



---

# **.NET Numerical Applications with NMath Core**

**Technical Report No. 1**

**CenterSpace Software  
Corvallis, Oregon**

---



---

## Introduction

Fortran has been the dominant language for numerical computing for many years. When C compilers became highly-optimized, numerical application developers began to take advantage of the features that language had to offer. More recently, developers have exploited the object-oriented design and implementation advantages inherent in C++.

Today, there is a new computing platform available: Microsoft .NET. Formally launched in February, 2002, Microsoft .NET is Microsoft's latest and most ambitious effort yet to redefine enterprise computing. Microsoft has described .NET as a platform that "allow[s] applications to communicate and share data over the Internet, regardless of operating system or programming language." For example, the .NET platform is designed to be language-neutral—all .NET-aware programming languages compile to the same platform- and language-agnostic Microsoft Intermediate Language (MSIL). All MSIL code runs under the Common Language Runtime (CLR).

C# is a new object-oriented programming language derived from C++ and Java, but designed by Microsoft from the ground up. Though it is just one of the languages that can be used to write code for .NET, it is the only language that was written from the outset with .NET in mind, making C# the language of choice for .NET development.

C#, and the .NET variant of C++ called "Managed C++", together offer access to pointer arithmetic and relatively efficient access to native, highly-optimized methods, making .NET a viable platform for numerical applications.

CenterSpace Software's **NMath™ Core** is a .NET class library providing the basic building blocks for numerical applications on the .NET platform. **NMath Core** contains vector classes, matrix classes, complex number classes, random number generators, numerical integration methods, and other high-performance functions for object-oriented numerics. Fully compatible with the Common Language Specification, all **NMath Core** routines are callable from any .NET language, including C# and Visual Basic.NET.

**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/Core/>

# Features

The features of **NMath Core** include:

- Single- and double-precision complex number classes.
- Full-featured vector and matrix classes for four datatypes: single- and double-precision floating point numbers, and single- and double-precision complex numbers.
- Flexible indexing using slices and ranges.
- 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.
- Extension of standard mathematical functions, such as `Cos()`, `Sqrt()`, and `Exp()`, to work with vectors, matrices, and complex number classes.
- LU factorization for matrices, as well as functions for solving linear systems, computing determinants, inverses, and condition numbers.
- Least squares solutions.
- Random number generation from various probability distributions.
- Classes for encapsulating functions of one variable, with support for numerical integration (Romberg and Gauss-Kronrod methods), differentiation (Ridders' method), and algebraic manipulation of functions.
- Polynomial encapsulation, interpolation, and exact differentiation and integration.
- Fully persistable data classes using standard .NET mechanisms.
- Integration with ADO.NET.

**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.

# Design

**NMath Core** employs the *data-view* design pattern by distinguishing between data, and the different ways mathematical objects such as vectors and matrices view the data. For example, a contiguous array of numbers in memory might be viewed by one object as the elements of a vector, while another object might view the same data as the elements of a matrix, laid out row by row. At any given point in time, many different objects might share a given block of data. The data-view pattern has definite advantages for both storage efficiency and performance. Combined with slicing, the data-view pattern also offers a very rich set of matrix and vector manipulation semantics.

## Complex Number Classes

In **NMath Core**, classes **FloatComplex** and **DoubleComplex** represent complex numbers, consisting of real and imaginary parts of single- and double-precision floating point numbers.

These classes support equality operations, conversion from `float`, `double`, or a string representation, and basic arithmetic operations. Static member functions are also provided for returning the argument (or phase) of a complex number, the complex conjugate, the norm (or modulus), and for converting from polar coordinates. Trigonometric functions for complex numbers, and mathematical functions such as exponents, logarithms, powers, and square roots, are also available.

## DataBlock Classes

The classes that encapsulate blocks of data in **NMath Core** are named `<Type>DataBlock`, where `<Type>` is **Float**, **Double**, **FloatComplex**, or **DoubleComplex**. For example, the **DoubleDataBlock** class represents an array of double-precision floating point numbers.

Each `<Type>DataBlock` class contains a reference to an array of the appropriate data type, and an offset into the array. You can think of a data block as encapsulating the concept of a pointer without using unsafe code. (The value of an equivalent pointer would be the address of the first element of the array, plus the offset.) The data referenced by **NMath Core** vector and matrix classes is in the form of an instance to a data block.

You rarely need to deal directly with `<Type>DataBlock` objects, though accessing a pointer to the underlying data is one instance where dealing with a **DataBlock** object is necessary (see below).

## Vector and Matrix Classes

The classes that encapsulates matrices and vectors in **NMath Core** are named **<Type>Matrix** and **<Type>Vector**, where **<Type>** is **Float**, **Double**, **FloatComplex**, or **DoubleComplex**. For example, the **FloatComplexVector** class represents a vector of single-precision complex numbers.

The matrix and vector classes each contain a reference to the data block they are viewing, along with the parameter values necessary to define their view. For example, an instance of a vector class contains a reference to a block of data, the number of elements referenced, and a *stride*, or step increment, between elements of the block of data. Similarly, a matrix object contains a reference to a block of data, the number of rows and columns, the distance between successive row elements, and the distance between successive column elements.

All of this is transparent to you. The provided indexers perform the necessary indirection. For example, **v[i]** always returns the *i*th element of vector **v**'s view of the data, and **A[i, j]** always returns the element in the *i*th row and *j*th column of matrix **A**'s view of the data.

## Ranges and Slices

The most common way of obtaining a different view of a specific block of data is by using **Slice** and **Range** indexing objects. These classes provide a way to specify a subset on non-negative integers with constant spacing. (For those of you familiar with MATLAB, this is essentially the colon operator.) An integer subset can then be used as an indexing object into matrices and vectors.

The difference between a **Slice** and a **Range** is only in how the integer subset is specified. For a **Slice**, a beginning integer is specified, along with the number of integers and a stride. For example, to create a slice specifying the integers { 2, 4, 6, 8, 10 }, specify a start of 2, 5 total elements, and a stride of 2.

A **Range** defines an integer subset by specifying a first and last integer, inclusive, along with a stride. Thus, to create a range object defining the set of integers above, specify a starting point of 2, a stopping point of 10, and a stride of 2.

Here's an example using **Slice** objects to create a new view of a vector's data:

```
using CenterSpace.NMath.Core;

// Create a vector of length 10 containing the integers 1-10:
DoubleVector v = new DoubleVector( 10, 1, 1 );

// Construct a new vector, u, that views the first three elements
// of v
Slice first3 = new Slice( 0, 3 );
DoubleVector u = v[first3];
```

Notice that the **DoubleVector** indexer is overloaded to accept **Slice** objects, and return a new view of the indexed data.

**DoubleVector** *u* behaves exactly like a vector constructed with 3 elements whose values are 1, 2, 3. That is:

```
u[0] == 1; // true
u[1] == 2; // true
u[2] == 3; // true
u[3]; //Index out of bounds exception!
```

The difference between *u* and a newly constructed vector becomes clear when a value in *u* is changed. This changes the corresponding value in *v*, since they both reference the same data.

```
u[2] = 99;
v[2] == 99; // true!
```

This feature can come in handy at times. Suppose, for instance, that you want to increment every element along the main diagonal of a matrix. Here is what the code would look like:

```
using CenterSpace.NMath.Core;

DoubleMatrix A = new DoubleMatrix( 5, 8 );
A.Diagonal()++;
```

Here the `Diagonal()` method on the matrix class uses slices beneath the covers to construct a vector that views the subset of the data that composes the main diagonal of the matrix.

If, after constructing a different view of an object's data, you want your own private view of the data that you can modify without affecting any other objects, you can invoke the `DeepenThisCopy()` method:

```
using CenterSpace.NMath.Core;

DoubleMatrix A = new DoubleMatrix( 8, 8 );
Range topLeft = new Range( 0, 3 );

// Construct a matrix that views the top left
// corner of A.
DoubleMatrix AtopLeft = A[ topLeft, topLeft ];

// Make a deep copy of this data
AtopLeft.DeepenThisCopy();
```

The data-view pattern combined with slicing offers a very rich set of matrix and vector manipulation semantics.

## Accessing the Underlying Data

For applications that need to interface with native or legacy code, the **NMath Core** vector and matrix classes can be used to obtain a pointer to the underlying data. Each of these classes has a property called `DataBlock` that returns the `<Type>DataBlock` object being viewed. As mentioned above, each `<Type>DataBlock` class contains an array and an offset that allows you to extract a pointer to the beginning of the data. For example:

```
using CenterSpace.NMath.Core;

DoubleVector v = new DoubleVector( 12, 0, 1 );

DoubleDataBlock dataBlock = v.DataBlock;
unsafe
{
    double *ptr = &(dataBlock.Data[dataBlock.Offset]);

    // Do with *ptr something here
}
```

Exercise caution when using raw data pointers.

## Applying Functions

**NMath Core** provides convenience methods for applying unary and binary functions to elements of a vector or matrix object. Each of these methods takes a function **Delegate**. The `Apply()` method returns a new object whose contents are the result of applying the given function to each element of the matrix or vector. The `Transform()` method modifies a matrix or vector object by applying the given function to each of its elements. Thus, assuming `MyFunc` is a function that takes a `double` and returns a `double`:

```
using CenterSpace.NMath.Core;

DoubleVector v = new DoubleVector ( 10, 0, -1 );

// Construct a delegate for MyFunc
DoubleUnaryFunction MyFuncDelegate =
    new DoubleUnaryFunction( MyFunc );

// Construct a new vector whose values are the result of applying
// MyFunc to the values in vector v. v remains unchanged.
DoubleVector w = v.Apply( MyFuncDelegate );
```

```
// Transform the contents of v.  
v.Transform( MyFuncDelegate );
```

```
v == w; // true!
```

**NMath Core** also provides the `ApplyColumns()` method on the matrix classes for applying a vector function to columns of a matrix. This function takes a function delegate that accepts a vector and returns a single value.

For instance, assuming `MyFunc` takes a **FloatVector** and returns a `float`:

```
using CenterSpace.NMath.Core;  
  
FloatMatrix A = new FloatMatrix( 5, 5, 0, Math.Pi/4 );  
  
FloatVectorFunction MyFuncDelegate =  
    new FloatVectorFunction( MyFunc );  
  
FloatVector v = A.ApplyColumns( MyFuncDelegate );
```

To apply a function to the rows of matrix, just transpose the matrix first. The `Transpose()` method simply swaps the number of rows and the number of columns, and other view parameters of the matrix. No data is copied, so it's a relatively cheap operation. For instance:

```
using CenterSpace.NMath.Core;  
  
FloatVector v = A.Transpose().ApplyColumns( MyFuncDelegate );  
A.Transpose(); // return A to original view
```

**NMath Core** provides delegates for many commonly used math functions in the **NMathFunctions** class. This class also provides overloaded versions of these functions that accept matrix, vector, and complex number objects as arguments:

```
using CenterSpace.NMath.Core;  
  
FloatComplexMatrix A = new FloatComplexMatrix( 5, 5 );  
  
// Construct a matrix whose contents are the cosines  
// of the elements of A.  
FloatComplexMatrix cosA = NMathFunctions.Cos( A );
```

## Solutions of Linear Systems

**NMath Core** provides classes for computing and storing the LU factorization for a matrix, as well as several static functions for solving linear systems, computing determinants, inverses, and condition numbers. Once an LU factorization is constructed, it can be reused to solve for different right hand sides, to compute inverses, to compute condition numbers, and so on.

```

using CenterSpace.NMath.Core;

DoubleComplexMatrix A = new DoubleComplexMatrix( 5, 5, 1, 1 );

DoubleComplexLUFact lu = new DoubleComplexLUFact( A );

// Solve for one right-hand side.
DoubleVector b = new DoubleVector( 5 );
DoubleVector x = lu.Solve( b );
double conditionNumber = lu.ConditionNumber();

// Solve for several right-hand sides.
DoubleComplexMatrix B = new DoubleComplexMatrix( 5, 5, 5, 2 );
DoubleMatrix X;
if ( lu.IsGood )
{
    X = lu.Solve( B );
}

// One shot solution.
try
{
    X = NMathFunctions.Solve( A, B );
}
catch ( NMathException )
{
    // Can be one of SingularMatrixException,
    // MatrixNotSquareException, or MismatchedSizeException
}

```

## Least Squares

**NMath Core** provides classes for computing the minimum-norm solution to a linear system  $Ax = y$ . These classes use a complete orthogonal factorization of **A** to compute the solution. For instance, this code calculates the slope and intercept of a linear least squares fit through five data points, then prints out the properties of the solution:

```

using CenterSpace.NMath.Core;

DoubleMatrix A =
    new DoubleMatrix( "5x1[20.0 30.0 40.0 50.0 60.0]" );
DoubleVector y = new DoubleVector( "[.446 .601 .786 .928 .950]" );

DoubleLeastSquares lsq = new DoubleLeastSquares( A, y, true );

```

```

Console.WriteLine( "Y-intercpt = {0}", lsq.X[0] );
Console.WriteLine( "Slope = {0}", lsq.X[1] );
Console.WriteLine( "Residuals = {0}", lsq.Residuals.ToString() );
Console.WriteLine( "Residual Sum of Squares (RSS) = {0}",
    lsq.ResidualSumOfSquares.ToString

```

## Random Number Generators

**NMath Core** provides random number generators that generate random deviates from a variety probability distributions, including the uniform, normal, log-normal, Poisson, gamma, binomial, Pareto, and exponential distributions.

All **NMath Core** generators inherit from the abstract base class **RandomNumberGenerator**, providing a common interface. All **NMath Core** generators provide a `Next()` method that returns a random deviate. For instance, this code prints out 100 random deviates from a normal distribution with mean of -12.9 and variance of 2.066:

```

using CenterSpace.NMath.Core;

double mean = -12.9;
double variance = 2.066;
RanGenNormal gen = new RandGenNormal( mean, variance );

for (int i=0; i<100, i++)
{
    Console.WriteLine( gen.Next() );
}

```

As a convenience, **NMath Core** vector and matrix classes provide constructor overloads that initialize all elements with random values. For example:

```

using CenterSpace.NMath.Core;

RandGenUniform gen = new RandGenUniform( 0, 100 );
FloatVector v = new DoubleVector( 10, gen );
DoubleComplexMatrix A = new DoubleComplexMatrix( 25, 25, gen );

```

All **NMath Core** random number generators, regardless of the distribution, require an underlying uniform random number generator that returns random deviates in the range zero to one. Each generator first generates a random uniform deviate in the range zero to one, then from this deviate derives a random number from the appropriate probability distribution.

By default, all generators use the **NMath Core** class **RandGenMTwist** as the underlying uniform generator. **RandGenMTwist** implements the Mersenne Twister algorithm, developed by Makoto Matsumoto and Takuji Nishimura in

1996-1997. This algorithm is faster and more efficient, and has a far longer period and far higher order of equidistribution, than other existing generators.

## Encapsulating Functions

In **NMath Core**, class **OneVariableFunction** encapsulates an arbitrary function, and works with other numerical classes to approximate integrals and derivatives. For example, suppose you wish to encapsulate this function:

```
public double MyFunction( double x )
{
    return Math.Sin( x ) + Math.Pow( x, 3 ) / Math.PI;
}
```

This code creates a delegate for the `MyFunction()` method, then constructs a **OneVariableFunction** encapsulating the delegate:

```
using CenterSpace.NMath.Core;

NMathFunctions.DoubleUnaryFunction d =
    new NMathFunctions.DoubleUnaryFunction( MyFunction );

OneVariableFunction f = new OneVariableFunction( d );
```

The `Evaluate()` method on **OneVariableFunction** evaluates a function at a given  $x$ -value or vector of  $x$ -values. Thus, if `f` is a **OneVariableFunction**, this code evaluates `f` at 100 points between 0 and 1:

```
DoubleVector x = new DoubleVector( 100, 0, 1.0/100 );
DoubleVector y = f.Evaluate( x );
```

**NMath Core** provides overloaded arithmetic operators for functions with their conventional meanings for those .NET languages that support them, and equivalent named methods for those that do not. For example, if `f` and `g` are **OneVariableFunction** objects:

```
OneVariableFunction sum = f + g;
double y = sum.Evaluate( Math.PI );
```

## Numerical Integration and Differentiation

Numerical integration, also called *quadrature*, computes an approximation of the integral of a function over some interval. **NMath Core** provides two of the most widely used, general purpose families of methods: *Romberg* integration, and *Gauss-Kronrod* integration.

The `Integrate()` method on **OneVariableFunction** computes the integral of a function over a given interval. For example, if `f` is **OneVariableFunction**, this code integrates `f` over the interval `-1` to `1`:

```
double integral = f.Integrate( -1, 1 );
```

By default, **OneVariableFunction** objects use **RombergIntegrator** objects to compute integrals, but this may be changed using the `Integrator` property. For instance:

```
f.Integrator = new GaussKronrodIntegrator();  
double integral = f.Integrate( 0, Math.PI );
```

The automatic **GaussKronrodIntegrator** class uses Gauss-Kronrod rules with increasing number of points. Approximation ends when the estimated error is less than a specified tolerance, or when the maximum number of points is reached.

Similarly, the `Differentiate()` method on **OneVariableFunction** computes the derivative of a function at a given `x`-value. For example, if `f` is **OneVariableFunction**, this code estimates the derivative at `0`:

```
double d = f.Differentiate( 0 );
```

**OneVariableFunction** objects use **RiddersDifferentiator** objects, which compute the derivative of a given function by *Ridders' method* of polynomial extrapolation, and implements. Extrapolations of higher and higher order are produced. Iteration stops when either the estimated error is less than a specified error tolerance, the error estimate is significantly worse than the previous order, or the maximum order is reached.

## Polynomials

Class **Polynomial** extends **OneVariableFunction**. Rather than encapsulating an arbitrary function delegate, **Polynomial** represents a polynomial by its coefficients, arranged in ascending order. A **Polynomial** instance can be constructed in two ways. If you know the exact form of the polynomial, simply pass in the vector of coefficients:

```
DoubleVector coef = new DoubleVector( "1 0 2"); // 2x^2 + 1  
Polynomial p = new Polynomial( coef );
```

Alternatively, you can interpolate a polynomial through a set of points. If the number of points is  $n$ , then the constructed polynomial will have degree  $n - 1$  and pass through the interpolation points.

Class **Polynomial** inherits the `Integrate()` method from **OneVariableFunction**, which computes the integral of the current function over a given interval. **Polynomial** also extends the interface to include an `AntiDerivative()` method

that returns a new polynomial encapsulating the antiderivative (indefinite integral) of the current polynomial. For example:

```
Polynomial p = new Polynomial( new DoubleVector( "5 3 0 2" ) );  
Polynomial i = p.AntiDerivative();
```

The constant of integration is assumed to be zero.

Each **Polynomial** object has a **PolynomialIntegrator** associated with it. Because the antiderivative of a polynomial can be easily constructed, **PolynomialIntegrator** simply constructs the antiderivative and evaluates it at the lower and upper bounds. This gives the exact integral.

Class **Polynomial** inherits both `Differentiate()` and `Derivative()` methods from **OneVariableFunction**. `Differentiate()` returns the derivative of the current function at a given  $x$ -value. `Derivative()` is overridden to return a new polynomial that is the first derivative of the current polynomial. Thus:

```
DoubleVector coeff = new DoubleVector( "1 -2 3" );  
Polynomial p = new Polynomial( coeff );  
Polynomial der = p.Differentiate(); // der.Coeff = "-2 6"
```

This gives the exact derivative.

## ADO.NET Integration

**NMath Core** provides convenience methods for creating ADO.NET objects from vectors and matrices, and for creating vectors and matrices from database objects. Real-value **NMath Core** vector and matrix classes provide `ToDataTable()` methods for creating ADO.NET **DataTable** objects. Complex number vector and matrix classes provide paired methods `ToRealDataTable()` and `ToImagDataTable()` for creating **DataTable** objects containing the real and imaginary parts, respectively. For example, this code creates a data table named `MyMatrixTable` containing the values in a **DoubleMatrix**:

```
using System.Data;  
using CenterSpace.NMath.Core;  
  
DoubleMatrix A = new DoubleMatrix( 8, 5, 3.1415 );  
DataTable table = A.ToDataTable( "MyMatrixTable" );
```

You can also construct **NMath Core** vector and matrix classes from standard ADO.NET database objects. Real-value vector and matrix class constructors accept **DataTable**, **DataRow**, **DataRowCollection**, and **DataRowView** objects, typically obtained from a database query.

## Serialization

**NMath Core** data classes are fully persistable using standard .NET mechanisms. All matrix, vector, and LU decomposition classes implement the **ISerializable** interface to control their own serialization and deserialization. Common Language Runtime (CLR) serialization **Formatter** classes call the provided `GetObjectData()` methods at serialization time to populate **SerializationInfo** objects with all the data required to represent **NMath Core** objects. For instance, this code serializes two **FloatComplexMatrix** objects to a file in binary format:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using CenterSpace.NMath.Core;

FloatComplexMatrix A =
    new FloatComplexMatrix( "2x2[ (5,9.8) (-6,.9) (7,-8) (8,8) ]" );
FloatComplexMatrix B = new FloatComplexMatrix( 4, 4, .1F, .1F );

FileStream binStream = File.Create( "myData.dat" );
BinaryFormatter binFormatter = new BinaryFormatter();

binFormatter.Serialize( binStream, A );
binFormatter.Serialize( binStream, B );

binStream.Close();
```

This code restores the objects from the file:

```
binStream = File.OpenRead( "myData.dat" );

FloatComplexMatrix A2 =
    (FloatComplexMatrix)binFormatter.Deserialize( binStream );
FloatComplexMatrix B2 =
    (FloatComplexMatrix)binFormatter.Deserialize( binStream );

binStream.Close();
File.Delete( "myData.dat" );
```

## Exception Handling

All exceptions thrown by **NMath Core** inherit from the **NMathException** class, enabling you to easily catch all **NMath Core** exceptions.

## Conclusions

**NMath Core** provides the basic building blocks for numerical applications on the .NET platform. By providing elegant object-oriented interfaces to highly-optimized linear algebra subroutine libraries, and offering performance levels comparable to C or Fortran, **NMath Core** makes numerical computing on the .NET platform a reality.

## **.NET NUMERICAL APPLICATIONS WITH NMATH CORE**

© 2009 Copyright CenterSpace Software, LLC. All Rights Reserved.

The correct bibliographic reference for this document is:

*.NET Numerical Applications with NMath Core*, Technical Report No. 1, CenterSpace Software, Corvallis, OR.

Printed in the United States.

Printing Date: April, 2009

### **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>