矩阵和向量运算

本节旨在提供有关如何使用Eigen在矩阵、向量和标量之间执行算术的概述和一些详细信息。

介绍

Eigen通过重载常见的 C++ 算术运算符（例如 +、-、*）或通过特殊方法（例如 dot()、cross() 等）提供矩阵/向量算术运算。对于Matrix类（矩阵和向量），运算符仅重载以支持线性代数运算。例如，`matrix1 * matrix2` 表示矩阵-矩阵乘积，**并且 `vector + scalar` 是不允许的。**如果要执行各种数组运算，而不是线性代数，请参阅下一页。

加减

当然，**左侧和右侧必须具有相同的行数和列数。**它们还**必须具有相同的 `Scalar` 类型**，因为Eigen不进行自动类型提升。这里的操作员是：

- 二元运算符 + as in `a+b`
- 二元运算符 - 如 `a-b`
- 一元运算符 - 如 `-a`
- 复合运算符 += 如 `a+=b`
- 复合运算符 -= 如 `a-=b`

例子：

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

using namespace Eigen;

int main()
{
    Matrix2d a;
    a << 1, 2,
    3, 4;
    MatrixXd b(2,2);
    b << 2, 3,
    1, 4;
    std::cout << "a + b =\n" << a + b << std::endl;
    std::cout << "a - b =\n" << a - b << std::endl;
    std::cout << "Doing a += b;" << std::endl;
    a += b;
    std::cout << "Now a =\n" << a << std::endl;
    Vector3d v(1,2,3);
    Vector3d w(1,0,0);
    std::cout << "-v + w - v =\n" << -v + w - v << std::endl;
}
```

输出：

```
a + b =
3 5
4 8
```

```
a - b =
-1 -1
 2  0
Doing a += b;
Now a =
3 5
4 8
-v + w - v =
-1
-4
-6
```

标量乘除

标量的乘法和除法也非常简单。这里的操作员是：

- 二元运算符 * 如 `matrix*scalar`
- 二元运算符 * 如 `scalar*matrix`
- 二元运算符 / 如 `matrix/scalar`
- 复合运算符 *= 如 `matrix*=scalar`
- 复合运算符 /= 如 `matrix/=scalar`

例子：

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

using namespace Eigen;

int main()
{
    Matrix2d a;
    a << 1, 2,
    3, 4;
    Vector3d v(1,2,3);
    std::cout << "a * 2.5 =\n" << a * 2.5 << std::endl;
    std::cout << "0.1 * v =\n" << 0.1 * v << std::endl;
    std::cout << "Doing v *= 2;" << std::endl;
    v *= 2;
    std::cout << "Now v =\n" << v << std::endl;
}
```

输出：

```
a * 2.5 =
2.5   5
7.5  10
0.1 * v =
0.1
0.2
0.3
Doing v *= 2;
Now v =
2
4
6
```

关于表达式模板的说明

这是我们在此页面上解释的高级主题，但现在仅提及它很有用。在Eigen 中，诸如算术运算符 `operator+` 本身不执行任何计算，它们只返回一个描述要执行的计算的“表达式对象”。实际计算发生在稍后，当整个表达式被计算时，通常在 `operator=`. 虽然这听起来很沉重，但任何现代优化编译器都能够优化掉这种抽象，结果是完美优化的代码。例如，当您执行以下操作时：

```
VectorXf a(50), b(50), c(50), d(50);
...
a = 3*b + 4*c + 5*d;
```

Eigen将其编译为一个 for 循环，因此数组只被遍历一次。简化（例如忽略 SIMD 优化），这个循环看起来像这样：

```
for (int i = 0; i < 50; ++i)
    a[i] = 3*b[i] + 4*c[i] + 5*d[i];
```

因此，您不应该害怕在Eigen中使用相对较大的算术表达式：它只会为Eigen 提供更多优化机会。

转置和共轭

转置 a^T, 共轭 $\bar{a}$，和伴随（即共轭转置） a^* 矩阵或向量的分别由成员函数 transpose()、conjugate() 和 adjoint() 获得。

例子：

```
MatrixXcf a = MatrixXcf::Random(2,2);

cout << "Here is the matrix a\n" << a << endl;

cout << "Here is the matrix a^T\n" << a.transpose() << endl;

cout << "Here is the conjugate of a\n" << a.conjugate() << endl;

cout << "Here is the matrix a^*\n" << a.adjoint() << endl;
```

输出：

```
Here is the matrix a
 (-0.211,0.68) (-0.605,0.823)
 (0.597,0.566)  (0.536,-0.33)
Here is the matrix a^T
 (-0.211,0.68)  (0.597,0.566)
(-0.605,0.823)  (0.536,-0.33)
Here is the conjugate of a
 (-0.211,-0.68) (-0.605,-0.823)
 (0.597,-0.566)    (0.536,0.33)
Here is the matrix a^*
 (-0.211,-0.68)  (0.597,-0.566)
(-0.605,-0.823)    (0.536,0.33)
```

对于实矩阵，`conjugate()`是无运算，因此`adjoint()`等价于`transpose()`。

至于基本的算术运算符，`transpose()`和`adjoint()`简单地返回一个代理对象没有做实际的换位。如果这样做`b = a.transpose()`，计算转置的同时将结果写入`b`。然而，这里有一个复杂的问题。**如果这样做`a = a.transpose()`，则Eigen`a`在转置评估完成之前开始将结果写入。**因此，正如人们所期望的那样，该指令`a = a.transpose()`不会替换`a`为它的转置：

例子：

```
Matrix2i a;
a << 1, 2, 3, 4;
cout << "Here is the matrix a:\n" << a << endl;

a = a.transpose(); // !!! 不要这样做 !!!
cout << "and the result of the aliasing effect:\n" << a << endl;
```

输出：

```
Here is the matrix a:
1 2
3 4
and the result of the aliasing effect:
1 2
3 4
```

这就是所谓的混叠问题。在“调试模式”下，即当断言没有被禁用时，这种常见的缺陷会被自动检测到。

对于*就地*转置，例如在 中`a = a.transpose()`，只需使用transposeInPlace()函数：

```
MatrixXf a(2,3);
a << 1, 2, 3, 4, 5, 6;
cout << "Here is the initial matrix a:\n" << a << endl;

a.transposeInPlace();
cout << "and after being transposed:\n" << a << endl;
```

输出：

```
Here is the initial matrix a:
1 2 3
4 5 6
and after being transposed:
1 4
2 5
3 6
```

还有用于复杂矩阵的 adjointInPlace() 函数。

矩阵-矩阵和矩阵-向量乘法

矩阵-矩阵乘法再次使用 `operator*`. 由于向量是矩阵的特例，它们也在那里被隐式处理，所以矩阵-向量乘积实际上只是矩阵-矩阵乘积的一个特例，向量-向量外积也是如此。因此，所有这些情况都由两个操作员处理：

- 二元运算符 * 如 `a*b`
- 复合运算符 *= 如 `a*=b`（在右侧乘法：`a*=b` 相当于 `a = a*b`）

例子：

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

using namespace Eigen;
int main()
{
    Matrix2d mat;
    mat << 1, 2, 3, 4;
    Vector2d u(-1,1), v(2,0);

    std::cout << "Here is mat*mat:\n" << mat * mat << std::endl;
    std::cout << "Here is mat*u:\n" << mat * u << std::endl;
    std::cout << "Here is u^T*mat:\n" << u.transpose() * mat << std::endl;
    std::cout << "Here is u^T*v:\n" << u.transpose() * v << std::endl;
    std::cout << "Here is u*v^T:\n" << u * v.transpose() << std::endl;
    std::cout << "Let's multiply mat by itself" << std::endl;

    mat = mat * mat;
    std::cout << "Now mat is mat:\n" << mat << std::endl;
}
```

输出：

```
Here is mat*mat:
 7 10
15 22
Here is mat*u:
1
1
Here is u^T*mat:
2 2
Here is u^T*v:
-2
Here is u*v^T:
```

```
-2 -0
 2  0
Let's multiply mat by itself
Now mat is mat:
 7 10
15 22
```

注意：如果您阅读了上面关于表达式模板的段落并且担心这样做`m = m * m`可能会导致混叠问题，请放心：**Eigen将矩阵乘法视为一种特殊情况并在此处注意引入一个临时值**，因此它将编译`m = m * m`为：

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

如果您知道您的矩阵乘积可以安全地计算到目标矩阵中而不会出现混叠问题，那么您可以使用 noalias() 函数来避免临时性，例如：

```
c.noalias() += a * b;
```

有关此主题的更多详细信息，请参阅有关别名的页面。

注意：对于担心性能的 BLAS 用户，诸如此类的表达式`c.noalias() -= 2 * a.adjoint() * b;`已完全优化并触发单个类似 gemm 的函数调用。

点积和叉积

对于点积和叉积，您需要 dot() 和 cross() 方 法。当然，点积也可以作为`u.adjoint() * v`得到1x1矩阵。

例子：

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

using namespace Eigen;
using namespace std;
int main()
{
    Vector3d v(1,2,3);
    Vector3d w(0,1,2);

    cout << "Dot product: " << v.dot(w) << endl;
    double dp = v.adjoint() * w; // automatic conversion of the inner
product to a scalar
    cout << "Dot product via a matrix product: " << dp << endl;
    cout << "Cross product:\n" << v.cross(w) << endl;
}
```

输出：

```
Dot product: 8
Dot product via a matrix product: 8
Cross product:
 1
-2
 1
```

请记住，**叉积仅适用于大小为 3 的向量。点积适用于任何大小的向量。**使用复数时，Eigen的点积在第一个变量中是共轭线性的，在第二个变量中是线性的。

基本算术归约运算

Eigen还提供了一些归约操作来将给定的矩阵或向量归约为单个值，例如总和（由sum()计算）、乘积（prod()）或最大值（maxCoeff()）和最小值（minCoeff()）的所有系数。

例子：

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

using namespace std;
int main()
{
    Eigen::Matrix2d mat;
    mat << 1, 2, 3, 4;
    cout << "Here is mat.sum():       " << mat.sum()       << endl;
    cout << "Here is mat.prod():      " << mat.prod()      << endl;
    cout << "Here is mat.mean():      " << mat.mean()      << endl;
    cout << "Here is mat.minCoeff():  " << mat.minCoeff()  << endl;
    cout << "Here is mat.maxCoeff():  " << mat.maxCoeff()  << endl;
    cout << "Here is mat.trace():     " << mat.trace()     << endl;
}
```

输出：

```
Here is mat.sum():       10
Here is mat.prod():      24
Here is mat.mean():      2.5
Here is mat.minCoeff():  1
Here is mat.maxCoeff():  4
Here is mat.trace():     5
```

所述*迹线*的矩阵，如由该 trace() 函数返回的，是对角线系数的和，也可以计算为有效地利用 `a.diagonal().sum()`，如我们将在后面上看到。

还存在通过参数返回相应系数坐标的`minCoeff`和`maxCoeff`函数的变体：

例子：

```
Matrix3f m = Matrix3f::Random();
std::ptrdiff_t i, j;
float minOfM = m.minCoeff(&i, &j);
cout << "Here is the matrix m:\n" << m << endl;
cout << "Its minimum coefficient (" << minOfM << ") is at position (" << i
<< "," << j << ")\n\n";

RowVector4i v = RowVector4i::Random();
int maxOfV = v.maxCoeff(&i);
cout << "Here is the vector v: " << v << endl;
cout << "Its maximum coefficient (" << maxOfV << ") is at position " << i <<
endl;
```

输出：

```
Here is the matrix m:
  0.68  0.597  -0.33
-0.211  0.823  0.536
 0.566 -0.605 -0.444
Its minimum coefficient (-0.605) is at position (2,1)

Here is the vector v:  1  0  3 -3
Its maximum coefficient (3) is at position 2
```

操作的有效性

Eigen检查您执行的操作的有效性。如果可能，它会在编译时检查它们，从而产生编译错误。这些错误消息可能又长又丑，但Eigen将重要消息写在 UPPERCASE_LETTERS_SO_IT_STANDS_OUT 中。例如：

```
Matrix3f m;
Vector4f v;
v = m * v;       // 编译时错误：  YOU_MIXED_MATRICES_OF_DIFFERENT_SIZES
```

当然，在很多情况下，例如在检查动态大小时，是无法在编译时进行检查的。Eigen然后使用运行时断言。这意味着如果程序在“调试模式”下运行，程序将在执行非法操作时中止并显示错误消息，如果断言关闭，它可能会崩溃。

```
MatrixXf m(3,3);
VectorXf v(4);
v = m * v;  // 此处运行时断言失败：“无效矩阵乘积”
```

有关此主题的更多详细信息，请参阅此页面。