1.1矩阵类

1.矩阵的模板参数

eigen库中的矩阵是以模板类出现的，因此，若要使用矩阵，需要先实例化特定的矩阵类。

Matrix<**typename** scalar,**int** rowatcomiletime,**int** colsatcompiletime>

scalar表示矩阵中系数的类型， rowatcomiletime和 colsatcompiletime表示行数和列数。

同时，eigen库也为特例化(方阵)的矩阵模板定义了类型别名，例如：

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

浮点型的4阶方阵，可以表示为Matrix4f,其中4表示阶数，f表示类型。同样适用于其他阶数的其他类型的矩阵。同时还能用于动态绑定（即未知阶数）的方阵如下：

**typedef** Matrix<**float**,dynamic,dynamic> MatrixXf;

2.向量

向量就是矩阵的特殊形式，只有一行或一列。

例如，三维列向量可定义为

Matrix<**float** ,3,1> vector;

同样的，eigen库也为向量定义了类型别名：

列向量的类型别名为：

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

行向量的类型别名为：

**typedef** Matrix<**float** ,1,2> RowVector2f;

3.动态绑定值

eigen库为未知维数的矩阵提供了动态大小，只有运行时才确定其维数。定义形式：

Matrix<**float**,Dynamic,Dynamic>

同样地，也为未知维数的矩阵和向量定义了类型别名：

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

**typedef** Matrix<**double** ,Dynamic,1> VectorXd;

**typedef** Matrix<**double** ,1,Dynamic> RowVectorXd;

也可以只定义行数或者列数是动态值。

4.构造函数

（1）默认构造函数

Matrix3f a;

MatrixXf b;

这里，a是一个3x3的矩阵，系数未初始化。b是一个动态大小的矩阵，默认它为0x0维，它的系数数组还未分配内存。

（2）接受行数和列数的构造函数

MatrixXf a(10,15);

VectorXf b(30);

这里，as是一个动态大小矩阵，已分配了包含10x15维的数组内存，但元素未初始化。b是一个动态大小的向量，初始化为30维，分配了包含30维的数组内存，但元素未初始化。

为了统一构造函数接口，在特定大小矩阵中也可以使用接受行数和列数的构造函数，即：

Matrix3f a(3,3);

（3）初始化向量系数的构造函数

提供一些构造函数，来初始化向量维数小于等于4维的系数：

Vector2d a(5.0, 6.0);//初始化2维列向量，元素值分别为5.0和6.0

Vector3d b(5.0, 6.0, 7.0);//同上

Vector4d c(5.0, 6.0, 7.0, 8.0);//同上

记住是小于等于4维的向量才能这样初始化元素值。

3.系数访问

对矩阵和向量的系数访问是重载的输出运算符，对矩阵来说先传递行索引。

#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;//对矩阵的访问

VectorXd v(2);

v(0) = 4;

v(1) = v(0) - 1;

std::cout << "Here is the vector v:\n" << v << std::endl;//对向量的访问

}

4.系数初始化

对矩阵和向量系数的初始化可以使用输出运算符

Matrix3f m;

m << 1, 2, 3,

4, 5, 6,

7, 8, 9;

std::cout << m;

5.检索大小

矩阵的大小可以根据成员函数row(),cols(),size()来确定，它们分别返回矩阵的行数，列数和元素的数目。改变动态绑定的矩阵大小可以使用resize()函数：

#include <iostream>

#include <Eigen/Dense>

using namespace Eigen;

int main()

{

MatrixXd m(2,5);

m.resize(4,3);//改变动态大小

std::cout << "The matrix m is of size "

<< m.rows() << "x" << m.cols() << std::endl;//行数、列数

std::cout << "It has " << m.size() << " coefficients" << std::endl;//元素数目

VectorXd v(2);

v.resize(5);//改变向量大小

std::cout << "The vector v is of size " << v.size() << std::endl;//元素数目

std::cout << "As a matrix, v is of size "

<< v.rows() << "x" << v.cols() << std::endl;//行数、列数

}

如果实际的矩阵大小是不变的，是不可以使用resize()的成员函数的，不然它会被破坏：这样的化矩阵内元素的值可能会改变。如果想保持一个保守的resize()变量，即不改变元素值，应使用conservativeResize()成员函数。

为了统一接口，resize()对于固定大小的矩阵还是有效的，但是你不能实际地改变固定大小的矩阵。如果试图将固定大小矩阵改为不同维度的矩阵将触发插入失败。但是以下的代码是可接受的：

#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;

}

6.赋值和调整大小

矩阵赋值是利用=操作符。eigen调整等号左边的矩阵大小等于等号右边的矩阵大小，例如：

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;

a is of size 2x2

a is now of size 3x3

如果等号左边的矩阵是固定大小的，那么将无法进行赋值改变操作。

如果逆你不想赋值自动改变矩阵，你可以让它失效，具体详见此页

7.固定大小和动态大小

对于小维度的矩阵，我们使用固定大小；对于较大维度和动态分配时的矩阵，我们使用动态绑定的矩阵。

对于小维度矩阵，特别是维数小于16的，使用固定大小的矩阵提高程序运行效率，因为它避开了动态内存分配和循环展开。

一个固定大小的矩阵就相当与是一个普通的数组：

Matrix4f mymatrix;等价于 float mymatrix[16];

因此，固定大小矩阵只有零运行时间成本。相反，动态大小的矩阵分配数组进堆：

MatrixXf mymatrix(rows,columns);

等价于

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

除此之外，对动态绑定的矩阵还需要保存它们的行数和列数作为数据成员。

对于维数足够大的矩阵，在32维以上的矩阵，利用固定大小的矩阵运行效率的优点将会忽略不计。

更糟糕的是，试图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将尝试将数组自动分配为局部变量，而这通常是在堆栈上完成的。 最后，根据具体情况，当使用动态大小时，EIGEN还可以更积极地尝试矢量化（使用SIMD指令），请参见[Vectorization](http://eigen.tuxfamily.org/dox/TopicVectorization.html)。

8.可选择的矩阵模板参数

目前为止，我们主要讨论的是三个模板参数的矩阵类，其实模板矩阵类有六个模板参数，另外三个模板参数是默认实参，我们也可以修改定义：

Matrix<typename Scalar,

int RowsAtCompileTime,

int ColsAtCompileTime,

int Options = 0,

int MaxRowsAtCompileTime = RowsAtCompileTime,

int MaxColsAtCompileTime = ColsAtCompileTime>

前三个和之前一样，表示元素类型、行数、列数。

（1）Options表示矩阵存储顺序，存储顺序主要有两种，一种是以行为主的储存顺序（RowMajor），一种是以列为主的储存顺序(ColMajor)。

例：一个A矩阵

如果这个矩阵是以行为主的存储顺序的话，在内存中的放置顺序就是：

822991443545

如果是以列为主的存储顺序的话，在内存中的放置顺序就是：

893215244945

为了解释这个存储顺序，使用data()成员函数，这个函数返回指向矩阵存储位置的首元素的内存地址的指针：

Matrix<int, 3, 4, ColMajor> Acolmajor;//以列为主的存储顺序

Acolmajor << 8, 2, 2, 9,

9, 1, 4, 4,

3, 5, 4, 5;

cout << "The matrix A:" << endl;

cout << Acolmajor << endl << endl;

cout << "In memory (column-major):" << endl;

for (int i = 0; i < Acolmajor.size(); i++)

cout << \*(Acolmajor.data() + i) << " ";//使用data（）函数访问存储矩阵的首元素指针

cout << endl << endl;

Matrix<int, 3, 4, RowMajor> Arowmajor = Acolmajor;

cout << "In memory (row-major):" << endl;

for (int i = 0; i < Arowmajor.size(); i++)

cout << \*(Arowmajor.data() + i) << " ";

cout << endl;

eigen默认矩阵中的元素是以列为主的存储顺序。

（2）MaxRowsAtCompileTime 和 MaxColsAtCompileTime则表示定义矩阵维度的最大值。这个主要是为了避免动态分配内存的情况下使用，默认时是等于行数和列数的：

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

以上的例子定义了一个普通的12个浮点元素的数组，没有使用动态内存分配。

9.简单的类型别名

eigen定义了如下的矩阵类型别名：

* MatrixNt 表示 Matrix<type, N, N>
* VectorNt 表示 Matrix<type, N, 1>
* RowVectorNt 表示 Matrix<type, 1, N>.

其中，N代表维数，t代表类型的第一个字母。

t可以代表这些类型的任一一种：i(int),f(float),d(double),cf(复数类型的float)和cd(复数类型的double)，同时t可以表示任何标准内置类型！