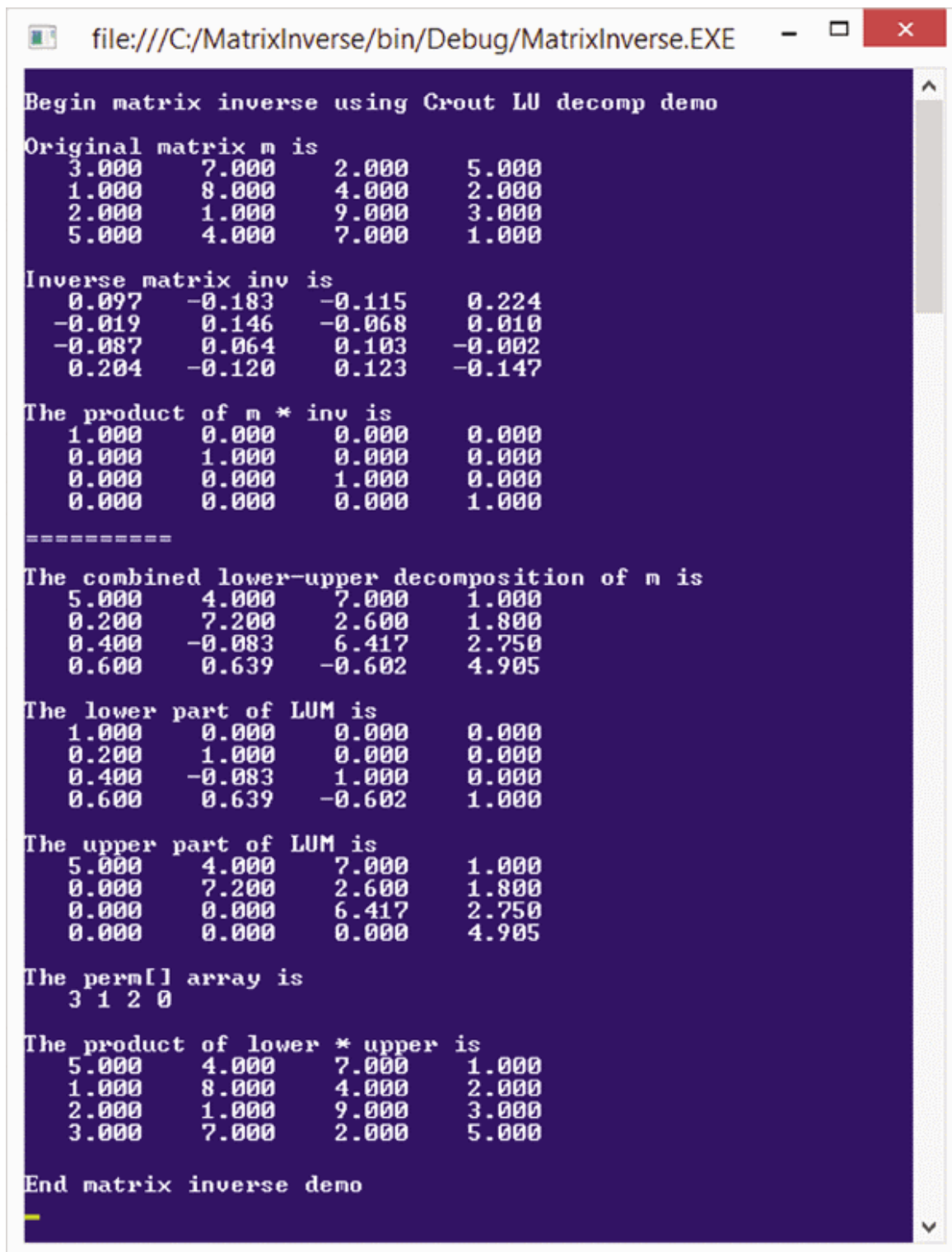


# Matrix Inversion Using C#

One of the most fundamental techniques in machine learning (ML) software systems is matrix inversion. For reasons that aren't clear to me, the Microsoft .NET Framework doesn't seem to have a matrix inversion method (or if there is such a method, it's very well hidden). In this article I present and explain the code for a matrix inversion method that uses an algorithm called Crout's LU decomposition.

Let me be the first to admit that matrix inversion isn't a very flashy topic. But if you want to create ML systems without relying on external libraries, having a matrix inversion method is essential because matrix inversion is used by dozens of important ML algorithms.

A good way to see where this article is headed is to take a look at the demo program in Figure 1.



```
file:///C:/MatrixInverse/bin/Debug/MatrixInverse.EXE

Begin matrix inverse using Crout LU decomp demo

Original matrix m is
  3.000  7.000  2.000  5.000
  1.000  8.000  4.000  2.000
  2.000  1.000  9.000  3.000
  5.000  4.000  7.000  1.000

Inverse matrix inv is
  0.097 -0.183 -0.115  0.224
 -0.019  0.146 -0.068  0.010
 -0.087  0.064  0.103 -0.002
  0.204 -0.120  0.123 -0.147

The product of m * inv is
  1.000  0.000  0.000  0.000
  0.000  1.000  0.000  0.000
  0.000  0.000  1.000  0.000
  0.000  0.000  0.000  1.000

=====

The combined lower-upper decomposition of m is
  5.000  4.000  7.000  1.000
  0.200  7.200  2.600  1.800
  0.400 -0.083  6.417  2.750
  0.600  0.639 -0.602  4.905

The lower part of LUM is
  1.000  0.000  0.000  0.000
  0.200  1.000  0.000  0.000
  0.400 -0.083  1.000  0.000
  0.600  0.639 -0.602  1.000

The upper part of LUM is
  5.000  4.000  7.000  1.000
  0.000  7.200  2.600  1.800
  0.000  0.000  6.417  2.750
  0.000  0.000  0.000  4.905

The perm[] array is
  3 1 2 0

The product of lower * upper is
  5.000  4.000  7.000  1.000
  1.000  8.000  4.000  2.000
  2.000  1.000  9.000  3.000
  3.000  7.000  2.000  5.000

End matrix inverse demo
```

## Figure 1 Matrix Inversion Demo

The demo begins by setting up and displaying a 4x4 (4 rows, 4 columns) matrix m:

```
3.0  7.0  2.0  5.0
1.0  8.0  4.0  2.0
2.0  1.0  9.0  3.0
5.0  4.0  7.0  1.0
```

It then calculates the inverse of the matrix using a program-defined method and displays the result:

```
0.097  -0.183  -0.115  0.224
-0.019   0.146  -0.068   0.010
-0.087   0.064   0.103  -0.002
 0.204  -0.120   0.123  -0.147
```

That's pretty much it. But as you'll see shortly, matrix inversion is surprisingly tricky. Next, the demo verifies that the calculated inverse is correct by multiplying the original matrix times the inverse:

```
1.0  0.0  0.0  0.0
0.0  1.0  0.0  0.0
0.0  0.0  1.0  0.0
0.0  0.0  0.0  1.0
```

The result of the multiplication is the identity matrix (1.0 values on the diagonal, 0.0 values elsewhere) indicating the inverse result is correct.

Behind the scenes, the matrix inversion method uses a technique called matrix decomposition. Decomposition factors a matrix into two matrices, called L (lower) and U (upper), that when multiplied together give the original matrix, but with some of the rows rearranged. The demo displays the decomposition:

```
5.000  4.000  7.000  1.000
0.200  7.200  2.600  1.800
0.400 -0.083  6.417  2.750
0.600  0.639 -0.602  4.905
```

The decomposition actually contains both the L and U matrices. The L matrix consists of the values in the lower-left part of the combined LU matrix, with dummy 1.0 values on the diagonal and 0.0 values in the upper part:

```
1.000  0.000  0.000  0.000
0.200  1.000  0.000  0.000
0.400 -0.083  1.000  0.000
0.600  0.639 -0.602  1.000
```

The U matrix consists of the values in the upper-right part and the diagonal of the combined LU matrix:

5.000	4.000	7.000	1.000
0.000	7.200	2.600	1.800
0.000	0.000	6.417	2.750
0.000	0.000	0.000	4.905

The demo verifies that the LU decomposition is correct by multiplying the L and U matrices and displaying the result:

5.0	4.0	7.0	1.0
1.0	8.0	4.0	2.0
2.0	1.0	9.0	3.0
3.0	7.0	2.0	5.0

If you compare  $L*U$  with the original matrix  $m$ , you'll see that  $L*U$  is almost the same as  $m$ , but the rows of  $L*U$  have been permuted (rearranged). The row permutation information is:

3	1	2	0
---	---	---	---

This means row[0] of  $m$  is at row[3] of  $L*U$ ; row[1] of  $m$  is at row[1] of  $L*U$ ; row[2] of  $m$  is at row[2] of  $L*U$ ; and row[3] of  $m$  is at row[0] of  $L*U$ .

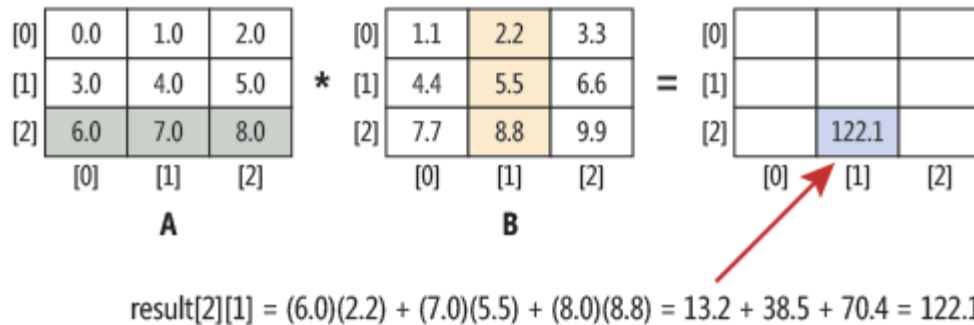
## Understanding Matrix Inversion

In normal arithmetic, the inverse of a number  $z$  is a number that when multiplied by  $z$  gives 1. For example, if  $z = 3$ , the inverse of  $z$  is  $1/3 = 0.33$  because  $3 * (1/3) = 1$ .

Matrix inversion extends this idea. The inverse of an  $n \times n$  (called a “square matrix” because the number of rows equals the number of columns) matrix  $m$  is a matrix  $m_i$  such that  $m * m_i = I$  where  $I$  is the identity matrix (1.0s on the diagonal, 0.0s elsewhere).

Not all matrices have an inverse. As it turns out, there is a scalar value called the determinant of a matrix. If the determinant of a matrix is zero, then the matrix doesn't have an inverse.

Note that to fully understand matrix inversion, you must understand matrix multiplication. Matrix multiplication is best explained by example. Take a look at the example in **Figure 2**. The value at cell  $[r][c]$  of the result matrix is the product of the values in row  $r$  of the first matrix and the values in column  $c$  of the second matrix.



## Figure 2 Matrix Multiplication

When finding the inverse of a matrix, you work only with square matrices, but matrix multiplication can be applied to matrices with different shapes. In these situations the matrices must be what's called conformable. If matrix A has shape  $axn$  and matrix B has shape  $nxb$ , the result of multiplication has shape  $axb$ . The number of columns in the first matrix must equal the number of rows in the second matrix.

The demo program implements matrix multiplication with method `MatrixProduct` and helper method `MatrixCreate`, as shown in **Figure 3**. The demo uses a brute force approach, but because the calculation of each cell in the result matrix is independent, matrix multiplication could be performed in parallel using the `Parallel.For` method from the .NET Task Parallel Library.

## Figure 3 Matrix Multiplication

```
static double[][] MatrixCreate(int rows, int cols)
{
    double[][] result = new double[rows][];
    for (int i = 0; i < rows; ++i)
        result[i] = new double[cols];
    return result;
}

static double[][] MatrixProduct(double[][] matrixA,
    double[][] matrixB)
{
    int aRows = matrixA.Length;
    int aCols = matrixA[0].Length;
    int bRows = matrixB.Length;
    int bCols = matrixB[0].Length;
    if (aCols != bRows)
        throw new Exception("Non-conformable matrices");
    double[][] result = MatrixCreate(aRows, bCols);
    for (int i = 0; i < aRows; ++i)
        for (int j = 0; j < bCols; ++j)
            for (int k = 0; k < aCols; ++k)
                result[i][j] += matrixA[i][k] *
                    matrixB[k][j];
}
```

```
    return result;
}
```

## The Demo Program

I coded the demo program using C#, but you should have no difficulty porting the code to another language, such as Visual Basic or Python, if you wish. The demo code is too long to present in its entirety, but the complete code is available in the download that accompanies this article. The code is also available at [quaetrix.com/Matrix/code.html](http://quaetrix.com/Matrix/code.html) (the URL is case-sensitive).

To create the demo program, I launched Visual Studio and created a new C# console application named MatrixInverse. The demo program has no significant .NET Framework dependencies so any version of Visual Studio will work. After the template code loaded, in the Solution Explorer window I right-clicked on file Program.cs and renamed it to the more descriptive MatrixInverseProgram.cs and Visual Studio then automatically renamed class Program for me.

At the top of the editor window I deleted all using statements that referenced unnecessary namespaces, leaving just the one reference to the top-level System namespace.

The Main method begins by setting up a matrix to invert:

```
Console.WriteLine("Begin matrix inverse demo");
double[][] m = MatrixCreate(4, 4);
m[0][0] = 3; m[0][1] = 7; m[0][2] = 2; m[0][3] = 5;
m[1][0] = 1; m[1][1] = 8; m[1][2] = 4; m[1][3] = 2;
m[2][0] = 2; m[2][1] = 1; m[2][2] = 9; m[2][3] = 3;
m[3][0] = 5; m[3][1] = 4; m[3][2] = 7; m[3][3] = 1;
```

In many cases, your source data will be stored in a text file so you'll have to write a helper method to load a matrix from the file. The demo uses an array-of-arrays-style matrix. Unlike most programming languages, C# supports a true n-dimensional matrix type, but I prefer using the standard array-of-arrays approach.

Next, Main displays the matrix m, and then computes and displays the inverse:

```
Console.WriteLine("Original matrix m is ");
Console.WriteLine(MatrixAsString(m));
double[][] inv = MatrixInverse(m);
Console.WriteLine("Inverse matrix inv is ");
Console.WriteLine(MatrixAsString(inv));
```

All the work is performed by method `MatrixInverse`. Helper method `MatrixAsString` returns a string representation of a matrix:

```
static string MatrixAsString(double[][] matrix)
{
    string s = "";
    for (int i = 0; i < matrix.Length; ++i) {
        for (int j = 0; j < matrix[i].Length; ++j)
            s += matrix[i][j].ToString("F3").PadLeft(8) + " ";
        s += Environment.NewLine;
    }
    return s;
}
```

Here the number of decimals (3) and value width (8) are hardcoded for simplicity. A more general approach would pass those values as input parameters. Next, the demo multiplies the original matrix and the inverse matrix in order to verify that the result is the identity matrix:

```
double[][] prod = MatrixProduct(m, inv);
Console.WriteLine("The product of m * inv is ");
Console.WriteLine(MatrixAsString(prod));
```

In this version, you have to visually verify that the result is the identity matrix. A more sophisticated approach would be to write a method that accepts a matrix and returns true if the matrix is an identity matrix, subject to some small difference ( $1.0e-5$  is typical) in cell values.

Next, the demo illustrates some of the behind-the-scenes work by decomposing the original matrix:

```
double[][] lum;
int[] perm;
int toggle = MatrixDecompose(m, out lum, out perm);
Console.WriteLine("The decomposition is");
Console.WriteLine(MatrixAsString(lum));
```

The calling signature of method `MatrixDecompose` might appear a bit unusual to you. The explicit return value is either +1 or -1 depending on the number of row permutations there were (even or odd, respectively). The toggle return value isn't used by the demo, but is needed if you want to compute the determinant of the matrix, which tells you if the inverse of a matrix exists, as I'll explain shortly.

The lum out parameter is the combined LU (lower-upper) decomposition. The perm out parameter is an array of integer values that encode how the rows have been permuted.

Next, the demo extracts the lower and upper matrices from the combined LU matrix and displays them:

```
double[][] lower = ExtractLower(lum);
double[][] upper = ExtractUpper(lum);
Console.WriteLine("The lower part of LUM is");
Console.WriteLine(MatrixAsString(lower));
Console.WriteLine("The upper part of LUM is");
Console.WriteLine(MatrixAsString(upper));
```

Helper methods ExtractLower and ExtractUpper really aren't needed to perform matrix inversion, but are used to illustrate how matrix decomposition works.

The demo program concludes by displaying the row permutation information, multiplying the lower and upper decomposition matrices, and displaying the result:

```
Console.WriteLine("The perm[] array is");
ShowVector(perm);
double[][] lowTimesUp = MatrixProduct(lower, upper);
Console.WriteLine("The product of lower * upper is ");
Console.WriteLine(MatrixAsString(lowTimesUp));
Console.WriteLine("End matrix inverse demo");
```

Program-defined helper method ShowVector is just a convenience to keep the Main method clean. As noted, the result of lower \* upper is the original matrix, except that the rows of the result are permuted according to the information in the perm array.

## Method MatrixInverse

The definition of method MatrixInverse begins with:

```
static double[][] MatrixInverse(double[][] matrix)
{
    int n = matrix.Length;
    double[][] result = MatrixCreate(n, n);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            result[i][j] = matrix[i][j];
    ...
}
```



The method assumes that its input parameter does, in fact, have a matrix. This means you should check before calling, along the lines of:

```
int d = MatrixDeterminant(m);
if (d == 0.0)
    Console.WriteLine("No inverse");
else
    double[][] mi = MatrixInverse(m);
```

An alternative is to place this error-checking code inside method `MatrixInverse`, which creates a copy of the input matrix. You could also perform matrix inversion in place, which saves memory but destroys the original matrix.

Next, `MatrixInverse` decomposes the copy of the input matrix:

```
double[][] lum; // Combined lower & upper
int[] perm;
int toggle;
toggle = MatrixDecompose(matrix, out lum, out perm);
```

It may seem strange to go to all the trouble of decomposing a matrix in order to compute its inverse, but trust me, this approach is much easier than inverting a matrix directly.

Next, the demo computes the inverse using yet another helper method named `Helper`:

```
double[] b = new double[n];
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j)
        if (i == perm[j]) b[j] = 1.0;
        else b[j] = 0.0;
    double[] x = Helper(lum, b); //
    for (int j = 0; j < n; ++j)
        result[j][i] = x[j];
}
```

This code is very subtle. Fortunately, you won't ever have to modify this part of the code.

Method `MatrixInverse` concludes by returning the inverse:

```
...
return result;
}
```

It should be clear that method `MatrixInverse` is essentially a wrapper around methods `MatrixDecompose` and `Helper`, which do most of the work. The code for those two methods are in the download that accompanies this article.

## The Determinant of a Matrix

If the determinant of a matrix is zero, then the matrix doesn't have an inverse. Suppose a 3x3 matrix is:

```
1.0  4.0  0.0
3.0  2.0  5.0
7.0  8.0  6.0
```

The determinant of the matrix is:

```
+1.0 * [(2.0)(6.0) - (5.0)(8.0)]
-4.0 * [(3.0)(6.0) - (5.0)(7.0)]
+0.0 * [(3.0)(8.0) - (2.0)(7.0)]
= +1.0 * (-28.0) -4.0 * (-17.0) = -28.0 + 68.0 = 40.0
```

Every square matrix has a determinant. For matrices with shapes larger than 3x3, calculating the determinant is surprisingly difficult. However, as it turns out, if you decompose a matrix you can use the combined lower-upper result to calculate the determinant rather easily by multiplying the diagonal elements of the result. The demo program defines a method `MatrixDeterminant` as:

```
static double MatrixDeterminant(double[][] matrix)
{
    double[][] lum;
    int[] perm;
    int toggle = MatrixDecompose(matrix, out lum,
        out perm);
    double result = toggle;
    for (int i = 0; i < lum.Length; ++i)
        result *= lum[i][i];
    return result;
}
```

## Wrapping Up

The key to efficient matrix inversion is matrix decomposition. There are several algorithms that decompose a matrix. The demo code uses a technique called Crout's algorithm. A common alternative is Doolittle's algorithm. I used to prefer Doolittle's

algorithm because it's a bit simpler than Crout's, but I now favor Crout's algorithm because it has fewer places to fail.