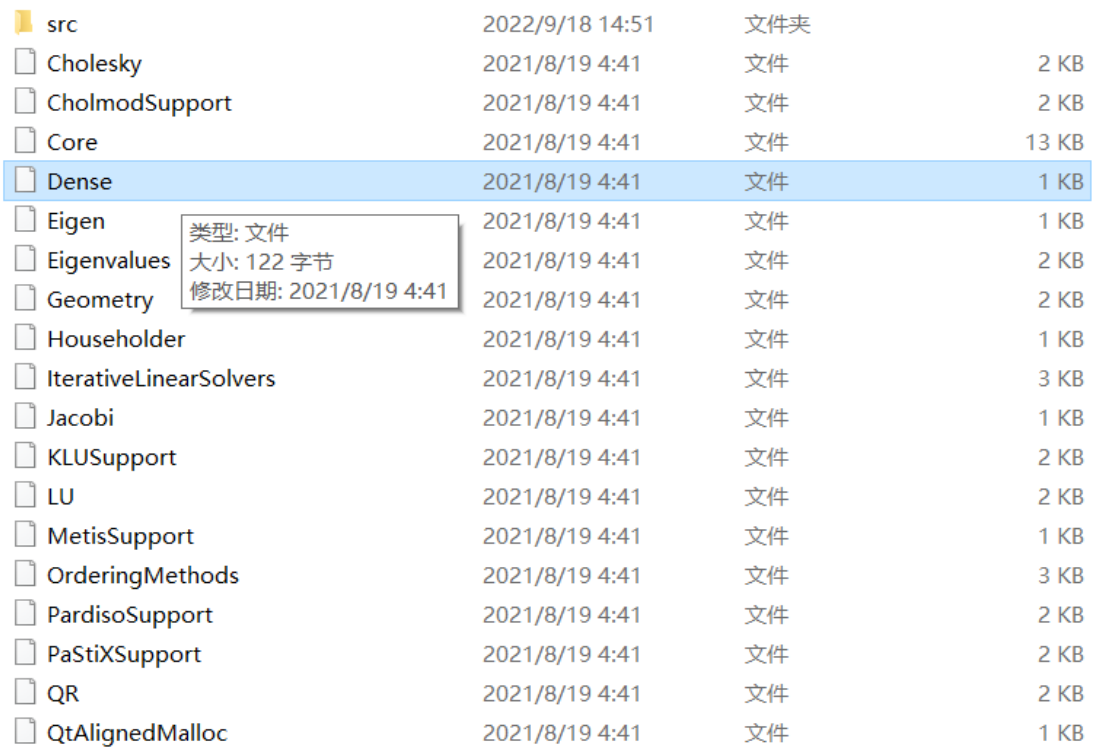

图 2-3 Eigen 目录

下面介绍 Eigen 如何使用的一个简单用例，但是此处不解释代码。其它版本的 Visual Studio 有着类似的操作方法。

1) 打开 Visual Studio2019，新建一个 C++项目(依次点击 文件->新建->项目)，养成自己设置工程路径的好习惯，一般不将工程建在 C 盘，如图 2-4。

图 2-4 vs2019 新建项目

2) 进入到工程存储的位置，查看工程目录，熟悉 vs 创建的目录结构，以创

建一个名为 vs2019-eigen 项目为例，其目录结构如图 2-5 所示。

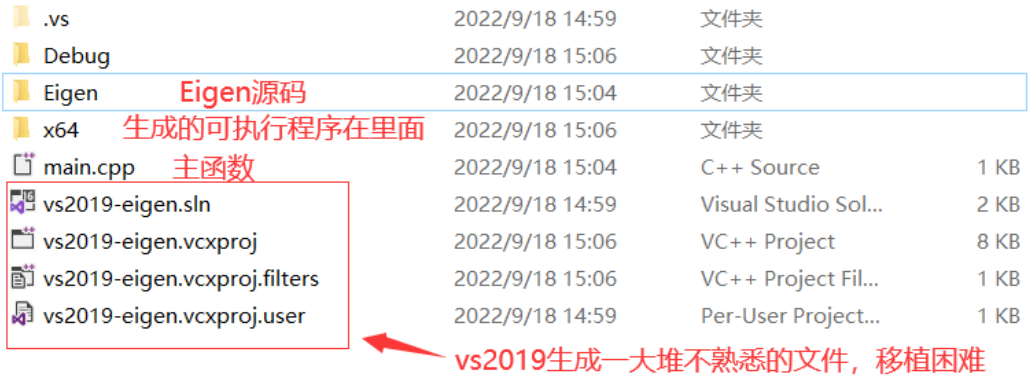

图 2-5 vs2019 新建项目目录结构

3) 在 vs2019 中添加源文件，命名为 main.cpp，如图 2-6 所示，源码中包含了 Dense 文件，这个文件是使用 Eigen 库的接口。

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

using std::cout;
using std::endl;

int main()
{
 Eigen::Vector3d x;
 x.setZero();
 x(0) = -0.1;
 std::cout << x << endl;

 return 0;
}
```

图 2-6 调用 Eigen 简单运行示例

4) 将图 2-2 中的 Eigen 文件夹复制到图 2-5 vs2019-eigen 解决方案目录下，保证 Eigen 源码在工程的可访问目录下，按图 2-7 的方法可以将 Eigen 添加至工程可访问的目录。VS2019 运行代码是很快，但是各种各样的配置着实有点混乱。

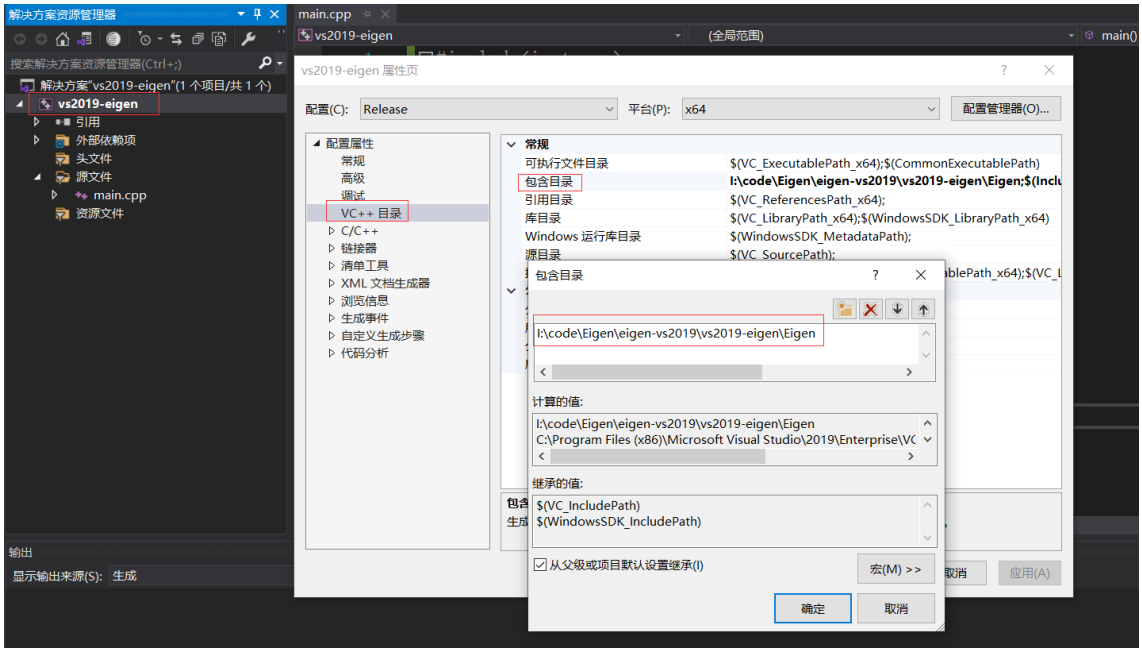

图 2-7 调用 Eigen 简单运行示例

5) 编译运行，如果可以正常编译运行，那么说明 Eigen 库已经成功配置。

2.3 Ubuntu+VScode

Ubuntu 桌面操作系统广受开发者的喜爱，我只能说 Ubuntu+VScode 无比强大。在前面的方案中，每构建一个工程就将 Eigen 复制一份。这里介绍使用 Eigen 的另外一种方法，将 Eigen 添加到程序可访问的公共目录，使得 Eigen 可以和 iostream 一样使用。如图 2-8 所示，将 Eigen 复制到了/usr/include 目录下，该目录默认可以被程序访问，所以 Eigen 写成如下的包含形式。

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

```
root@ubuntu:/usr/include# ls |grep Eigen
Eigen
```

图 2-8 将 Eigen 复制到/usr/include 下，程序默认可访问这个目录

补充介绍 VScode 如何创建工程，只要安装插件 C/C++ Project Generator，如图 2-9 所示。

1) 创建工程。打开命令面板，模糊搜索“create”即可弹出匹配到的选项，其中就有创建 C 或者 C++工程的选项。

2) 修改 main 函数，代码与图 2-6 显示的代码一致。因为已经将 Eigen 添加到了/usr/include 目录，所以可以直接引用 Eigen 了，非常便捷。

3) VScode 按 F5 进行调试，按 CTRL+F5 运行。

图 2-9 安装 C/C++ Project Generator

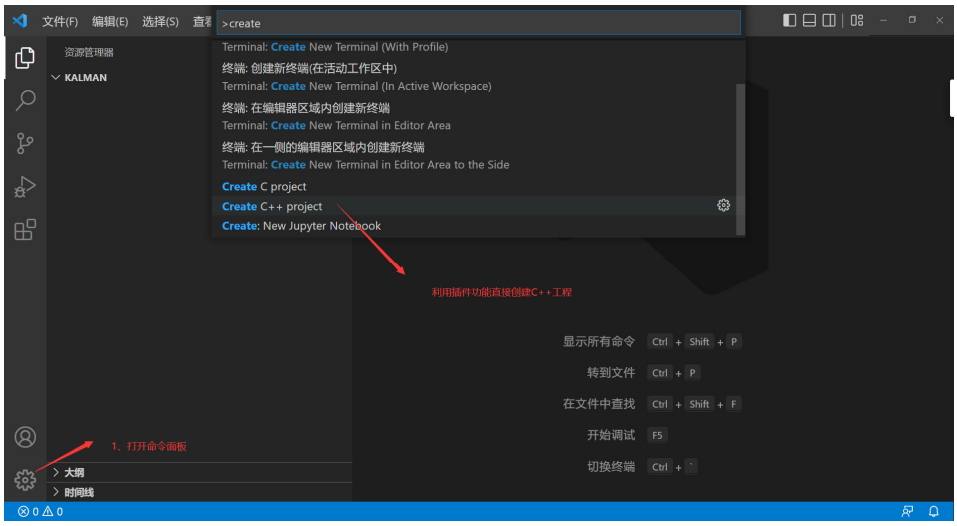

图 2-10 使用 C/C++ Project Generator 创建工程

第三章 Eigen 基础知识

在 Eigen 中所有的矩阵和向量都是 Matrix 模板类的对象，向量只是一种特殊的矩阵。

3.1 矩阵类 Matrix

Matrix 类总共有六个模板参数首先只介绍前三个参数，剩下的三个参数有其默认值。三个强制型的参数如下：

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

Scalar 是 scalar 类型，如想要构造一个单精度的浮点类型矩阵，可以选择 float。所有支持的 Scalar 类型参见 Scalar types。

RowsAtCompileTime 和 ColsAtCompileTime 分别表示行数和列数，要求这两个参数在编译时已知。

在 Eigen 中提供宏定义来便捷的访问一些常用的类型，如：

```
typedef Matrix<float, 4, 4> Matrix4f;
typedef Matrix<float, 3, 1> Vector3f;
typedef Matrix<int, 1, 2> RowVector2i;
```

当然，Eigen 并不局限于那些矩阵维数在编译时已知的情形。RowsAtCompileTime 和 ColsAtCompileTime 可以取一个特殊的值 Dynamic，这表示矩阵的维度在编译时是未知的，必须作为一个运行时变量来处理。在 Eigen 术语中，Dynamic 称为动态大小(dynamic size)，而运行时已知的大小称为固定大小(fixed size)。

```
// 创建一个双精度的动态矩阵
typedef Matrix<double, Dynamic, Dynamic> MatrixXd;
// 创建一个整型列向量
typedef Matrix<int, Dynamic, 1> VectorXi;
```

Matrix 的另外三个模板参数是可以选择的，完整的参数如下：

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

Options 是一个位域，它的值只能取 RowMajor、ColMajor，分别表示行优先存

储、列优先存储。默认情况下是使用列优先存储，一般地 Eigen 库在列优先存储的情况下效率要高。Eigen 的存储形式的指定，只是影响矩阵在内存的形式，不会影响矩阵的访问习惯。

MaxRowsAtCompileTime 和 MaxColsAtCompileTime 用于指定矩阵的维数的最大值。尽管有时候不知道矩阵的确切大小，但在编译时已经知道了矩阵维数的上界，那么可以指定这两个参数来避免动态内存分配。

这里强调一下，Eigen 默认是列优先的，比如定义一个 Vector3d 向量，对应的维度是 3 行 1 列。 列向量默认

3.2 矩阵初始化与访问

Eigen 提供了默认构造函数，它不会提供动态内存的分配，也不会初始化任何矩阵的值。Eigen 类型可以这样使用：

```
// 一个3*3 的矩阵，矩阵的元素都没有被初始化。
Matrix3f a;
// 一个动态矩阵，它的大小是0*0，也就是说还没有为该矩阵分配内存。
MatrixXf b;
```

构造函数提供指定矩阵大小的重载。对矩阵来说，第一个参数是矩阵的行数。对向量来说只需指定向量的大小。它会分配矩阵或向量所需的内存的大小，但是不会初始化他们的值。

```
// 一个10*15 的动态矩阵，内存进行了分配，但没有初始化。
MatrixXf a(10,15);
// 一个大小为30 的动态数组，但没有初始化。
VectorXf b(30);
```

Eigen 库重载圆括号()访问矩阵或者向量的元素，序号从 0 开始。Eigen 不支持使用方括号[]访问矩阵的元素(向量除外)。

```
MatrixXd m(2,2);
m(0,0) = 3;
VectorXd v(2);
v(0) = 4;
```

逗号表达式初始化

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

可以使用 rows()、cols()、size()访问矩阵当前大小，使用 resize()重置矩阵的大小。如果矩阵的大小没有变化，那么 resize()操作没有任何影响。如果矩阵的大小

改变了，那么矩阵的值可能会改变。如果你想在重设大小过程中不改变矩阵的值使用 conservativeResize()。使用赋值操作符“=”，Eigen 将左操作数的大小重置为右操作数的大小。

3.3 矩阵和向量代数

加减法：重载 C++中“+”、“-”、“+=”、“-=”操作符，要求左右操作数的维度相同。不允许一个向量加上或者减去一个数。只要维度一样，向量和矩阵是可以相加的，这也说明了向量是特殊的矩阵。

数乘与数除：重载 C++中“*”、“/”、“*=”、“/=”操作符，支持矩阵和向量乘以或者除以一个数。

转置与共轭：转置 a^T、共轭 $\bar{a}$ 、共轭转置 a^H 分别通过 transpose()、conjugate()、adjoint()实现。调用格式 a.transpose()，a.conjugate()，a.adjoint()。对于实数而言，共轭没有任何影响，共轭转置等价于转置。使用 a = a.transpose()可能会出现错误，这是因为 Eigen 在进行转置或者共轭操作时，会同时写左操作数，从而得到意想不到的结果。要实现这种功能可以使用 a.transposeInPlace()。类似的，也支持 adjointInPlace()。

矩阵-矩阵与矩阵-向量乘法：由于在 Eigen 中向量只是特殊的矩阵，因此只需重载“*”、“*=”即可实现矩阵和向量的乘法。如果你担心 m=m*m 会导致混淆，现在可以消除这个疑虑，因为 Eigen 以一种特殊的方式处理矩阵乘法，编译 m=m*m 时，作为。

```
tmp = m*m;
m = tmp;
```

点积和叉乘：点积又可以称为内积，Eigen 分别使用 dot()和 cross()来实现内积和向量积。叉乘只适用于三维向量。

```
Vector3d v(1,2,3);
Vector3d w(0,1,2);
v.dot(w);
v.cross(w);
```

基础的代数计算：mat.sum()计算所有矩阵元素的和，mat.pro()计算所有元素的连乘积，mat.mean()计算所有元素的平均值，mat.minCoeff()计算矩阵元素的最小值，mat.maxCoeff 计算矩阵元素的最大值，mat.trace()计算矩阵的迹。计算最大值和最小值的函数支持返回最大值和最小值的位置：

```
Matrix3f m = Matrix3f::Random();
//ptrdiff_t 是 stddef.h 中用于表示两个指针见的间隔的数据类型，是有符号型
std::ptrdiff_t i, j;
```

```
float minOfM = m.minCoeff(&i,&j);
```

以上的操作都不会影响操作数本身。在计算过程中如果由于矩阵维度不满足相应的条件，那么 Eigen 可以在编译时检查出静态矩阵维度的矛盾，对于动态矩阵有相应动态检查方法，如果出现维度矛盾可能会使程序崩溃。通过“.”操作符调用的操作可以作为左值，当然这在一般情况下是没多大意义的。

经过编程实验发现，如果在进行矩阵运算时矩阵的维度不匹配，那么 Eigen 会因为调用了 assert 而使程序 abort。如果不想程序 abort，那么可以在包含 Eigen 头文件之前添加#define NDEBUG。这里强调一下，生产程序中如果要使用 Eigen，**用户代码需要保证矩阵维度能够正确匹配**。

3.4 数组类 Array

Array 类提供通常意义上的数组，它提供一些方便的对元素的非线性操作。例如让所有元素都加上一个常量，或者让两个 Arrays 的值对应相乘。Array 类模板与 Matrix 相似，其含义参见 Matrix 类介绍。Array 在其存储形式上具有矩阵的形式，只是说 Array 支持的运算和 Matrix 不一样。

```
Array<typename Scalar, int RowsAtCompileTime, int ColsAtCompileTime>;
Array<float,Dynamic,1> ArrayXf;
Array<float,3,1> Array3f;
Array<double,Dynamic,Dynamic> ArrayXXd;
```

访问 Array 中的值的方式和 Matrix 是一样的。下面介绍和 Matrix 不一样的操作：

加减乘除：将对应位置元素相加（减、乘、除）。

```
ArrayXXf a(2,2);
ArrayXXf b(2,2);
a << 1,2,3,4;
b << 5,6,7,8;
cout<<a*b;

输出结果为:
5 12
21 32
```

平方根、绝对值：通过调用 sqrt()、abs()函数，a.abs()。

求两个矩阵的最小值：a.min(b)，要求 a 和 b 的维度相同，求对应位置的两个数的最小值。

Eigen 支持的数组元操作见图 3-1 所示。那么，如何选择 Matrix 和 Array 呢？如果要支持线性代数的操作，就选择 Matrix；如果要进行 coefficient-wise 操作，就

选择 Array。如果又要支持线性代数操作，又要支持 coefficient-wise 操作，那么就可以使用.array()和.matrix()实现类型的转换，这种转换是没有计算代价的。这些操作也不会改变调用该函数的矩阵或者数组本身而是返回一个副本。

```
1.  array1.abs2()
2.  array1.abs()                abs(array1)
3.  array1.sqrt()               sqrt(array1)
4.  array1.log()                log(array1)
5.  array1.log10()              log10(array1)
6.  array1.exp()                exp(array1)
7.  array1.pow(array2)          pow(array1,array2)
8.  array1.pow(scalar)          pow(array1,scalar)
9.                              pow(scalar,array2)
10. array1.square()
11. array1.cube()
12. array1.inverse()
13.
14. array1.sin()                sin(array1)
15. array1.cos()                cos(array1)
16. array1.tan()                tan(array1)
17. array1.asin()               asin(array1)
18. array1.acos()               acos(array1)
19. array1.atan()               atan(array1)
20. array1.sinh()               sinh(array1)
21. array1.cosh()               cosh(array1)
22. array1.tanh()               tanh(array1)
23. array1.arg()                arg(array1)
24.
25. array1.floor()              floor(array1)
26. array1.ceil()               ceil(array1)
27. array1.round()              round(aray1)
28.
29. array1.isFinite()           isfinite(array1)
30. array1.isInf()              isinf(array1)
31. array1.isNaN()              isnan(array1)
```

图 3-1 数组元支持的操作

3.5 块操作

块是矩阵的一个矩形区域，块表达式可以是左值也可以是右值。最常用的是block()操作，它有两个版本：

```
// 动态大小的块
matrix.block(i,j,p,q);
// 固定大小的块
matrix.block<p,q>(i,j);
```

i、j 表示块的起始位置(块的左上角元素在矩阵中的角标)，p、q 表示块的大小。固定大小的块在运行时效率要高。

```
Eigen::MatrixXf m(4,4);
  m <<  1, 2, 3, 4,
        5, 6, 7, 8,
        9,10,11,12,
       13,14,15,16;
cout << m.block(0,0,i,i) << endl;
```

```
结果:
Block of size 3x3
  1  2  3
  5  6  7
  9 10  11
```

```
// block()操作也可用于左值:
MatrixXd m(3,3);
VectorXd v(3);
m << 1, 2, 3,
 4, 5, 6,
 7, 8, 9;
v << 11, 12, 13;
m.block(0, 0, 3, 1) = v;
cout << m.block(0,0,3,1) << endl;
```

```
结果:
  11
  12
  13
```

block()操作支持任意的块操作，对于一些特殊的访问 Eigen 也提供了 API。

```
Block operation Method
ith row * matrix.row(i);
jth column * matrix.col(j);
```

同样的，row()操作和 col ()操作既可以是左值又可以是右值。除此之外，Eigen 还提供了访问一些特殊位置的块的快捷操作，如图 3-2 所示。

对于向量或者说一维数组，Eigen 也提供了特殊的块操作，如图 3-3 所示。这些块操作也有静态和动态两个版本。v.head(*n*)可以访问向量的头 *n* 个元素，v.tail(*n*)可以访问向量的尾 *n* 个元素，v.segment(*i*,*j*)可以访问从标号为 *i* 开始的 *j* 个元素。

Block operation	Version constructing a dynamic-size block expression	Version constructing a fixed-size block expression
Top-left p by q block *	`matrix.topLeftCorner(p,q);`	`matrix.topLeftCorner<p,q>();`
Bottom-left p by q block *	`matrix.bottomLeftCorner(p,q);`	`matrix.bottomLeftCorner<p,q>();`
Top-right p by q block *	`matrix.topRightCorner(p,q);`	`matrix.topRightCorner<p,q>();`
Bottom-right p by q block *	`matrix.bottomRightCorner(p,q);`	`matrix.bottomRightCorner<p,q>();`
Block containing the first q rows *	`matrix.topRows(q);`	`matrix.topRows<q>();`
Block containing the last q rows *	`matrix.bottomRows(q);`	`matrix.bottomRows<q>();`
Block containing the first p columns *	`matrix.leftCols(p);`	`matrix.leftCols<p>();`
Block containing the last q columns *	`matrix.rightCols(q);`	`matrix.rightCols<q>();`

图 3-2 Eigen 支持的块便捷操作

3.6 库函数

随机向量或矩阵生成

使用 Random 函数，生成双精度的随机数在-1 到 1 之间，生成整数的随机数在某一个负整数到某一个正整数之间与机器有关。样例：

```
MatrixXd m=MatrixXd::Random(2,3);
VectorXi v= VectorXi::Random(1);
```

零向量或矩阵生成

使用 Zero()函数，生成值全为零的矩阵或向量。

```
MatrixXd m=MatrixXd::Zero(2,3);
VectorXi v= VectorXi::Zero(1);
```

生成单位矩阵

使用 Identity()函数，生成单位矩阵或准单位矩阵。

```
MatrixXd m= MatrixXd::Identity(5,5);
MatrixXd m= MatrixXd::Identity(5,4);
```