



---

# **Advanced Matrix Manipulation with NMath Matrix**

**Technical Report No. 3**

**CenterSpace Software  
Corvallis, Oregon**

---



---

## Introduction

**NMath Core** is part of CenterSpace Software's **NMath™** product suite, which provides object-oriented components for mathematical, engineering, scientific, and financial applications on the .NET platform. **NMath Core** extends the foundational library **NMath Core** to include structured sparse matrix classes and factorizations, general matrix decompositions, advanced least squares solutions, and solutions to eigenvalue problems.

Like **NMath Core**, **NMath Core** uses machine-specific, highly-optimized versions of the public domain computing packages known as the BLAS (Basic Linear Algebra Subroutines) and LAPACK (Linear Algebra PACKage) for most computations. This gives **NMath Core** classes performance levels comparable to C or Fortran.

Fully compliant with the Microsoft Common Language Specification, all **NMath Core** routines are callable from any .NET language, including C#, Visual Basic.NET, and Managed C++.

**NOTE—Code samples in this document are shown in C#. Complete NMath Core code examples in both C# and Visual Basic.NET are available on the CenterSpace website:**

<http://www.centerspace.net/examples/NMath/Matrix/>

## Features

The features of **NMath Core** include:

- Full-featured structured sparse matrix classes, including triangular, symmetric, Hermitian, banded, tridiagonal, symmetric banded, and Hermitian banded.
- Support for four datatypes: single- and double-precision floating point numbers, and single- and double-precision complex numbers.
- Functions for converting between general matrices and structured sparse matrix types.
- Functions for transposing structured sparse matrices, computing inner products, and calculating matrix norms.

- Overloaded arithmetic operators with their conventional meanings for those .NET languages that support them, and equivalent named methods (`Add()`, `Subtract()`, and so on) for those that do not.
- Classes for factoring structured sparse matrices, including LU factorization for banded and tridiagonal matrices, Bunch-Kaufman factorization for symmetric and Hermitian matrices, and Cholesky decomposition for symmetric and Hermitian positive definite matrices. Once constructed, matrix factorizations can be used to solve linear systems and compute determinants, inverses, and condition numbers.
- Orthogonal decomposition classes for general matrices, including QR decomposition and singular value decomposition (SVD).
- Advanced least squares factorization classes for general matrices, including Cholesky, QR, and SVD.
- Classes for solving symmetric, Hermitian, and nonsymmetric eigenvalue problems.
- Fully persistable data classes using standard .NET mechanisms.

## Design

**NMath Core** is built on **NMath Core**, the foundational library in the **NMath** product suite. **NMath Core** includes general vector and matrix classes, complex number classes, and random number generators.<sup>1</sup> **NMath Core** extends the general matrix classes of **NMath Core** to include structured sparse matrix classes and factorizations, general matrix decompositions, and advanced least squares solutions.

### Structured Sparse Matrix Classes

**NMath Core** provides a wide variety of structured sparse matrix types. A *sparse matrix* is a matrix with only a small number of nonzero elements. A *structured sparse matrix* is one in which the zero elements (or elements contributing no new information) are distributed according to some pattern. By exploiting this pattern, structured sparse matrices can be manipulated more efficiently than general matrices, since all of the elements do not need to be stored.

---

<sup>1</sup> For more information on **NMath Core**, see *.NET Numerical Applications with NMath Core* (Technical Report No. 1, CenterSpace Software).

NMath Core includes classes for representing:

- **Triangular Matrices**

A *lower triangular* matrix is a square matrix with all elements above the main diagonal equal to zero. An *upper triangular* matrix is a square matrix with all elements below the main diagonal equal to zero. For efficiency, only the lower or upper triangle, respectively, is stored. Triangular matrices often arise at an intermediate stage in solving systems of equations and inverting matrices.

- **Symmetric and Hermitian Matrices**

A *symmetric* matrix is a square matrix that satisfies  $A = A^T$  where  $A^T$  denotes the transpose of  $A$ . That is,  $a_{ij} = a_{ji}$  for all  $i, j$ . In a *Hermitian* matrix,  $a_{ij} = \overline{a_{ji}}$  for all  $i, j$ , where  $\bar{z}$  denotes the complex conjugate. A symmetric matrix is thus a special case of a Hermitian matrix where all the elements are real. For efficiency, only the upper triangle is stored. Symmetric and Hermitian matrices are often used to represent quadratic forms.

- **Banded Matrices**

A *banded* matrix is a matrix that has all its non-zero entries near the diagonal. Entries farther above the diagonal than the *upper bandwidth*, or farther below the diagonal than the *lower bandwidth*, are defined to be zero. For efficiency, zero elements outside the bandwidth are not stored.

- **Symmetric Banded and Hermitian Banded Matrices**

A *symmetric banded* matrix is a symmetric matrix that has all its non-zero entries near the diagonal. Entries farther away from the diagonal than the *half bandwidth* are defined to be zero. *Hermitian banded* matrices are a generalization of symmetric banded matrices for complex types.

- **Tridiagonal Matrices**

A *tridiagonal* matrix is a matrix which has all its non-zero entries on the main diagonal, the superdiagonal, and the subdiagonal. For efficiency, zero elements outside the main diagonal, superdiagonal, and subdiagonal are not stored. Tridiagonal matrices often occur in one-dimensional problems and at an intermediate stage in the process of finding eigenvalues.

For example, this code creates a tridiagonal matrix of single precision complex numbers using the **FloatComplexTriDiagMatrix** class:

```
int rows = 8; cols = 8;
FloatComplexTriDiagMatrix A =
    new FloatComplexTriDiagMatrix( rows, cols );
```

You can quickly set values on the main diagonal, superdiagonal, and subdiagonal of a tridiagonal matrix using the `Diagonal()` method:

```
A.Diagonal( -1 ).Set( Slice.All, 1 );
A.Diagonal( 0 ).Set( Slice.All, 2 );
A.Diagonal( 1 ).Set( Slice.All, 3 );

Console.WriteLine( "A = {0}", A.ToString() );

// A = 8x8 [ (2,0) (3,0) (0,0) (0,0) (0,0) (0,0) (0,0) (0,0)
//          (1,0) (2,0) (3,0) (0,0) (0,0) (0,0) (0,0) (0,0)
//          (0,0) (1,0) (2,0) (3,0) (0,0) (0,0) (0,0) (0,0)
//          (0,0) (0,0) (1,0) (2,0) (3,0) (0,0) (0,0) (0,0)
//          (0,0) (0,0) (0,0) (1,0) (2,0) (3,0) (0,0) (0,0)
//          (0,0) (0,0) (0,0) (0,0) (1,0) (2,0) (3,0) (0,0)
//          (0,0) (0,0) (0,0) (0,0) (0,0) (1,0) (2,0) (3,0)
//          (0,0) (0,0) (0,0) (0,0) (0,0) (0,0) (1,0) (2,0) ]
```

The indexer works just like it does for general matrices:

```
FloatComplex c = A[7,0];
```

You can also set the values of elements on the main, super- and sub-diagonals of a tridiagonal matrix using the indexer:

```
A[2,1] = new FloatComplex( 2, -1 );
```

However, attempting to set an element that would destroy the tridiagonal structure of the matrix raises a **NonModifiableElementException**:

```
try
{
    A[7,0] = new FloatComplex( 1, 1 );
}
catch( NonModifiableElementException e )
{
    // Do something here
}
```

**NMath Core** provides overloaded arithmetic operators for structured sparse matrices with their conventional meanings for those .NET languages that support them, and equivalent named methods for those that do not. All binary operators and equivalent named methods work either with two matrices, or with a matrix and a scalar (except Hermitian types). For instance, this code creates symmetric matrix by extracting the upper triangular region of a general matrix, then performs various arithmetic operations on it:

```

int rows = 6, cols = 6;
FloatMatrix A =
    new FloatMatrix( rows, cols, new RandGenUniform(-2,2) );
FloatSymmetricMatrix S = new FloatSymmetricMatrix( A );

float s = -.123F;
FloatSymmetricMatrix B = s * A;
FloatSymmetricMatrix C = B + A;

FloatVector x =
    new FloatVector( A.Cols, new RandGenUniform( -1,1 ) );
FloatVector y = A * x;

```

**NMath Core** also provides a variety of standard mathematical functions that take structured sparse matrix types as arguments. Methods are provided either as member functions on the matrix classes, or as static methods on class **MatrixFunctions**.

## Factorizations

**NMath Core** provides classes for computing and storing factorizations of structured sparse matrices, including:

- **LU factorization** for banded and tridiagonal matrices
- **Bunch-Kaufman factorization** for symmetric and Hermitian matrices
- **Cholesky factorization** for symmetric and Hermitian positive definite matrices

**NOTE—NMath Core provides two factorization classes for symmetric and Hermitian types: one for indefinite matrices, and one for positive definite (PD) matrices.**

Once a factorization is constructed, it can be reused to solve for different right-hand sides, and to compute inverses, determinants, and condition numbers. Similar static methods are also provided on class **MatrixFunctions**.

For instance, this code solves for one right-hand side:

```

DoubleMatrix genMat = new DoubleMatrix(
    "5x5 [ 1.0000 0.5000 0.2500 0.1250 0.0625
          0.5000 1.0000 0.5000 0.2500 0.1250
          0.2500 0.5000 1.0000 0.5000 0.2500
          0.1250 0.2500 0.5000 1.0000 0.5000
          0.0625 0.1250 0.2500 0.5000 1.0000 ]" );
DoubleSymmetricMatrix A = new DoubleSymmetricMatrix( genMat );
DoubleSymPDFact F = new DoubleSymPDFact( A );
DoubleVector v =
    new DoubleVector( A.Order, new RandGenUniform(-1,1) );
DoubleVector x = F.Solve( v );

```

The returned vector  $x$  is the solution to the linear system  $Ax = v$ . To do the same thing without explicitly constructing a factorization object, you could do this:

```
DoubleVector x = MatrixFunctions.Solve( A, v, true );
```

The optional third, boolean parameter indicates that  $A$  is positive definite.

Similarly, you can use the `Solve()` method to solve for multiple right-hand sides. This code solves for 10 right-hand sides:

```
int rows = 8, cols = 8;
DoubleComplexVector data =
    new DoubleComplexVector( cols*3, new RandGenUniform(-1, 1) );
DoubleComplexTriDiagMatrix A =
    new DoubleComplexTriDiagMatrix( data, rows, cols );
DoubleComplexTriDiagFact F = new DoubleComplexTriDiagFact( A );

DoubleComplexMatrix B =
    new DoubleComplexMatrix( A.Rows, 10, new RandGenUniform(-1,1) );

DoubleComplexMatrix X = F.Solve( B );
```

The returned matrix  $X$  is the solution to the linear system  $AX = B$ . That is, the right-hand sides are the columns of  $B$ , and the solutions are the columns of  $X$ . Matrix  $B$  must have the same number of rows as the factored matrix  $A$ .

You can also use a factorization to compute inverses using the `Inverse()` method, and determinants using the `Determinant()` method. For example:

```
int rows = 8, cols = 8;
FloatComplexMatrix Lehmer = new FloatComplexMatrix( rows, cols );
for ( int i = 0; i < rows; ++i )
{
    for ( int j = 0; j < cols; ++j )
    {
        if ( j >= i )
        {
            Lehmer[i,j] = (float)(i+1)/(float)(j+1);
        }
    }
}
FloatHermitianMatrix A = new FloatHermitianMatrix( Lehmer );

FloatHermitianPDFact F = new FloatHermitianPDFact( A );
FloatHermitianMatrix AInv = F.Inverse();
FloatComplex det = F.Determinant();
```

A `ConditionNumber()` method computes an estimate of the condition number in the one-norm.

## Decompositions

**NMath Core** includes decomposition classes for constructing and manipulating QR and singular value decompositions of the general matrix types in **NMath Core**. A *QR decomposition* is a representation of a matrix **A** of the form:

$$AP = QR$$

where **P** is a permutation matrix, **Q** is orthogonal, and **R** is upper trapezoidal (or upper triangular if **A** has more rows than columns and full rank). A *singular value decomposition* (SVD) is a representation of a matrix **A** of the form:

$$A = USV^*$$

where **U** and **V** are orthogonal, **S** is diagonal, and  $v^*$  denotes the transpose of a real matrix **V** or the conjugate transpose of a complex matrix **V**. The entries along the diagonal of **S** are the *singular values*. The columns of **U** are the *left singular vectors*, and the columns of **V** are the *right singular vectors*.

Instances of the decomposition classes are constructed from general matrices of the appropriate datatype. For example, this code creates a **FloatQRDecomp** from a **FloatMatrix**:

```
FloatMatrix A =
    new FloatMatrix( "5x3 [ 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 ]" );
FloatQRDecomp qr = new FloatQRDecomp( A );
```

**NMath Core** also provides decomposition *server* classes that construct instances of the decomposition classes, allowing you greater control over how decomposition is performed. For example, class **DoubleQRDecomp** computes the QR decomposition of a **DoubleMatrix**. By default, this decomposition class performs *pivoting*—that is, it may move columns in the input matrix to increase the robustness of the calculation. For control over how pivoting is performed, or to turn off pivoting altogether, the associated decomposition server class, **DoubleQRDecompServer**, may be used to create instances of **DoubleQRDecomp** with non-default decomposition parameters. For example, this code creates a **DoubleQRDecomp** without pivoting:

```
DoubleComplexQRDecompServer qrs = new
    DoubleComplexQRDecompServer();
qrs.Pivoting = false;

int rows = 10, cols = 3;
DoubleComplexMatrix A = new DoubleComplexMatrix( rows, cols,
    new RandGenUniform( -1, 1 ) );
DoubleComplexQRDecomp qr = qrs.GetDecomp( A );
```

## Least Squares Solutions

NMath Core includes least squares classes for solving the overdetermined linear system:

$$Ax = y$$

In a linear model, a quantity  $y$  depends on one or more independent variables  $a_1, a_2, \dots, a_n$  such that  $y = x_0 + x_1a_1 + \dots + x_na_n$ . A common goal of a least squares problem is to solve for the best values of  $x_0, x_1, \dots, x_n$ . The least squares solution is the value of  $x$  that minimizes the *residual vector*  $\|Ax - y\|$ .

Classes are provided that compute solutions using various methods: Cholesky factorization, QR decomposition, and singular value decomposition. The interface is virtually identical for all least squares classes.

- **Least Squares Using Cholesky Factorization**

The Cholesky least squares classes solve least square problems by using the Cholesky factorization to solve the normal equations. The normal equations for the least squares problem  $Ax = y$  are:

$$A^*Ax = A^*y$$

where  $A^*$  denotes the transpose of a real matrix  $A$  or the conjugate transpose of a complex matrix  $A$ . If  $A$  has full rank, then  $A^*A$  is symmetric/Hermitian positive definite—the converse is also true—and the Cholesky factorization may be used to solve the normal equations. This method will fail if the matrix  $A$  is rank deficient.

Finding least squares solutions using the normal equations is often the best method when speed is the only consideration.

- **Least Squares Using QR Decomposition**

The QR decomposition least squares classes solve least squares problems by using a QR decomposition to find the minimal norm solution to the linear system  $Ax = y$ . That is, they find the vector  $x$  that minimizes the 2-norm of the residual vector  $Ax - y$ . Matrix  $A$  must have more rows than columns, and be of full rank.

Finding least squares solutions via QR decomposition is the “standard” method for least squares problems, and is recommended for general use.

- **Least Squares Using SVD**

If the matrix  $A$  is close to rank-deficient, the QR decomposition method described above has less than ideal stability properties. In such cases, a method based on singular value decomposition is a better choice.

Instances of the least squares classes are constructed from general matrices of the appropriate datatype. For example, this code creates a **FloatCholeskyLeastSq** from a **FloatMatrix**:

```
FloatMatrix A = new FloatMatrix( "4x2[ 1 0  0 1  0 0  0 0 ]" );
FloatCholeskyLeastSq lsq = new FloatCholeskyLeastSq( A );
```

Once a least squares object has been constructed from a matrix, it may be used to solve least squares problems. All least squares classes provide a `Solve()` method that accepts a vector  $y$ , and computes the solution to the least squares problem  $Ax = y$ . For example:

```
int rows = 6, cols = 3;
RandGenUniform rng = new RandGenUniform( -2, 2 );

DoubleMatrix A = GenerateData( rows, cols, rng );
DoubleCholeskyLeastSq lsq = new DoubleCholeskyLeastSq( A );

DoubleVector y = new DoubleVector( rows, rng );
if ( lsq.IsGood )
{
    DoubleVector x = lsq.Solve( y );
}
```

Method `ResidualVector()` returns the residual vector  $Ax - y$ ; `ResidualNormSqr()` computes the 2-norm squared of the residual vector. Finally, an existing least squares object can factor other matrices using the `Factor()` method.

## Eigenvalue Problems

**NMath Core** includes classes for solving symmetric, Hermitian, and nonsymmetric eigenvalue problems. For example, class **DoubleSymEigDecomp** computes the eigenvalues and eigenvectors of a **DoubleSymmetricMatrix**:

```
DoubleSymEigDecomp decomp = new DoubleSymEigDecomp( A );
Console.WriteLine( "Eigenvalues = " +
    decomp.EigenValues );
Console.WriteLine( "Left eigenvectors = " +
    decomp.LeftEigenVectors );
Console.WriteLine( "Right eigenvectors = " +
    decomp.RightEigenVectors );
```

By default, eigenvalue classes compute both eigenvalues and eigenvectors.

**NMath Core** also provides eigenvalue *server* classes that construct instances of the eigenvalue classes, allowing you greater control over how the eigenvalue decomposition is performed. Server classes can be configured to compute

eigenvalues only, or both eigenvalues and eigenvectors. In addition, the server can be configured to compute only the eigenvalues in a given range.

```
FloatEigDecompServer eigServer = new FloatEigDecompServer();
eigServer.ComputeLeftVectors = false;
eigServer.ComputeRightVectors = false;
server.ComputeEigenValueRange( 0, 3 );

FloatEigDecomp decomp = eigServer.Factor( A );
```

A tolerance for the convergence of the algorithm may also be specified.

## Conclusions

**NMath Core** makes advanced matrix manipulation easy. By leveraging the power of the general matrix classes in **NMath Core**, **NMath Core** provides elegant object-oriented interfaces for structured sparse matrix types, factorizations, general matrix decompositions, advanced least squares solutions, and solutions to eigenvalue problems. By utilizing highly-optimized linear algebra subroutine libraries for most computation, **NMath Core** offers performance levels comparable to C or Fortran.

## **ADVANCED MATRIX MANIPULATION WITH NMATH MATRIX**

© 2008 Copyright CenterSpace Software, LLC. All Rights Reserved.

The correct bibliographic reference for this document is:

*Advanced Matrix Manipulation with NMath Matrix*, Technical Report No. 3, CenterSpace Software, Corvallis, OR.

Printed in the United States.

Printing Date: November, 2008

### **CENTERSPACE SOFTWARE**

Address:	301 SW 4th St, Suite #240, Corvallis, OR 97333 USA
Phone:	(866) 864-7202
Web:	<a href="http://www.centerspace.net">http://www.centerspace.net</a>
Technical Support:	<a href="mailto:support@centerspace.net">support@centerspace.net</a>