线性代数和分解

密集线性问题和分解

本页解释了如何求解线性系统，计算各种分解，如 LU、QR、SVD、特征分解...阅读本页后，不要错过我们的密集矩阵分解目录。

基本线性求解

问题：你有一个方程组，你已经写成一个矩阵方程

$$Ax = b \tag{1}$$

其中 *A* 和 *b* 是矩阵（*b*可以是向量，作为特殊情况）。你想找到一个解决方案 *x*。

解决方案：可将各种分解之间进行选择，这取决于你的矩阵的性质一取决于你是否赞成速度或准确性和。但是，让我们从一个适用于所有情况的示例开始，这是一个很好的折衷方案：

例子：

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

using namespace std;
using namespace Eigen;

int main()
{
    Matrix3f A;
    Vector3f b;
    A << 1, 2, 3,
         4, 5, 6,
         7, 8, 10;
    b << 3, 3, 4;
    cout << "Here is the matrix A:\n" << A << endl;
    cout << "Here is the vector b:\n" << b << endl;
    Vector3f x = A.colPivHouseholderQr().solve(b);
    cout << "The solution is:\n" << x << endl;
}
```

输出：

```
Here is the matrix A:
 1  2  3
 4  5  6
 7  8 10
Here is the vector b:
3
3
4
The solution is:
-2
 1
 1
```

在此示例中， colPivHouseholderQr() 方法返回类ColPivHouseholderQR的对象。由于这里的矩阵是 Matrix3f 类型，因此该行可以替换为：

```
ColPivHouseholderQR<Matrix3f> dec(A);
Vector3f x = dec.solve(b);
```

这里，ColPivHouseholderQR是一个带有列旋转的 QR 分解。这是本教程的一个很好的折衷方案，因为它适用于所有矩阵，而且速度非常快。这是您可以选择的其他一些分解表，具体取决于您的矩阵、您尝试解决的问题以及您想要进行的权衡：

分解	方法	对矩阵的要求	速度（中小型）	速度（大）	准确性
PartialPivLU	partialPivLu()	可逆的	++	++	+
FullPivLU	fullPivLu()	None	-	--	+++
HouseholderQR	householderQr()	None	++	++	+
ColPivHouseholderQR	colPivHouseholderQr()	None	+	-	+++
FullPivHouseholderQR	fullPivHouseholderQr()	None	-	--	+++
completeOrthogonalDecomposition	completeOrthogonalDecomposition()	None	+	-	+++
LLT	llt()	正定	+++	+++	+
LDLT	ldlt()	正半定或负半定	+++	+	++
BDCSVD	bdcSvd()	None	-	--	+++
雅可比SVD	jacobiSvd()	None	-	---	+++

要大致了解不同分解的真实相对速度，请查看此基准测试。

所有这些分解都提供了一个 solve() 方法，它的工作原理与上面的例子一样。

如果您更了解矩阵的属性，则可以使用上表来选择最佳方法。例如，**使用全秩非对称矩阵求解线性系统的一个不错的选择是PartialPivLU。 如果你知道你的矩阵也是对称的和正定的，上表说一个很好的选择是LLT或LDLT分解。**这是一个例子，也证明了使用通用矩阵（而不是向量）作为右手边是可能的：

例子：

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

```

```cpp
using namespace std;
using namespace Eigen;

int main()
{
    Matrix2f A, b;
    A << 2, -1, -1, 3;
    b << 1, 2, 3, 1;
    cout << "Here is the matrix A:\n" << A << endl;
    cout << "Here is the right hand side b:\n" << b << endl;
    Matrix2f x = A.ldlt().solve(b);
    cout << "The solution is:\n" << x << endl;
}
```

输出：

```
Here is the matrix A:
 2 -1
-1  3
Here is the right hand side b:
1 2
3 1
The solution is:
1.2 1.4
1.4 0.8
```

对于一个更完整的表格比较受支持的所有分解Eigen（注意，Eigen支持许多其他的分解），请参阅我们的特殊页面这个话题。

最小二乘求解

在最小二乘意义上求解欠定或超定线性系统的最通用和最准确的方法是 SVD 分解。Eigen提供了两种实现。推荐的一个是BDCSVD类，它可以很好地解决大问题，并自动回退到 JacobiSVD类来处理较小的问题。对于这两个类，它们的 solve() 方法在最小二乘意义上求解线性系统。

下面是一个例子：

例子：

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

using namespace std;
using namespace Eigen;

int main()
{
    MatrixXf A = MatrixXf::Random(3, 2);
    cout << "Here is the matrix A:\n" << A << endl;
    VectorXf b = VectorXf::Random(3);
    cout << "Here is the right hand side b:\n" << b << endl;
    cout << "The least-squares solution is:\n"
        << A.bdcSvd(ComputeThinU | ComputeThinV).solve(b) << endl;
}
```

输出：

```
Here is the matrix A:
  0.68  0.597
-0.211  0.823
 0.566 -0.605
Here is the right hand side b:
 -0.33
 0.536
-0.444
The least-squares solution is:
-0.67
0.314
```

SVD 的另一种替代方法是CompleteOrthogonalDecomposition，它通常更快且准确。

同样，如果您对问题有更多的了解，上表包含可能更快的方法。如果您的矩阵是满秩的，则 HouseHolderQR 是首选方法。如果您的矩阵是满秩且条件良好的，则对正规方程的矩阵使用 Cholesky 分解 (LLT) 会更快。我们关于最小二乘求解的页面有更多细节。

检查矩阵是否奇异

只有您知道您希望解决方案被认为有效的误差幅度。因此，如果您愿意，Eigen允许您自己进行计算，如下例所示：

例子：

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

using namespace std;
using namespace Eigen;

int main()
{
    MatrixXd A = MatrixXd::Random(100,100);
    MatrixXd b = MatrixXd::Random(100,50);
    MatrixXd x = A.fullPivLu().solve(b);
    double relative_error = (A * x - b).norm() / b.norm(); // norm() is L2 norm
    cout << "The relative error is:\n" << relative_error << endl;
}
```

输出：

```
The relative error is:
2.31495e-14
```

计算特征值和特征向量

您需要在此处进行特征分解，请参阅此页面上的可用此类分解。确保检查您的矩阵是否自伴随，这在这些问题中经常发生。这是一个使用SelfAdjointEigenSolver的例子，它可以很容易地适应使用EigenSolver或ComplexEigenSolver 的一般矩阵。

特征值和特征向量的计算不一定会收敛，但这种不收敛的情况非常少见。调用 info() 是为了检查这种可能性。

例子：

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

using namespace std;
using namespace Eigen;

int main()
{
    Matrix2f A;
    A << 1, 2, 2, 3;
    cout << "Here is the matrix A:\n" << A << endl;
    SelfAdjointEigenSolver<Matrix2f> eigensolver(A);
    if (eigensolver.info() != Success)
        abort();
    cout << "The eigenvalues of A are:\n" << eigensolver.eigenvalues() << endl;
    cout << "Here's a matrix whose columns are eigenvectors of A \n"
        << "corresponding to these eigenvalues:\n"
        << eigensolver.eigenvectors() << endl;
}
```

输出：

```
Here is the matrix A:
1 2
2 3
The eigenvalues of A are:
-0.236
  4.24
Here's a matrix whose columns are eigenvectors of A
corresponding to these eigenvalues:
-0.851 -0.526
 0.526 -0.851
```

计算逆和行列式

首先，确保你真的想要这个。虽然逆和行列式是基本的数学概念，但在*数值*线性代数中它们不如在纯数学中有用。逆计算通常有利地被 solve() 操作取代，并且行列式通常*不是*检查矩阵是否可逆的好方法。

然而，对于*非常小的*矩阵，上述可能不正确，逆和行列式可能非常有用。

虽然某些分解（例如PartialPivLU和FullPivLU）提供inverse()和 determinant() 方法，但您也可以直接在矩阵上调用inverse()和 determinant() 。如果您的矩阵具有非常小的固定大小（最多 4x4），这将允许Eigen避免执行 LU 分解，而是使用在此类小矩阵上更有效的公式。

下面是一个例子：

例子：

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

using namespace std;
using namespace Eigen;
```

```cpp

int main()
{
    Matrix3f A;
    A << 1, 2, 1,
    2, 1, 0,
    -1, 1, 2;
    cout << "Here is the matrix A:\n" << A << endl;
    cout << "The determinant of A is " << A.determinant() << endl;
    cout << "The inverse of A is:\n" << A.inverse() << endl;
}
```

输出：

```
Here is the matrix A:
 1  2  1
 2  1  0
-1  1  2
The determinant of A is -3
The inverse of A is:
-0.667      1  0.333
  1.33     -1 -0.667
    -1      1      1
```

将计算与构造分离

在上面的例子中，分解是在构造分解对象的同时计算的。然而，在某些情况下，您可能希望将这两件事分开，例如，如果您在构建时不知道要分解的矩阵；或者如果您想重用现有的分解对象。

使这成为可能的是：

- 所有分解都有一个默认构造函数，
- 所有分解都有一个计算（矩阵）方法来进行计算，并且可以在已经计算的分解上再次调用，重新初始化它。

例如：

例子：

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

using namespace std;
using namespace Eigen;

int main()
{
    Matrix2f A, b;
    LLT<Matrix2f> llt;
    A << 2, -1, -1, 3;
    b << 1, 2, 3, 1;
    cout << "Here is the matrix A:\n" << A << endl;
    cout << "Here is the right hand side b:\n" << b << endl;
    cout << "Computing LLT decomposition..." << endl;
    llt.compute(A);
    cout << "The solution is:\n" << llt.solve(b) << endl;
```

```
    A(1,1)++;
    cout << "The matrix A is now:\n" << A << endl;
    cout << "Computing LLT decomposition..." << endl;
    llt.compute(A);
    cout << "The solution is now:\n" << llt.solve(b) << endl;
}
```

输出：

```
Here is the matrix A:
 2 -1
-1  3
Here is the right hand side b:
1 2
3 1
Computing LLT decomposition...
The solution is:
1.2 1.4
1.4 0.8
The matrix A is now:
 2 -1
-1  4
Computing LLT decomposition...
The solution is now:
    1  1.29
    1 0.571
```

最后，您可以告诉分解构造函数为分解给定大小的矩阵预先分配存储空间，这样当您随后分解此类矩阵时，不会执行动态内存分配（当然，如果您使用的是固定大小的矩阵，则没有动态内存分配发生）。这是通过将大小传递给分解构造函数来完成的，如下例所示：

HouseholderQR qr(50,50);

MatrixXf A = MatrixXf::Random (50,50);

qr.compute(A); // 没有动态内存分配

秩揭示分解

某些分解是秩揭示的，即能够计算矩阵的秩。这些通常也是在面对非满秩矩阵（在方形情况下意味着奇异矩阵）时表现最佳的分解。在这张表上，您可以看到我们所有的分解是否显示等级。

秩揭示分解至少提供了一个 rank() 方法。它们还可以提供诸如 isInvertible() 之类的便捷方法，有些还提供计算矩阵的内核（零空间）和图像（列空间）的方法，就像FullPivLU的情况：

例子：

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

using namespace std;
using namespace Eigen;

int main()
{
    Matrix3f A;
    A << 1, 2, 5,
```

```
        2, 1, 4,
        3, 0, 3;
    cout << "Here is the matrix A:\n" << A << endl;
    FullPivLU<Matrix3f> lu_decomp(A);
    cout << "The rank of A is " << lu_decomp.rank() << endl;
    cout << "Here is a matrix whose columns form a basis of the null-space
of A:\n"
        << lu_decomp.kernel() << endl;
    cout << "Here is a matrix whose columns form a basis of the column-space
of A:\n"
        << lu_decomp.image(A) << endl; // yes, have to pass the original A
}
```

输出：

```
Here is the matrix A:
1 2 5
2 1 4
3 0 3
The rank of A is 2
Here is a matrix whose columns form a basis of the null-space of A:
 0.5
   1
-0.5
Here is a matrix whose columns form a basis of the column-space of A:
5 1
4 2
3 3
```

当然，任何秩计算都取决于任意阈值的选择，因为实际上没有浮点矩阵是*完全*秩亏的。Eigen选择一个合理的默认阈值，这取决于分解，但通常是对角线大小乘以机器 epsilon。虽然这是我们可以选择的最佳默认值，但只有您知道应用程序的正确阈值是多少。在调用 rank() 或任何其他需要使用此类阈值的方法之前，您可以通过在分解对象上调用 setThreshold() 来设置它。分解本身，即compute() 方法，与阈值无关。更改阈值后，您无需重新计算分解。

例子：

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

using namespace std;
using namespace Eigen;

int main()
{
    Matrix2d A;
    A << 2, 1,
    2, 0.9999999999;
    FullPivLU<Matrix2d> lu(A);
    cout << "By default, the rank of A is found to be " << lu.rank() <<
endl;
    lu.setThreshold(1e-5);
    cout << "With threshold 1e-5, the rank of A is found to be " <<
lu.rank() << endl;
}
```

输出：

```
By default, the rank of A is found to be 2
With threshold 1e-5, the rank of A is found to be 1
```