.NET Numerical Applications with NMath

CenterSpace™ Software

Introduction

CenterSpace Software's **NMath**™ numerical library provides object-oriented components for mathematical, engineering, scientific, and financial applications on the .NET platform. **NMath** contains vector classes, matrix classes, complex number classes, random number generators, and other high-performance functions for object-oriented numerics. **NMath** also provides functions and data structures for statistical computation, including descriptive statistics, probability distributions, combinatorial functions, multiple linear regression, hypothesis testing, and analysis of variance.

NOTE— Starting with the release of NMath 7.0, NMath includes all of the statistical routines previously available in the NMath Stats product. The NMath Stats product is no longer available separately.

For most core computations, **NMath** uses the Intel® Math Kernel Library (MKL), which contains highly-optimized, extensively-threaded versions of the C and FORTRAN public domain computing packages known as the BLAS (Basic Linear Algebra Subroutines) and LAPACK (Linear Algebra PACKage). This gives **NMath** classes performance levels comparable to C, and often results in performance an order of magnitude faster than non-platform-optimized implementations.

NMath provides a modern, easy to use, object-oriented interface, including a very rich set of matrix and vector manipulation semantics. Fully compliant with the Microsoft Common Language Specification (CLS), all **NMath** routines are callable from any .NET language, including C#, Visual Basic.NET, and F#.

NMath supports the .NET Standard Library.

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

`http://www.centerspace.net/examples/NMath/`

NMath Features

The features of **NMath** 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.
- Full-featured structured sparse matrix classes, including triangular, symmetric, Hermitian, banded, tridiagonal, symmetric banded, and Hermitian banded.
- Functions for converting between general matrices and structured sparse matrix types.
- Functions for transposing structured sparse matrices, computing inner products, and calculating matrix norms.
- 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.
- General sparse vector and matrix classes, and matrix factorizations.
- 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.
- LU factorization for general matrices, as well as functions for solving linear systems, computing determinants, inverses, and condition numbers.
- Classes for solving symmetric, Hermitian, and nonsymmetric eigenvalue problems.
- Extension of standard mathematical functions, such as `Cos()`, `Sqrt()`, and `Exp()`, to work with vectors, matrices, and complex number classes.
- Random number generation from various probability distributions.
- Fast Fourier Transforms (FFTs), and linear convolution and correlation.
- Discrete Wavelet Transforms (DWTs).

- 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.
- Classes for minimizing univariate functions using golden section search and Brent's method.
- Classes for minimizing multivariate functions using the downhill simplex method, Powell's direction set method, the conjugate gradient method, and the variable metric (or quasi-Newton) method.
- Simulated annealing.
- Classes for linear programming (LP), nonlinear programming (NLP), and quadratic programming (QP) using the Microsoft Solver Foundation.
- Least squares polynomial fitting.
- Nonlinear least squares minimization, curve fitting, and surface fitting.
- Classes for finding roots of univariate functions using the secant method, Ridders' method, and the Newton-Raphson method.
- Numerical methods for double integration of functions of two variables.
- Nonlinear least squares minimization using the Trust-Region method, a variant of the Levenberg-Marquardt method.
- Curve and surface fitting by nonlinear least squares.
- Classes for solving stiff and non-stiff first order initial value differential equations.
- Special functions, such factorial, binomial, the gamma function and related functions, Bessel functions, elliptic integrals, and many more.
- Fully persistable data classes using standard .NET mechanisms.
- Integration with ADO.NET.

NMath also includes a powerful set of statistical functions:

- A data frame class for holding data of various types (numeric, string, boolean, datetime, and generic), with methods for appending, inserting, removing, sorting, and permuting rows and columns.
- Functions for computing descriptive statistics, such as mean, variance, standard deviation, percentile, median, quartiles, geometric mean, harmonic mean, RMS, kurtosis, skewness, and many more.

- Probability density function (PDF), cumulative distribution function (CDF), inverse CDF, and random variable moments for a variety of probability distributions, including normal (Gaussian), Poisson, chi-square, gamma, beta, Student's *t*, *F*, binomial, and negative binomial.
- Multiple linear regression and logistic regression.
- Basic hypothesis tests, such as z-test, t-test, F-test, and Pearson's chi-square test, with calculation of p-values, critical values, and confidence intervals.
- One-way and two-way analysis of variance (ANOVA) and analysis of variance with repeated measures (RANOVA).
- Non-parametric tests, such as the Kolmogorov-Smirnov test and Kruskal-Wallis rank sum test.
- Multivariate statistical analyses, including principal component analysis, factor analysis, hierarchical cluster analysis, and *k*-means cluster analysis.
- Nonnegative matrix factorization (NMF), and data clustering using NMF.
- Partial least squares (PLS).
- Statistical process control.

Namespaces

NMath types are organized into a single namespace:

- `CenterSpace.NMath.Core`

  All **NMath** classes reside in the `Core` namespace.

NOTE—The legacy NMath namespaces of `CenterSpace.NMath.Matrix` and `CenterSpace.NMath.Analysis` are supported but serve only as synonyms for the `CenterSpace.NMath.Core` namespace. The `CenterSpace.NMath.Stats` namespace has also been similarly deprecated now that the retired NMath Stats product is exclusively included within NMath.

General Vectors and Matrices

NMath 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.

General Vector and Matrix Classes

The classes that encapsulate general matrices and vectors in **NMath** 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:

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

// Construct a new vector, u, that views the first three elements
// of v
var 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:

```
var 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:

```
var A = new DoubleMatrix( 8, 8 );
var 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** 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:

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

var 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 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`:

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

// Construct a delegate for MyFunc
var MyFuncDelegate = new Func<double, double>( 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 provides delegates for many commonly used math functions in the **NMathFunctions** class.

Matrix Decompositions

NMath includes decomposition classes for constructing and manipulating QR and singular value decompositions of the general matrix types in **NMath**. 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**:

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

NMath 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:

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

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

Structured Sparse Matrix Classes

NMath 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.

NMath 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;
var 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
}
```

Structured Sparse Matrix Factorizations

NMath 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 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:

```
var 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 ]" );
var A = new DoubleSymmetricMatrix( genMat );
var F = new DoubleSymPDFact( A );
var 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;
var data =
  new DoubleComplexVector( cols*3, new RandGenUniform(-1, 1) );
```

```
var A = new DoubleComplexTriDiagMatrix( data, rows, cols );
var F = new DoubleComplexTriDiagFact( A );

var 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;
var 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);
    }
  }
}
var A = new FloatHermitianMatrix( Lehmer );

var 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.

General Sparse Vectors and Matrices

NMath provides classes for storing *general* sparse vector and matrix data. By storing only the non-zero values, the storage savings are significant. Unlike the *structured* sparse matrices described above, where the zero elements are distributed according to some pattern, general sparse matrices make no assumptions about the sparsity structure of the matrix. General sparse matrix data is stored in compressed row storage format (CSR).

For example, this code constructs a **DoubleCsrSparseMatrix** from an **IDictionary** of values, where row-column pairs are the keys and the non-zero entries are the values:

```
var values = new Dictionary<IntPair, double>();

values.Add( new IntPair( 0, 0 ), 1 );
values.Add( new IntPair( 0, 2 ), 2 );
values.Add( new IntPair( 1, 2 ), 3 );
values.Add( new IntPair( 2, 1 ), 4 );

int cols = 3;
var sA = new DoubleCsrSparseMatrix( values, cols );
```

NMath also provides classes for computing and storing factorizations of general sparse matrices. Instances of the factorization classes calculate solutions to the equation `Ax = B` where *A* is a sparse matrix and *B* is a single vector, or multiple vectors. Once a factorization is constructed, it can be reused to solve for different right-hand sides. This code creates a symmetric sparse matrix from the given data, factors the matrix, then solves for one right-hand side:

```
var sA = new DoubleSymCsrSparseMatrix( sparseData);
var fact = new DoubleSparseSymFact( sA );

if ( fact.ErrorStatus == DoubleSparseSymFact.Error.NoError )
{
  var b = new DoubleVector( 8, 1 );
  DoubleVector x = fact.Solve( b );
}
```

Eigenvalue Problems

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

```
var 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 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.

```
var 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.

Random Number Generators

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

All **NMath** generators inherit from the abstract base class **RandomNumberGenerator**, providing a common interface. All **NMath** 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`:

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

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

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

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

All **NMath** 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** 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.

Fourier Transforms, Convolution and Correlation

NMath provides classes for performing Fast Fourier Transforms (FFTs) on real and complex 1D and 2D data, and for performing linear convolution and correlation on real and complex 1D data.

For example, the following code creates some random signal data, then performs an FFT in place:

```
int dataLength = 1024;
var data = new DoubleVector( dataLength, new RandGenUniform() );
var fft = new DoubleForward1DFFT( dataLength );
```

This creates a complex conjugate symmetric packed result. You can ask the FFT instance for the correct reader, to unpacked the result:

```
DoubleSymmetricSignalReader reader = fft.GetSignalReader( data );
```

This code performs a convolution on the signal data:

```
var kernel = new DoubleVector( ".2 .2 .2 .2 .2" );
var conv = new Double1DConvolution( kernel, dataLength );
DoubleVector result = conv.Convolve( data );
```

Wavelets

NMath provides classes for performing discrete wavelet transforms (DWTs) using most common wavelet families, including Harr, Daubechies, Symlet, Best Localized, and Coiflet. Custom wavelets can also be created. DWT classes support both single step forward and reverse DWTs, and multilevel signal deconstruction

and reconstruction. Details thresholding at any level and threshold calculations are also supported.

For instance, this code constructs a Daubechies wavelet of length 4, builds a DWT object using the wavelet and some signal data, decomposes the signal to level 5, thresholds all detail levels, and rebuilds the filtered signal:

```
var wavelet = new DoubleWavelet( Wavelet.Wavelets.D4 );
var dwt = new DoubleDWT( data, wavelet );
dwt.Decompose( 5 );

double lambdaU = dwt.ComputeThreshold(
  DoubleDWT.ThresholdMethod.Universal, 1 );
dwt.ThresholdAllLevels( DoubleDWT.ThresholdPolicy.Soft,
  new double[] { lambdaU, lambdaU, lambdaU, lambdaU, lambdaU } );
double[] reconstructedData = dwt.Reconstruct();
```

Signal Processing

NMath provides classes for processing 1D signal data, including filtering using **MovingWindowFilter** or **SavitzkyGolayFilter**, and peak finding using **PeakFinderSavitzkyGolay**.

For example, **MovingWindowFilter** replaces data points *f(i)* with a linear combination, *g(i)*, of the data points immediately to the left and right of *f(i)*, based on a given set of coefficients, *c*, to use in the linear combination. Static class methods are provided for generating coefficients to implement a moving average filter and a Savitzky-Golay smoothing filter.

The following code constructs a noisy cosine signal, and then filters the data using a Sovitzky-Golay filter that replaces each input data point with the value of a fourth degree polynomial fit through the input value and it's surrounding points:

```
var rng = new RandGenNormal();
var noisySignal = new DoubleVector( length );
for ( int i = 0; i < length; i++ )
{
  noisySignal[i] = Math.Cos( .2*i ) + rng.Next();
}

int numberLeft = 3;
int numberRight = 3;
int degree = 4;
filterCoefficients =
  MovingWindowFilter.SavitzkyGolayCoefficients( numberLeft,
                                                numberRight,
                                                degree );
```

```
var filter = new MovingWindowFilter( numberLeft, numberRight,
  filterCoefficients );

DoubleVector filteredSignal = filter.Filter( noisySignal,
  MovingWindowFilter.BoundaryOption.PadWithZeros );
```

Solutions of Linear Systems

NMath 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.

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

var lu = new DoubleComplexLUFact( A );

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

// Solve for several right-hand sides.
var 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 Solutions

NMath 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 $a_1, a_2, \ldots, a_n$ such that $y = x_0 + x_1a_1 + \ldots + x_na_n$. A common goal of a least squares problem is to solve for the best values of $x_0, x_1, \ldots, x_n$. 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**:

```
var A = new FloatMatrix( "4x2[ 1 0  0 1  0 0  0 0 ]" );
var 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;
var rng = new RandGenUniform( -2, 2 );

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

var 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.

Encapsulating Univariate Functions

In **NMath**, 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:

```
var d = new Func<double, double>( MyFunction );

var 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`:

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

NMath 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:

```
OneVariabeFunction 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** 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.

Minimizing Univariate Functions

NMath includes classes for minimizing univariate functions using golden section search and Brent's method.

Minima of univariate functions must be *bracketed* before they can be isolated. A bracket is a triplet of points, $x_{lower} < x_{interior} < x_{upper}$, such that $f(x_{interior}) < f(x_{lower})$ and $f(x_{interior}) < f(x_{upper})$. These conditions ensure that there is some local minimum in the interval (x_{lower}, x_{upper}). If you know in advance that a local minimum falls within a given interval, you can simply call the **NMath** minimization routines using that interval. Before beginning minimization, the routine will search for an interior point that satisfies the bracketing condition. Otherwise, construct a **Bracket** object.

For example, the function:

$$y = (x - 1)^4$$

has a minimum at `1.0`. To compute the minimum, first encapsulate the function:

```
public static double MyFunction( double x )
{
  return Math.Pow( x - 1, 4 );
}

var f = new OneVariableFunction(
  new NMathFunctions.DoubleUnaryFunction( MyFunction ) );
```

This code constructs a **Bracket** starting from `(0,10)`, then finds a minimum of *f* using *Brent's Method*:

```
double tol = 1e-9;
int maxIter = 25;
var minimizer = new BrentMinimizer( tol, maxIter );
var bracket = new Bracket( f, 0, 10 );
double min = minimizer.Minimize( bracket );
```

NMath also includes class **DBrentMinimizer** which minimizes a univariate function using Brent's method in combination with evaluations of the first derivative. The sign of the derivative at the central point of the bracketing triplet is used to decide which region should be chosen for the next test point. For instance, the function:

$$y = (x - 5)^2$$

has a minimum at `5.0`.

To compute the minimum, first encapsulate the function and its derivative:

```
public static double MyFunction( double x )
{
  return ( ( x - 5 ) * ( x - 5 ) );
}

public static double MyFunctionPrime( double x )
{
  return ( 2 * x ) - 10;
}

var f = new OneVariableFunction(
  new NMathFunctions.DoubleUnaryFunction( MyFunction ) );
var df = new OneVariableFunction(
  new NMathFunctions.DoubleUnaryFunction( MyFunctionPrime ) );
```

This code constructs a **Bracket** starting from `(1,2)`, then computes the minimum:

```
var minimizer = new DBrentMinimizer();
var bracket = new Bracket( f, 1, 2 );
double min = minimizer.Minimize( bracket, df );
```

Root Finding

NMath also provides classes that find roots of univariate functions using the secant method, Ridders' method, and the Newton-Raphson method. For instance, this polynomial has a root at `1`:

$$f(x) = -2x^3 + 9x^2 - 5x - 2$$

This code finds the root in the interval `(0, 3)`:

```
var p =
  new Polynomial( new DoubleVector( -2.0, -5.0, 9.0, -2.0 ) );
var finder = new NewtonRaphsonRootFinder();
double lower = 0;
double upper = 3;
double root = finder.Find( p, p.Derivative(), lower, upper );
```

Encapsulating 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:

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

```
var 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:

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

This gives the exact derivative.

Fitting Polynomials

NMath provides class **PolynomialLeastSquares**, which performs a least squares fit of a **Polynomial** to a set of points. For example, if `x` and `y` are paired vectors of known *x*- and *y*-values, this code fits a cubic:

```
int degree = 3;
var fit = new PolynomialLeastSquares( degree, x, y );
```

Encapsulating Multivariate Functions

In **NMath**, class **MultiVariableFunction** encapsulates an arbitrary function of one or more variables, and works with other **NMath** classes to approximate integrals and minima. A **MultiVariableFunction** is constructed from a `Func<DoubleVector, double>`, a delegate that takes a single **DoubleVector** parameter and returns a `double`. For example, suppose you wish to encapsulate this function:

```
public double MyFunction( DoubleVector v )
{
     return ( NMathFunctions.Sum( v * v ) );
}
```

First, create a delegate for the `MyFunction()` method:

```
var d = new Func<DoubleVector, double>( MyFunction );
```

Then construct a **MultiVariableFunction** encapsulating the delegate:

```
var f = new MultiVariableFunction( d );
```

Integrating Multivariable Functions

NMath provides class **TwoVariableIntegrator**, which computes the integral of a function of two variables. Class **TwoVariableIntegrator** computes the double integral by breaking up the problem into repeated one-dimensional integrals. For example, to compute the double integral:

$$\int_0^1 \int_0^1 \frac{dx\,dy}{1 - x^2 y^2}$$

First write the function:

```
private double F( DoubleVector v )
{
  return 1.0 / ( 1.0 - ( v[0] * v[0] * v[1] * v[1] ) );
}
```

Then encapsulate the function as a **MultiVariableFunction**:

```
var function = new MultiVariableFunction(
  new NMathFunctions.DoubleVectorDoubleFunction( F ) );
```

Finally, compute the integral:

```
var integrator = new TwoVariableIntegrator();
double xLower = 0;
double xUpper = 1;
double yLower = 0;
double yUpper = 1;
double integral =
  integrator.Integrate( function, xLower, xUpper, yLower, yUpper );
```

Minimizing Multivariate Functions

NMath provides classes for minimizing multivariate functions using the downhill simplex method, Powell's direction set method, the conjugate gradient method, and the variable metric (or quasi-Newton) method.

For instance, if `f` is an encapsulated multivariate function of three variables, this code minimizes the function using the downhill simplex method, starting at the origin:

```
var minimizer = new DownhillSimplexMinimizer();
var start = new DoubleVector( 0.0, 0.0, 0.0 );
DoubleVector min = minimizer.Minimize( f, start );
```

Similarly, given the following function and partial derivatives:

```
protected static double MyFunction( DoubleVector v )
{
  return ( ( v[0] - 5.0 ) * ( v[0] - 5.0 ) ) +
         ( ( v[1] + 3.0 ) * ( v[1] + 3.0 ) );
}

protected static double MyFunctionDx( DoubleVector v )
{
  return ( 2 * v[0] ) - 10;
}

protected static double MyFunctionDy( DoubleVector v )
{
  return ( 2 * v[1] ) + 6;
}
```

This code computes the minimum using a **ConjugateGradientMinimizer**, starting at the origin:

```
var function = new MultiVariableFunction(
    new NMathFunctions.DoubleVectorDoubleFunction( MyFunction ) );

var partialx = new MultiVariableFunction(
  new NMathFunctions.DoubleVectorDoubleFunction( MyFunctionDx ) );
```

```
var partialy = new MultiVariableFunction(
  new NMathFunctions.DoubleVectorDoubleFunction( MyFunctionDy ) );
var df = new MultiVariableFunction[] { partialx, partialy };

var minimizer = new ConjugateGradientMinimizer();
var start = new DoubleVector( 2, 0 );
DoubleVector min = minimizer.Minimize( f, df, start );
```

Simulated Annealing

In **NMath**, class **AnnealingMinimizer** minimizes a multivariable function using the simulated annealing method. Simulated annealing is based on an analogy from materials science. To produce a solid in a low energy state, such as a perfect crystal, a material is often first heated to a high temperature, then gradually cooled.

In the computational analogy of this method, a function is iteratively minimized with an added random temperature term. The temperature is gradually decreased according to an annealing schedule, as more optimizations are applied, increasing the likelihood of avoiding entrapment in local minima, and of finding the global minimum of the function.

For example, the annealing schedule shown in Table 1 has four steps.

Table 1 – A sample annealing schedule

Step	Temperature	Iterations
1	100	20
2	75	20
3	50	20
4	0	20

In this case, the temperature decays linearly from `100` to `0`, and the same number of iterations are performed at each step. Class **LinearAnnealingSchedule** encapsulates the linear decay of a starting temperature to zero. Each step has a specified number of iterations. For example, this code creates the annealing schedule shown in Table 1:

```
int steps = 4;
int iterationsPerStep = 20
double startTemp = 100.0;

var schedule = new LinearAnnealingSchedule( steps,
  iterationsPerStep, startTemp );
```

NOTE—For more control over the temperature decay, you can use class CustomAnnealingSchedule.

Instances of **AnnealingMinimizer** are constructed from an annealing schedule. For instance:

```
var schedule = new LinearAnnealingSchedule( 5, 25, 100.0 );;
var minimizer = new AnnealingMinimizer( schedule );
```

Once constructed, an **AnnealingMinimizer** provides a `Minimize()` method that takes a **MultiVariableFunction** to minimize, and a starting point. For instance, if `f` is an encapsulated multivariable function of five variables, this code minimizes the function using the downhill simplex method, starting at `(0.2,0.2,-.2,0.0,0.0)`:

```
var minimizer = new AnnealingMinimizer( schedule );
var start = new DoubleVector( 0.2, 0.2, -0.2, 0.0, 0.0 );
DoubleVector min = minimizer.Minimize( f, start );
```

Class **AnnealingMinimizer** can also be configured to keep a history of the annealing process. When history is turned on, the results of each step are recorded in an **AnnealingHistory** object. This data may be useful when adjusting the schedule for optimal performance. For example, this code prints out the complete history after a minimization:

```
var minimizer = new AnnealingMinimizer( schedule );
minimizer.KeepHistory = true;

DoubleVector min = minimizer.Minimize( f, startingPoint );
AnnealingHistory history = minimizer.AnnealingHistory;
Console.WriteLine( history );
```

Linear, Nonlinear, and Quadratic Programming

NMath provides classes for linear programming (LP), non-linear programming (NLP), and quadratic programming (QP) using the Microsoft Solver Foundation.

For example, class **InteriorPointQPSolver** solves QP problems using an interior point algorithm.

```
var solver = new InteriorPointQPSolver();
```

Parameters are specified using an instance of **InteriorPointQPSolverParams**.

```
var solverParams = new InteriorPointQPSolverParams
{
  KktForm = InteriorPointQPSolverParams.KktFormOption.Blended,
  Tolerance = 1e-6,
```

```
  MaxDenseColumnRatio = 0.9,
  PresolveLevel =
    InteriorPointQPSolverParams.PresolveLevelOption.Full,
  SymbolicOrdering = InteriorPointQPSolverParams.
    SymbolicOrderingOption.ApproximateMinDegree
};
```

This code performs the actual solve and prints out the results:

```
solver.Solve( problem, solverParams );

Console.WriteLine( "Solver Parameters:" );
Console.WriteLine( solverParams.ToString() );
Console.WriteLine( "\nResult = " + solver.Result );
Console.WriteLine( "Optimal x = " + solver.OptimalX );
Console.WriteLine( "Optimal Function value = " +
  solver.OptimalObjectiveFunctionValue );

Console.WriteLine( "iterations = " + solver.IterationCount );
```

Nonlinear Least Squares

NMath provides classes for solving nonlinear least squares problems:

- **TrustRegionMinimizer** solves both constrained and unconstrained nonlinear least squares problems using the Trust Region method.
- **LevenbergMarquardtMinimizer** solves nonlinear least squares problems using the Levenberg-Marquardt method.
- **OneVariableFunctionFitter** fits generalized one variable functions to data, by finding a minimum in the curve parameter space in the sum of the squared residuals with respect to a set of data points.
- **MultiVariableFunctionFitter** fits generalized multivariable functions to data, by finding a minimum in the surface parameter space in the sum of the squared residuals with respect to a set of data points.

For example, this code fits a predefined four parameter logistic function (4PL) to a set of points:

```
NMathFunctions.GeneralizedDoubleUnaryFunction f =
  AnalysisFunctions.FourParameterLogistic;

var fitter = new OneVariableFunctionFitter( f );

var x = new DoubleVector( 0.00, 0.00, 0.00, 0.00, 0.00,
                          0.00, 0.94, 0.94, 0.94, 1.88,
```

```
                            1.88, 1.88, 3.75, 3.75, 3.75,
                            7.50, 7.50, 7.50, 15.00, 15.00,
                            15.00, 30.00, 30.00, 30.00 );

var y = new DoubleVector( 7.58, 8.00, 8.32, 7.25, 7.37,
                            7.96, 8.35, 6.91, 7.75, 6.87,
                            6.45, 5.92, 1.92, 2.88, 4.23,
                            1.18, 0.85, 1.05, 0.68, 0.52,
                            0.82, 0.25, 0.22, 0.44 );

var start = new DoubleVector( "0.1 0.1 0.1 0.1" );

DoubleVector solution = fitter.Fit( x, y, start );
```

A selection of common generalized functions is provided as a convenience. Of course, you can also define your own functions to fit.

Differential Equations

NMath provides classes for solving stiff and non-stiff first order initial value differential equations. An *ordinary differential equation* (ODE) contains one or more derivatives of a dependent variable *y* with respect to a single independent variable. A *first-order* ODE contains only the first derivative of *y*. Since there are generally many functions that satisfy an ODE, an *initial value* is necessary to constrain the solution—that is, *y* is equal to *y0* at a given initial *x0*.

Class **FirstOrderInitialValueProblem** encapsulates a first order initial value differential equation.

```
Func<double, double, double> f =
  delegate( double x, double y )
  {
    return x * x;
  };
double x0 = 0.0;
double y0 = 1.0;

var prob = new FirstOrderInitialValueProblem( f, x0, y0 );
```

For example, class **RungeKuttaSolver** solves first order initial value differential equations by the Runge-Kutta method. Instances of **RungeKuttaSolver** are constructed from the number of tabulated points and a nonzero value *delta*. From the number of points and the delta, the set $\{x_i\}$ is determined as $x_i = x_0 + i\Delta$ for i=0,1,…, n .

```
int n = 2000;
double delta = .001;
var solver = new RungeKuttaSolver( n, delta );
TabulatedFunction f = null;
solver.Solve( prob, ref f );
```

Special Functions

NMath provides class **SpecialFunctions** for functions such factorial, binomial, the gamma function and related functions, Bessel functions, elliptic integrals, and many more. These functions cover many of the most commonly needed functions in physics and engineering. For example:

```
// Compute the Jacobi function Sn() with a complex argument.
var cmplx = new DoubleComplex( 0.1, 3.3 )
DoubleComplex sn = SpecialFunctions.Sn( cmplx, .3 );

// Compute the elliptic integral, K(m)
double ei = SpecialFunctions.EllipticK( 0.432 );
```

ADO.NET Integration

NMath provides convenience methods for creating ADO.NET objects from vectors and matrices, and for creating vectors and matrices from database objects. Real-value **NMath** 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**:

```
var A = new DoubleMatrix( 8, 5, 3.1415 );
var table = A.ToDataTable( "MyMatrixTable" );
```

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

Serialization

NMath 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** objects.

For instance, this code serializes two **FloatComplexMatrix** objects to a file in binary format:

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

FileStream binStream = File.Create( "myData.dat" );
var 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" );

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

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

NMath Statistical Library

NOTE—Starting with the release of NMath 7.0, NMath includes all of the statistical routines previously available in the NMath Stats product. The NMath Stats product is no longer available separately.

Data Frames

The statistical functions in **NMath** support the types **DoubleVector** and **DoubleMatrix**, as well as simple arrays of doubles. In many cases, these types are sufficient for storing and manipulating your statistical data. However, they suffer from two limitations: they can only store numeric data and they have limited support for adding, inserting, removing, and reordering data. Because the underlying data is an array of doubles, data must be copied to new storage every time manipulation operations such as these are performed.

For these reasons, **NMath** provides the **DataFrame** class which represents a two-dimensional data object consisting of a list of columns of the same length. Columns are themselves lists of different types of data: numeric, string, boolean, generic, and so on.

A **DataFrame** can be viewed as a kind of virtual database table. Columns can be accessed by numeric index (`0...n-1`) or by a string name supplied at construction time. Rows can be accessed by numeric index (`0...n-1`) or by a key object.

Data frames can be constructed in a variety of ways. The default constructor creates an empty data frame with no rows or columns. Columns and rows can then be added to the new data frame. For example:

```
var df = new DataFrame();

// Add some columns
df.AddColumn( new DFStringColumn( "Sex" ) );
df.AddColumn( new DFStringColumn( "AgeGroup" ) );
df.AddColumn( new DFNumericColumn( "Weight" ) );

// Add some rows
df.AddRow( "John Smith", "M", "Child", 45 );
df.AddRow( "Ruth Barnes", "F", "Senior", 115 );
df.AddRow( "Jane Jones", "F", "Adult", 115 );
df.AddRow( "Timmy Toddler", "M", "Child", 42 );
df.AddRow( "Betsy Young", "F", "Adult", 130 );
df.AddRow( "Arthur Smith", "M", "Senior", 142 );
df.AddRow( "Lucy Doe", "F", "Child", 30 );
```

This data frame contains three columns: column `0`, named `Sex`, contains string data; column `1`, named `AgeGroup`, also contains string data; column `2`, named `Weight`, contains numeric data. There are seven rows of data in this data frame, and the subjects' names are used as row keys.

Methods are provided for appending, inserting, removing, sorting, and permuting rows and columns in a data frame. This code manipulates a data frame:

```
// switch last two columns
df.PermuteColumns( 0, 2, 1 );
```

```
// sort rows primarily by AgeGroup in ascending order,
// and secondarily by Sex in descending order
int[] colIndices = { 2, 0 };
SortingType[] sortingTypes = { SortingType.Ascending,
                               SortingType.Descending };
df.SortRows( colIndices, sortingTypes );

// delete a row by key
df.RemoveRow( "Lucy Doe" );

// export to a ADO.NET DataTable
DataTable dt = df.ToDataTable();
```

Because the underlying data is in a list, elements can be added, removed, and reordered without having to copy all of the data to new storage.

Subsets

In addition to accessors for individual elements, columns, or rows in a data frame, class **DataFrame** provides a large number of indexers and member functions for accessing sub-frames containing any arbitrary subset of rows, columns, or both.

Such indexers and methods accept the types **Slice** and **Range** to indicate sets of row or column indices with constant spacing. In addition, **NMath** introduces a class called **Subset**. Like a **Slice** or **Range**, a **Subset** represents a collection of indices that can be used to view a subset of data from another data structure. Unlike a **Slice** or **Range**, however, a **Subset** need not be continuous, or even ordered. It is simply an arbitrary collection of indices.

For example, this code gets a new data frame containing columns `3-8` in reverse order, and all rows where the value in column `0` in greater than the value in column `1`:

```
var colRange = new Range( 8, 3, -1 );

var bArray = new bArray[ df.Rows ];
for ( int i = 0; i < df.Rows; i++ )
{
  bArray[i] = ( df[0][i] > df[1][i] );
}
var rowSubset = new Subset( bArray );

DataFrame df2 = df[ rowSubset, colRange ];
```

This code utilizes a very useful **Subset** constructor which takes an array of boolean values and constructs a **Subset** containing the indices of all `true` elements in the array.

Factors

The **Factor** class represents a categorical vector in which all elements are drawn from a finite number of factor levels. Thus, a **Factor** contains two parts: an object array of factor levels, and an integer array of categorical data, of which each element is an index into the array of levels. For example, this string data:

```
"A", "A", "C", "B", "A", "C", "B"
```

could be presented as a **Factor** with the following levels and categorical data:

```
object[] levels = { "A", "B", "C" };
int[] data = { 0, 0, 2, 1, 0, 2, 1 };
```

Factors are usually constructed from a data frame column using the `GetFactor()` method, which creates a **Factor** with levels for the sorted, unique values in the column.

The principal use of factors is in conjunction with the `GetGroupings()` methods on **Subset**. One overload of this method accepts a single **Factor** and returns an array of subsets containing the indices for each level of the given factor. Another overload accepts two **Factor** objects and returns a two-dimensional jagged array of subsets containing the indices for each combination of levels in the two factors.

For example, this code constructs factors from the values in the `Sex` and `AgeGroup` columns of a data frame, then uses these factors in conjunction with the `GetGroupings()` methods on **Subset** to create subsets representing the groups for each level of the factors, as well as all combinations of the factors:

```
Factor sex = df.GetFactor( "Sex" );
Factor age = df.GetFactor( "AgeGroup" );

Subset[] sexGroups = Subset.GetGroupings( sex );
Subset[] ageGroups = Subset.GetGroupings( age );
Subset[,] cellGroups = Subset.GetGroupings( sex, age );
```

These subsets can then be used to operate on the relevant portions of the data frame, as described above.

Descriptive Statistics

Class **StatsFunctions** provides a wide variety of static functions for computing descriptive statistics, such as mean, variance, standard deviation, percentile, median, quartiles, geometric mean, harmonic mean, RMS, kurtosis, skewness, and many more. Method overloads accept data as an array of doubles, as a **DoubleVector**, or as a numeric column in a **DataFrame**. For example:

```
double[] dblArray = { 1.12, -2.0, 3.88, 1.2, 15.345 };
double mean1 = StatsFunctions.Mean( dblArray );

var v = new DoubleVector( "1.12 -2.0 3.88 1.2 15.345"  );
double mean2 = StatsFunctions.Mean( v );

var df = new DataFrame();
df.AddColumn(
  new DFNumericColumn( "myData", 1.12, -2.0, 3.88, 1.2, 15.345 ) );
double mean3 = StatsFunctions.Mean( df[ "myData" ] );
```

Most functions in class **StatsFunctions** are accompanied by a paired function which ignores values that are Not-a-Number (NaN). For example, there are `Mean()` and `NaNMean()` functions, `Variance()` and `NaNVariance()` functions, and so forth. Unless a function is explicitly designed to handle missing values, it may return NaN or have unexpected results if values are missing.

Probability Distributions

NMath provides classes for computing the probability density function (PDF), the cumulative distribution function (CDF), the inverse cumulative distribution function, and random variable moments for a variety of probability distributions, including normal (Gaussian), Poisson, chi-square (χ^2), gamma, beta, Student's *t*, *F*, binomial, and negative binomial. For example, this code constructs an *F* distribution object with degrees of freedom `11,19`, then queries it:

```
int df1 = 11;
int df2 = 19;
var dist = new FDistribution( df1, df2 );

double pdf = dist.PDF( 1.45 );
double cdf = dist.CDF( 1.45 );
double invCdf = dist.InverseCDF( .95 );
double mean = dist.Mean;
double var = dist.Variance;
double kurt = dist.Kurtosis;
double skew = dist.Skewness;
```

Hypothesis Tests

NMath provides classes for many common hypothesis tests, such as the z-test, t-test, F-test, and Kolmogorov-Smirnov test, with calculation of p-values, critical values, and confidence intervals.

For example, class **OneSampleZTest** determines whether a sample from a normal distribution with known standard deviation could have a given mean. If the sample data is vectors `data`, this code constructs a hypothesis test object:

```
double mu0 = 100;
double sigma = 15;
var test = new OneSampleZTest( data, mu0, sigma );
```

By default, a **OneSampleZTest** object performs a two-sided hypothesis test ($H_1: \mu \neq \mu_0$) with $\alpha = 0.01$. Non-default test parameters can be specified at the time of construction using constructor overloads, or after construction using the provided `Type` and `Alpha` properties:

```
test.Type = HypothesisType.Right;
test.Alpha = 0.05;
```

All hypothesis test classes provide a `ToString()` method that returns a formatted string representation of the test results:

```
One Sample Z Test
-----------------

Sample mean = 112.8
Sample size = 9
Population mean = 100
Population standard deviation = 15
Computed Z statistic: 2.56

Hypothesis type: one-sided to the right
Null hypothesis: sample mean = population mean
Alt hypothesis: sample mean > population mean
P-value: 0.00523360816355578
REJECT the null hypothesis for alpha = 0.05
0.95 confidence interval: 104.575731865243 Infinity
```

Properties are also provided for accessing individual elements in the test results.

Linear Regression

Class **LinearRegression** computes a multiple linear regression from an input matrix of independent variable values (the *predictor matrix* or *regression matrix*) and a vector of dependent variable values (the *observation vector*). For example:

```
var predictors =
  new DoubleMatrix( " 8x4 [ 1 1450 .50 70
                            1 1600 .50 70
                            1 1450 .70 70
                            1 1600 .70 70
                            1 1450 .50 120
                            1 1600 .50 120
                            1 1450 .70 120
                            1 1600 .70 120 ]" );
var obs = new DoubleVector( "[ 67 79 61 75 59 90 52 87 ]" );
var lr = new LinearRegression( A, obs );
```

By default, model parameter values are computed by the *method of least squares* using a QR factorization, but you may elect to use a complete orthogonal factorization or singular value decomposition instead.

The *y*-intercept is the first element of the parameter array returned by the regression, and the slope is the second:

```
Console.WriteLine( "y-intercept = {0}", regression.Parameters[0] );
Console.WriteLine( "Slope = {0}", regression.Parameters[1] );
```

You can also use a linear regression object to generate predictions:

```
var predictors = new DoubleVector( 150.0, 33.5, 0.66, 80.0 );

double predicted = lr.PredictedObservation( predictors );
```

NMath also provides the **LinearRegressionParameter** class for testing statistical hypotheses about individual parameters in a **LinearRegression**. This code tests whether the fifth parameter in a model is significantly different than zero:

```
var param = new LinearRegressionParameter( lr, 4 );
double tstat = param.TStatistic( 0.0 );
double pValue = param.TStatisticPValue( 0.0 );
double criticalValue = param.TStatisticCriticalValue( 0.05 );
Interval confidenceInterval = param.ConfidenceInterval( 0.05 );
```

Class **LinearRegressionAnova** tests the overall model significance for linear regressions:

```
var lrAnova = new LinearRegressionAnova( lr );
double sse = lrAnova.ResidualSumOfSquares;
double r2 = lrAnova.RSquared;
double fstat = lrAnova.FStatistic;
double fstatPval = lrAnova.FStatisticPValue;
```

Analysis of Variance

NMath provides classes for both *one-way* (one-factor) and *two-way* (two-factor) ANOVAs. One-way ANOVA is supported for both balanced and unbalanced designs, and with or without repeated measures (RANOVA). Two-way ANOVA is supported for balanced designs only, with or without repeated measures.

For example, this code constructs a two-way ANOVA by grouping the numeric data in column `3` of **DataFrame** `df` by factors constructed from columns `0` and `4`:

```
var anova = new TwoWayAnova( df, 0, 4, 3 );
```

Once you've constructed an ANOVA object, you can display the complete ANOVA table:

```
Console.WriteLine( anova );
```

For example:

```
Source  Deg of Freedom  SumOfSq    Mean Square  F         P
FactorA       1          1782.0450  1782.0450    14.2121   0.0008
FactorB       1          2838.8113  2838.8113    22.6399   0.0001
Interaction   1          108.0450   108.0450     0.8617    0.3612
Error         28         3510.9075  125.3896     .         .
Total         31         8239.8088  .
```

Properties are provided for accessing individual elements in the ANOVA table.

NMath solves the two-way ANOVA problem using multiple linear regression. If all you wish to know is the information in the standard ANOVA table, you can safely ignore the regression details, but properties and member functions are provided for retrieving information about the underlying regression parameters.

Multivariate Statistics

NMath provides classes for dimension reduction using *principal component analysis* (PCA) and *factor analysis*, and case reduction using *hierarchical cluster analysis* and *k-means cluster analysis*. Principal component analysis finds a smaller set of synthetic variables that capture the variance in an original data set. The first principal component accounts for as much of the variability in the data as possible, and each succeeding orthogonal component accounts for as much of the remaining variability as possible. A **FloatPCA** or **DoublePCA** instance is constructed from a matrix or a dataframe containing numeric data. Each column represents a variable, and each row represents an observation. The data may optionally be zero-centered and scaled to have unit variance.

```
bool center = true;
bool scale = true;
```

```
var pca = new DoublePCA( data, center, scale );
Console.WriteLine( "Loading Martrix = " + pca.Loadings );
Console.WriteLine( "Variance Proportions = " +
                   pca.VarianceProportions );
Console.WriteLine( "Cumulative Variance Proportions = " +
                   pca.CumulativeVarianceProportions );
Console.WriteLine( "Scores = " + pca.Scores );
```

Hierarchical cluster analysis detects natural groupings in data. Each object is initially assigned to its own singleton cluster. The analysis then proceeds iteratively, at each stage joining the two most similar clusters into a new cluster, continuing until there is one overall cluster.

During clustering, the distance between individual objects is computed using a distance function delegate. Delegates are provided as static variables on class **Distance** for euclidean, squared euclidean, city-block (Manhattan), maximum (Chebychev), and power distance functions. You can also create your own distance function delegate. The distances between clusters of objects are computed using a linkage function delegate. Delegates are provided as static variables on class **Linkage** for single, complete, unweighted average, weighted average, centroid, median, and Ward's linkage functions. Again, you can also create your own linkage function delegate. For example, this code clusters `8` random vectors of length `3`:

```
var data = new DoubleMatrix( 8, 3, new RandGenUniform() ); var ca =
new ClusterAnalysis( data, Distance.SquaredEuclideanFunction,
  Linkage.CompleteFunction );
```

Property `Distances` gets the vector of distances between all possible object pair; `Linkages` gets the complete hierarchical linkage tree. The `CutTree()` method constructs a set of clusters by cutting the hierarchical linkage tree either at the specified height, or into the specified number of clusters.

```
Console.WriteLine( ca.Distances );
Console.WriteLine( ca.Linkages );

// cut linkage tree to form 3 clusters Console.WriteLine(
ca.CutTree( 3 ) );

// cut linkage tree at height of 0.75 Console.WriteLine(
ca.CutTree( 0.75 ) );
```

Nonnegative Matrix Factorization

Nonnegative matrix factorization (NMF) factors a matrix *V* into two matrices, *W* and *H*. NMF differs from many other factorizations by enforcing the constraint that the factors *W* and *H* must be non-negative—that is, all elements must be equal to or greater than zero.

If a set of *n*-dimensional *m* data vectors are placed in an *n* x *m* matrix *V*, then NMF can be used to approximately factor *V* into an *n* x *r* matrix *W* and an *r* x *m* matrix *H*. Usually *r* is chosen to be much smaller than either *m* or *n*, so that *W* and *H* are smaller than the original matrix *V*. Thus, each column *v* of *V* is approximated by a linear combination of the columns of *W*, with the coefficients being the corresponding column of *H*, $v \approx Wh$. This extracts underlying features of the data as basis vectors in *W*, which can then be used for identification and classification. By not allowing negative entries in *W* and *H*, NMF enables a non-subtractive combination of the parts to form a whole.

NMath provides classes for **NMFact** for basic NMF, and **NMFClustering** for data clustering using NMF. For example, the following code clusters data in a **DoubleMatrix** using NMF, and prints the results:

```
DoubleMatrix data = ...   // data to be factored
int k = ...               // number of columns in W

var nmfClustering = new NMFClustering<NMFDivergenceUpdate>();

nmfClustering.Factor( data, k );

ClusterSet cs = nmfClustering.ClusterSet;

// Print out the cluster each column belongs to
for ( int i = 0; i < cs.N; i++ ) {
  Console.WriteLine( "Column {0} belongs to cluster {1}",
    i, cs[i] );
}

// Print out the the members of each cluster
for ( int i = 0; i < cs.NumberOfClusters; i++ ) {
  int[] members = cs.Cluster( i );
  Console.Write( "Cluster number {0} contains: ", i );
  for ( int j = 0; j < members.Length; j++ ) {
    Console.Write( "{0} ", j );
  }
  Console.WriteLine();
}
```

Since NMF uses random starting values for *W* and *H*, and the factorization is not unique, you can get different clusterings for the columns of V on different runs. A *consensus matrix* is a way to average multiple clusterings, to produce a probability estimate that any pair of columns will be clustered together. **NMath** provides class **NMFConsensusMatrix** for compute a consensus matrix using NMF.

Partial Least Squares

Partial Least Squares (PLS) is a technique that generalizes and combines features from principal component analysis and multiple linear regression. It is particularly useful when you need to predict a set of response (dependent) variables from a large set of predictor (independent variables). **NMath** supports both the Nonlinear Iterative PArtial Least Squares (NIPALS) and Straightforward IMplementation of Partial Least Squares (SIMPLS) algorithms.

NMath provides two classes for performing PLS regression:

- **PLS1** is used when the responses, Y, in the model $Y=XB+E$ consist of a single variable. In this case Y is a vector containing the n response values.
- **PLS2** is used when the responses are multivariate. In this case Y is a matrix composed of n rows with each row containing the m response variable values.

Computing a PLS regression is accomplished by simply constructing a **PLS1** or **PLS2** instance. The basic parameters are:

- the matrix of predictor variables values
- the response variable values (a vector for **PLS1** and a matrix for **PLS2**)
- an integer specifying the number of factors or components

For example:

```
DoubleMatrix A = ...
DoubleVector y = = ...
int numComponents = 3;

var pls = new PLS1( A, y, numComponents );
```

NMath also provides the classes **PLS1Anova** and **PLS2Anova** for performing a classic ANOVA for **PLS1** and **PLS2** regression models. These classes calculate the sum of squares total, sum of squares residual, mean square error for prediction, and the coefficient of determination.

Conclusions

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

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

.NET Numerical Applications with NMath

The correct bibliographic reference for this document is:
.NET Numerical Applications with NMath, CenterSpace Software, Corvallis, OR.

Printed in the United States.
Printing Date: September, 2019

CenterSpace Software

Address:	622 NW 32nd St., Corvallis, OR 97330 USA
Phone:	(541) 896-1301
Web:	http://www.centerspace.net
Technical Support:	support@centerspace.net