NMath
User's Guide

Version 7.2

CenterSpace™
Software

NMATH USER'S GUIDE

The correct bibliographic reference for this document is:
NMath User's Guide, *Version 7.2*, CenterSpace Software, Corvallis, OR.

Printed in the United States.
Printing Date: January, 2021

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

CONTENTS

Part I - Introduction

Part II - NMath Core

Part IV - Analysis

Part V - Statistics

PART I - INTRODUCTION

CHAPTER 1.
OVERVIEW

Welcome to the *NMath User's Guide*.

CenterSpace Software's **NMath**™ numerical library provides object-oriented components for mathematical, engineering, scientific, and financial applications on the .NET platform. **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, and F#.

For most 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.

1.1 Product Components

All **NMath** types are organized into a single namespace for simplicity.

- `CenterSpace.NMath.Core`

Prior to **NMath** 7.0 the library was organized in four namespaces: Core, Matrix, Analysis, and Stats. Although these namespaces can still be included for backward compatibility, they all now simply forward to the single `CenterSpace.NMath.Core` namespace.

To avoid using fully qualified names, preface your code with the namespace statement.

Code Example – C#

```
using CenterSpace.NMath.Core;
```

Code Example – VB

```
Imports CenterSpace.NMath.Core
```

All **NMath** code shown in this manual assumes the presence of such namespace statements.

1.2 Software Requirements

NMath requires the following additional software to be installed on your system:

- To use the **NMath** library, you need the Microsoft .NET Framework, .NET 5, or .NET Core installed on your system. These frameworks are available without cost from:

  `https://dotnet.microsoft.com/download/`

- Use of Microsoft Visual Studio .NET (or other .NET IDE) is strongly encouraged. However, the .NET Framework includes command line compilers for .NET languages, so an IDE is not strictly required.

- Viewing PDF documentation requires Adobe Acrobat Reader, available without cost from:

  `http://www.adobe.com`

The Intel® Math Kernel Library (MKL) is included with **NMath**. You do not need to provide your own version.

1.3 NMath Assemblies

The **NMath** installer places the following .NET assemblies in directory *<installdir>*`/Assemblies`:

- `NMath.dll`, the main **NMath** assembly
- `System.Configuration.ConfigurationManager.dll` (>= 4.6.0)

Native assemblies are placed in architecture-specific subdirectories.

<installdir>`/Assemblies/x86`

- `nmath_native_x86.dll`, 32-bit native code, including the Intel® Math Kernel Library (MKL)
- `nmath_sf_x86.dll`, 32-bit native code for special functions
- `libiomp5md.dll`, dynamically-linked 32-bit Intel OMP threading library

<installdir>`/Assemblies/x64`

- `nmath_native_x64.dll`, 64-bit native code, including Intel® Math Kernel Library (MKL)
- `nmath_sf_x64.dll`, 64-bit native code for special functions

- `libiomp5md.dll`, dynamically-linked 64-bit Intel OMP threading library

The installer also places the .NET assemblies in your global assembly cache (GAC). The native DLLs are linked resources to the corresponding kernel assemblies.

Microsoft Solver Foundation

NMath nonlinear programming, and quadratic programming classes are built on the Microsoft Solver Foundation (MSF). The Standard Edition of MSF is included with **NMath** (`Microsoft.Solver.Foundation.dll`), but is limited to 100,000 non-zero coefficients. Note that this is not a limit on the number of variables, but rather on the total number of all non-zero coefficients used to specify the constraints. Given `n` variables and `m` constraints, there are between `0` and `m*n` non-zero coefficients.

Google OR Tools

NMath linear programming and mixed integer programming classes are built on the Google OR Tools libraries. This library is included with **NMath** (`Google.OrTools.dll`), and has no artificially imposed limits on the number of constraints or variables.

1.4 NMath License Key

NMath license information is stored in a license key which must be found at runtime. The license key governs the properties of your **NMath** installation.

Evaluation License

If no license key is found, a default evaluation license key is used which provides a free 30-day evaluation period for **NMath** on the current machine.

Product License

You can specify your license key using various mechanisms: by environment variable, by configuration app setting, and programmatically. These mechanisms may be preferable in group development environments, and at deployment. (See Section 1.5 for more information.)

1.5 NMath Configuration

NMath configuration settings govern the loading of the **NMath** license key and native library.

Property values can be set three ways:

1. by environment variable
2. by configuration setting
3. by programmatically setting properties on class **NMathConfiguration**

NOTE—Settings are applied in that order, and resetting a property takes precedent over any earlier values.

For example, here an environment variable is used:

```
NMATH_NATIVE_LOCATION="C:\temp"
```

This code uses an application configuration file:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="NMathNativeLocation" value="C:\temp" />
  </appSettings>
</configuration>
```

Configuration settings may also be placed in a DLL configuration file placed next to the main **NMath** assembly (`NMath.DLL.config`, for example). If a setting is specified in both a DLL and an application configuration file, the application configuration takes precedence.

This code accomplishes the same thing programmatically:

```
NMathConfiguration.NativeLocation = @"C:\temp";
```

The supported environment variables, configuration app setting keys, and property names are show in Table 1.

Table 1 – Configuration Properties

Environment Variable	Configuration Setting	Property or Method
`NMATH_LOG_FILENAME`	`NMathLogFilename`	`LogFilename`
`NMATH_LOG_LOCATION`	`NMathLogLocation`	`LogLocation`
`NMATH_LICENSE_KEY`	`NMathLicenseKey`	`LicenseKey`

Table 1 – Configuration Properties

Environment Variable	Configuration Setting	Property or Method
`NMATH_NATIVE_LOCATION`	`NMathNativeLocation`	`NativeLocation`
`NMATH_MKL_NUM_THREADS`	`NMathMKLNumThreads`	`SetMKLNumThreads()`
`NMATH_REPRODUCIBILITY`	`NMathReproducibility`	`Reproducibility`

NOTE—Assembly loading and license checking is normally performed the first time you make an NMath call. If you wish to explicitly control when these operations occur—at application start-up, for example—use the static NMathConfiguration.Init() method.

Logging

To debug configuration issues, specify a log file location. For example, setting the property programmatically:

```
NMathConfiguration.LogLocation = @"C:\temp";
```

NOTE—The specified location must exist.

Setting a log file location turns on logging at that location, using the currently defined log filename (`NMathConfiguration.log`, unless previously modified).

To turn off logging, set the log location to `null`.

For verbose logging, such as all native function calls, set `NMathConfiguration.LogVerbose` to `true`.

License Key

You can specify your **NMath** license key using the `NMathConfiguration.LicenseKey` property, or the equivalent environment variable or app config setting.

Native Location

The **NMath** native libraries must be found at runtime (Section 1.3). Failure to locate these files is one of the most common configuration issues, especially in deployment. The search order is determined by your `PATH`. Some standard locations are automatically prepended to your (process-specific) `PATH`. You can also use the `NMathConfiguration.NativeLocation` property, or the equivalent environment variable or app config setting, to prepend another location. An architecture-specific `/x86` and `/x64` subdirectory is also prepended. The appropriate architecture-specific natives are loaded at runtime.

MKL Threading Control

MKL contains highly optimized, extensively threaded math routines. In rare cases, these can cause conflicts between the Intel OMP threading library (`libiomp5md.dll`) and the .NET threading model. If your .NET application is itself highly multi-threaded, you may wish to use MKL in single-threaded mode. Set the suggested number of threads to `1` using the `SetMKLNumThreads()` method, or use the equivalent environment variable or app config setting.

NOTE—MKL does not always have a choice on the number of threads for certain reasons, such as system resources. Although Intel MKL may actually use a different number of threads from the number suggested, this method enables you to instruct the library to try using the suggested number when the number used in the calling application is unavailable.

MKL Conditional Numerical Reproducibility (CNR)

For general single and double precision IEEE floating-point math, the order of computation matters. For example, in infinite precision arithmetic, the associative property holds, `(a+b)+c = a+(b+c)`, but on a computer using double precision floating-point numbers, rounding error is introduced, and the equality is not guaranteed. The order of floating-point operations within a single executable program is affected by code-path selection based on a variety of factors: run-time processor dispatching, data array alignment, variation in number of threads, threaded algorithms, and so forth.

If strict reproducibility is a requirement, set the `Reproducibility` property equal to `true`, or use the equivalent environment variable or app config setting. You must also set the suggested number of MKL threads to a constant value (see above).

For more information, see

`https://software.intel.com/en-us/articles/introduction-to-the-conditional-numerical-reproducibility-cnr`

NOTE—Using MKL Conditional Numerical Reproducibility can significantly degrade performance, and is only recommended for use during testing or debugging, such as comparison to previous 'gold' results.

1.6 Building and Deploying NMath Applications

To use **NMath** types in your application, add a reference to `NMath.dll`. The search order is the same as for the common language runtime: first the GAC is searched, then the directory containing the currently executing assembly, and so on. (See Section 1.5 for more information.)

We recommend that you build your application using either the `x86` or `x64` build configuration (depending on which NuGet package is being used), so you can deploy to either 32-bit or 64-bit environments. Also note that if you are building for .NET 4.5 or higher and targeting x64, ensure that the `Prefer 32-bit` flag is unchecked under **Build | Platform target** in your project properties.

To deploy your application, either

- Install the **NMath** .NET assemblies in the GAC (`NMath.dll`)—the appropriate native DLLs will also be placed in the GAC since they are linked resources; or
- Place the main .NET assembly (`NMath.dll`) in the same directory as your application. Use the `NativeLocation` property, or the equivalent environment variable or app config setting, to specify the location of the native assemblies (Section 1.5). The specified location should contain `/x86` and `/x64` subdirectories. The appropriate architecture-specific natives are loaded at runtime.

If your application fails to locate the native assemblies at runtime, enable configuration logging (Section 1.5), which will provide information on the search path.

License Key

A valid license key must accompany your deployed **NMath** code. The key can be specified using an environment variable, app config setting, or compiled in your code for greatest security. (See Section 1.5 for more information.)

C++ Runtime

NMath has a dependency on the Microsoft Visual C++ 2017 runtime. The **NMath** installer places the C++ runtime on your development machine, if necessary. However, when you deploy your application, you may need to add it to your installer.

There are two ways to do this:

- Add the Microsoft Visual C++ 2017 merge module to your installer. It can be found here:

  x86: `C:\Program Files (x86)\Common Files\Merge Modules\Microsoft_VC120_CRT_x86.msm`

  x64: `C:\Program Files (x86)\Common Files\Merge Modules\Microsoft_VC120_CRT_x64.msm`

  or on the web.

- Use the Microsoft Visual C++ 2017 redistributable:

  `https://support.microsoft.com/en-us/help/2977003/the-latest-supported-visual-c-downloads`

  Note: Visual C++ 2015, 2017 and 2019 all share the same redistributable files.

1.7 Web Applications

You can create ASP.NET web applications using **NMath**, just like any other .NET application. However, there are a few additional considerations for building and deploying ASP.NET applications.

Referencing NMath

To use **NMath** types in your web application, add a reference to `NMath.dll`, just as you would with other types of .NET applications. If you are using web projects in Visual Studio, you can simply right-click the **References** folder, and select the **Add Reference...** command. If you specify **Copy Local** equals `true` in the reference's properties, then the assembly will be copied to the `/bin` directory of the web application, facilitating deployment to a web server.

If you are not using web projects in Visual Studio—if you are using the **Open Web Site** command in Visual Studio, for example, or are using other development tools— you can alternatively specify the reference in the `web.config` file:

```
<configuration>
  <system.web>
    <compilation>
      <assemblies>
        <add assembly="NMath, Version=<Version>, Culture=neutral,
          PublicKeyToken=<Token>"/>
      </assemblies>
```

```
    </compilation>
  </system.web>
</configuration>
```

Native DLLs

For ASP.NET applications, Microsoft recommends that the `/bin` directory contain only .NET assemblies, not native DLLs.

If the deployment web server does not have **NMath** installed directly, then we recommend that the native DLLs be placed in a folder within the web application root directory, such as `/NativeBin`. This folder should then be copied to the deployment web server along with the rest of your application.

NMath Configuration

NMath settings can be configured as described in Section 1.5. However, when deploying web applications, especially to a shared hosting environment, it's quite common not to know the details of the physical structure of the file system, and to have restricted access to the system's environment variables. The references to resources within web applications are typically relative to the root of the virtual directory for the website, regardless of where they might physically reside on disk.

For this reason, the ASP.NET `~` operator can be used to specify the location of the **NMath** native libraries and the log file, relative to the web application root. That is, these can be specified in the `web.config` file like so:

```
<add key="NMathNativeLocation" value="~/NativeBin" />
<add key="NMathLogLocation" value="~/Logs" />
```

It is not sufficient to use relative paths, such as `bin/`, since the executing assembly is usually the ASP.NET worker process. Depending on the web server configuration, the working directory is usually a subdirectory of the `Windows` system directory (such as `c:\windows\system32`).

NOTE—The `~` operator can only be used in ASP.NET applications; specifying this in a Windows application will cause the path to be resolved incorrectly.

1.8 Very Large Objects

By default, the .NET runtime limits the size of any one object to 2 GB. For example, a matrix is limited to a theoretical maximum of `402,653,184` doubles or

805,306,368 floats —such as a 20,066 x 20,066 square **DoubleMatrix** or a 28,377 x 28,377 square **FloatMatrix**. With the release of .NET 4.5, developers can now create objects that exceed this limit by enabling gcAllowVeryLargeObjects in the run-time schema (x64 only), which controls the behavior of the .NET garbage collection system.

```
<configuration>
  <runtime>
    <gcAllowVeryLargeObjects enabled="true" />
  </runtime>
</configuration>
```

Very large objects are subject to the following restrictions:

- The maximum number of elements in an array is UInt32.MaxValue.
- The maximum index in any single dimension is 2,147,483,591 (0x7FFFFFC7) for byte arrays and arrays of single-byte structures, and 2,146,435,071 (0X7FEFFFFF) for other types.
- The maximum size for strings and other non-array objects is unchanged.

For more information, see

https://docs.microsoft.com/en-us/dotnet/framework/configure-apps/file-schema/runtime/gcallowverylargeobjects-element

Underlying all **NMath** vectors and matrices is a contiguous 1D array called a *data block* (Chapter 4). Thus, the number of elements for vectors or matrices must be less than 2,146,435,071. Table 2 summarizes the maximum size of various **NMath** objects under .NET 4.5 on a x64 OS with gcAllowVeryLargeObjects enabled.

Table 2 – Maximum object sizes

Class	Maximum size (elements)	Memory size (GBytes)
FloatVector	2,146,435,071	7.996
DoubleVector	2,146,435,071	15.992
FloatMatrix	2,146,435,071	7.996
DoubleMatrix	2,146,435,071	15.992

The complex versions of these classes have the same maximum number of elements but occupy twice the memory.

To use gcAllowVeryLargeObjects, you must target .NET 4.5 or later versions.

Very Large Objects with ASP.NET

The `gcAllowVeryLargeObjects` flag can only be set per-process, and only when the CLR is initializing. It cannot be set in the application-level `Web.config` file, because the CLR is already initialized by the time that file is read.

The workaround is to specify a `CLRConfigFile` in the `aspnet.config` file in the .NET framework installation. This little-known file is used to specify startup flags for both ASP.NET and CLR for those settings that are needed very early in the worker process lifetime, when the configuration system is not yet present.

Using `CLRConfigFile` allows you to specify an intermediate configuration file that the CLR can use for initialization. Once the CLR is up, ASP.NET will read your `Web.config` and run your application as normal.

For more information, see

`https://weblogs.asp.net/owscott/setting-an-aspnet-config-file-per-application-pool`

1.9 Documentation

NMath includes the following documentation:

- The *NMath User's Guide* (this manual)

  This document contains an overview of the product, and instructions on how to use it. You are encouraged to read the entire *User's Guide*. The *NMath User's Guide* is installed in:

  installdir`/Docs/NMath.UsersGuide.pdf`

  An HTML version of the *NMath User's Guide* may be viewed online using your browser at:

  `http://www.centerspace.net/doc/NMath/user/`

- The *NMath Reference*

  Complete API reference documentation may be viewed online using your browser at:

  `http://www.centerspace.net/doc/NMathSuite/ref/`

- A readme file

  This document describes the results of the installation process, how to build and run code examples, and lists any late-breaking product issues. The readme file is installed in:

installdir/`readme.txt`

This Manual

This manual assumes that you are familiar with the basics of .NET programming and object-oriented technology.

Most code examples in this manual are shown in both C# and Visual Basic. All **NMath** routines are callable from any .NET language.

This manual uses the following typographic conventions:

Table 3 – Typographic conventions

Convention	Purpose	Example
`Courier`	Function names, code, directories, file names, examples, and operating system commands.	`DoubleMatrix.Transform()` the `Assemblies` directory
italic	Conventional uses, such as emphasis and new terms.	The *data-view model* distinguishes between data and different views of the data.
bold	Class names, product names, and commands from an interface.	**FloatComplexVector** **NMath** Click **OK**.

specified range:

1.10 Technical Support

Technical support is available according to the terms of your CenterSpace License Agreement. You can also purchase extended support contracts through the CenterSpace website:

`http://www.centerspace.net`

To obtain technical support, contact CenterSpace by email at:

`mailto:support@centerspace.net`

You can save time if you isolate the problem to a small test case before contacting Technical Support.

PART II - NMATH CORE

CHAPTER 2. NMATH CORE

The `CenterSpace.NMath.Core` namespace is the unique **NMath** namespace. It includes the following core functionality:

- 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 a matrix, as well as functions for solving linear systems, computing determinants, inverses, and condition numbers.
- Least Squares solutions.
- 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.
- Data filtering, including a moving average filter and a Savitzky-Golay smoothing filter.
- Special functions, such factorial, binomial, the gamma function and related functions, Bessel functions, elliptic integrals, and many more.

To avoid using fully qualified names, preface your code with an appropriate namespace statement. For example:

Code Example – C#

```
using CenterSpace.NMath.Core;
```

Code Example – VB

```
imports CenterSpace.NMath.Core
```

CHAPTER 3. COMPLEX NUMBER TYPES

In **NMath**, the **FloatComplex** and **DoubleComplex** structures represent complex numbers, consisting of real and imaginary parts of single- and double-precision floating point numbers. **NMath** defines these types as structures, rather than classes, for greater efficiency. Remember that structures are value types in .NET, and are always passed by value.

These types support equality operations, conversion from `float`, `double`, or a string representation, and basic arithmetic operations. They also provide static member functions 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 transcendental functions such as exponents, logarithms, powers, and square roots, are available in the **NMathFunctions** class.

3.1 Creating Complex Numbers

This section describes how to construct instances of **FloatComplex** and **DoubleComplex**.

Creating Complex Numbers from Numeric Values

You can construct complex number objects from a pair of numeric values representing the real and imaginary parts. If only a single value is passed, it is assumed to be the real part, and the imaginary part is set to `0.0`. For example:

Code Example – C# complex numbers

```
var c = new FloatComplex( 1.3, 4.5 );    // 1.3 + 4.5i
var c2 = new DoubleComplex( 6.5 );       // 6.5 + 0.0i
```

Code Example – VB complex numbers

```
Dim C As New FloatComplex(1.3, 4.5)    ' 1.3 + 4.5i
Dim C2 As New DoubleComplex(6.5)       ' 6.5 + 0.0i
```

The static `FromPolar()` function constructs a complex number with a given magnitude and phase angle:

Code Example – C# complex numbers

```
var c = DoubleComplex.FromPolar( 2 * Math.Sqrt(2), Math.PI/4 );
// c = 2.0 + 2.0i
```

Code Example – VB complex numbers

```
Dim C As DoubleComplex =
  DoubleComplex.FromPolar(2 * Math.Sqrt(2), Math.PI / 4)
' c = 2.0 + 2.0i
```

Creating Complex Numbers from Strings

You can also construct complex number types from a string representation of the form `(real,imag)`. The parentheses are optional, and whitespace is ignored. Again, if only one value is supplied, it is assumed to be the real part. For instance, these are valid strings:

```
4.2,-5.1
(4.2,-5.1)
4.2
```

These are *not* valid strings:

```
4.2 - 5.1i
4.2 - 5.1
```

Thus:

Code Example – C# complex numbers

```
string s = "(1.1, -3.23)";
var c = new DoubleComplex( s );
```

Code Example – VB complex numbers

```
Dim S As String = "(1.1, -3.23)"
Dim C As New DoubleComplex(S)
```

The static `Parse()` method performs the same function:

Code Example – C# complex numbers

```
string s = "(1.1, -3.23)";
DoubleComplex c = DoubleComplex.Parse( s );
```

Code Example – VB complex numbers

```
Dim S As String = "(1.1, -3.23)"
Dim C As DoubleComplex = DoubleComplex.Parse(s)
```

NOTE—Note that you cannot use parentheses to represent negative numbers, as is done in some financial formats, when parsing complex number strings.

Conversely, the overridden `ToString()` member function returns a string representation of complex number:

Code Example – C# complex numbers

```
var c = new FloatComplex( 7.61, -1.2 );
Console.WriteLine( c.ToString() );  // prints "(7.61,-1.2)"
```

Code Example – VB complex numbers

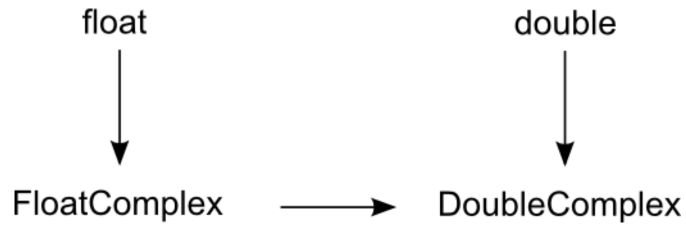
```
Dim C As New FloatComplex(7.61, -1.2)
Console.WriteLine(c.ToString())  ' prints "(7.61,-1.2)"
```

A variant of the `ToString()` method also accepts a standard .NET numeric format string. For example, the format string `"E"` indicates exponential (scientific) notion.

Implicit Conversion

The implicit conversion operators for the complex number classes are shown in Figure 1. An arrow indicates implicit promotion.

Figure 1 – Implicit conversion for complex numbers



3.2 Value Operations on Complex Numbers

Both **FloatComplex** and **DoubleComplex** have public instance variables `Real` and `Imag` that you can use to access and modify the real and imaginary parts of a complex number. For instance:

Code Example – C# complex numbers

```
var c1 = new DoubleComplex( 1.0 );
var c2 = new DoubleComplex( 2.13, 5.6 );
c1.Imag = c2.Imag;
c1.Real = -7.77;
```

Code Example – VB complex numbers

```
Dim C1 As New DoubleComplex(1.0)
Dim C2 As New DoubleComplex(2.13, 5.6)
C1.Imag = C2.Imag
C1.Real = -7.77
```

You can also use the static functions `Real()` and `Imag()` on class **NMathFunctions** to return the real and imaginary parts of a complex number:

Code Example – C# complex numbers

```
var c = new DoubleComplex( 2.13, 5.6 );
double d1 = c.Real();
double d2 = NMathFunctions.Real( c );   // d2 == d1
```

Code Example – VB complex numbers

```
Dim C As New DoubleComplex(2.13, 5.6)
Dim D1 = C.Real
Dim D2 = NMathFunctions.Real(C)   ' d2 == d1
```

3.3 Logical Operations on Complex Numbers

Operator `==` tests for equality of two complex numbers, and returns `true` if `left.Real==right.Real` and `left.Imag==right.Imag`; otherwise, `false`. Following the convention of the .NET Framework, if both objects are `null`, they test equal. Operator `!=` returns the logical negation of `==`.

The `Equals()` member function also tests for equality. `NaNEquals()` ignores values that are Not-a-Number (NaN).

NOTE—NMath provides no comparison operators for FloatComplex and DoubleComplex because there is no standard ordering for complex numbers.

3.4 Arithmetic Operations on Complex Numbers

NMath provides overloaded arithmetic operators for complex numbers with their conventional meanings for those .NET languages that support them, and equivalent named methods for those that do not. Table 4 lists the equivalent operators and methods.

Table 4 – Arithmetic operators

Operator	Equivalent Named Method
`+`	`Add()`
`-`	`Subtract()`
`*`	`Multiply()`
`/`	`Divide()`
Unary `-`	`Negate()`

All binary operators and equivalent named methods work either with two complex numbers, or with a complex number and a real value. For example, this C# code uses the overloaded operators:

Code Example – C# complex numbers

```
var c1 = new DoubleComplex( 3.2, 1.0 );
var c2 = new DoubleComplex( -11.002, -6.57 );
DoubleComplex c3 = c1 * c2;
c3 = (c1 / 3.5) - c2;
```

This Visual Basic code uses the equivalent named methods:

Code Example – VB complex numbers

```
Dim C1 As New DoubleComplex(3.2, 1.0)
Dim C2 As New DoubleComplex(-11.002, -6.57)
Dim C3 = DoubleComplex.Multiply(C1, C2)
C3 = DoubleComplex.Subtract(DoubleComplex.Divide(C1, 3.5), C2)
```

3.5 Functions of Complex Numbers

NMath provides a variety of functions that take complex numbers as arguments.

Conjugate, Norm, and Argument

NMath provides static functions on **FloatComplex** and **DoubleComplex** for common complex number functions:

- The static `Conj()` function returns the *conjugate* of a complex number. The conjugate of a complex number `a + bi` is defined as `a - bi`.
- The static `Norm()` method returns the *norm* (or *modulus*) of a complex number, defined as the square root of the sum of the squares of the real and imaginary parts.
- The static `Arg()` method returns the argument of a complex number, defined as the directed phase angle in polar coordinates.

For instance:

Code Example – C# complex numbers

```
var c = new FloatComplex( -8.2, 3.4 );
FloatComplex conj = FloatComplex.Conj( c );
float norm = FloatComplex.Norm( c );
float arg = FloatComplex.Arg( c );
```

Code Example – VB complex numbers

```
Dim C As New FloatComplex(-8.2, 3.4)
Dim Conj = FloatComplex.Conj(C)
Dim Norm = FloatComplex.Norm(C)
Dim Arg = FloatComplex.Arg(C)
```

Trigonometric Functions

NMath extends standard trigonometric functions `Sin()`, `Cos()`, `Sinh()`, `Cosh()`, `Tan()`, and `Tanh()` to take complex number arguments. Class **NMathFunctions** provides these functions as static methods; all take a single complex number as an argument and return a complex number as a result:

Code Example – C# complex numbers

```
var c = new DoubleComplex( 1.0, -3.9 );
DoubleComplex sin = NMathFunctions.Sin( c );
DoubleComplex cos = NMathFunctions.Cos( c );
```

Code Example – VB complex numbers

```
Dim C As New DoubleComplex(1.0, -3.9)
Dim Sin = NMathFunctions.Sin(C)
Dim Cos = NMathFunctions.Cos(C)
```

Transcendental Functions

NMath extends standard transcendental functions `Exp()` and `Log()` to take complex arguments. Class **NMathFunctions** provides these functions as static methods. For example:

Code Example – C# complex numbers

```
var c = new FloatComplex( -8.11, 3.04 );
FloatComplex exp = NMathFunctions.Exp( c );
FloatComplex log = NMathFunctions.Log( c );
```

Code Example – VB complex numbers

```
Dim C As New FloatComplex(-8.11, 3.04)
Dim Exp = NMathFunctions.Exp(C)
Dim Log = NMathFunctions.Log(C)
```

Class **NMathFunctions** also provides several static overloads of the exponential function `Pow()`. Versions exist to:

- raise a complex number to an integer exponent
- raise a complex number to a real exponent
- raise a complex number to a complex exponent
- raise a real value to a complex exponent

All return a complex number. For instance:

Code Example – C# complex numbers

```
var c1 = new DoubleComplex( 12.932, -4.0 );
DoubleComplex c2 = NMathFunctions.Pow( c1, 3 );
DoubleComplex c3 = NMathFunctions.Pow( c1, 1.12 );
DoubleComplex c4 = NMathFunctions.Pow( c1, c3 );
DoubleComplex c5 = NMathFunctions.Pow( 5.2, c1 );
```

Code Example – VB complex numbers

```
Dim C1 As New DoubleComplex(12.932, -4.0)
Dim C2 = NMathFunctions.Pow(C1, 3)
Dim C3 = NMathFunctions.Pow(C1, 1.12)
Dim C4 = NMathFunctions.Pow(C1, C3)
Dim C5 = NMathFunctions.Pow(5.2, C1)
```

Absolute Value and Square Root

The static `Abs()` function on class **NMathFunctions** returns the absolute value of a complex number, which is simply equal to the norm:

Code Example – C# complex numbers

```
var c = new DoubleComplex( 7.99, 0.3 );
double abs = NMathFunctions.Abs( c );
```

Code Example – VB complex numbers

```
Dim C As New DoubleComplex(7.99, 0.3)
Dim Abs = NMathFunctions.Abs(C)
```

NMath also extends the standard `Sqrt()` function to take a complex argument, again as a static method on class **NMathFunctions**. For example:

Code Example – C# complex numbers

```
var c = new FloatComplex( -8.11, 3.04 );
FloatComplex sqrt = NMathFunctions.Sqrt( c );
```

Code Example – VB complex numbers

```
Dim C As New FloatComplex(-8.11, 3.04)
Dim Sqrt = NMathFunctions.Sqrt(C)
```

CHAPTER 4. VIEWING DATA

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.

4.1 DataBlock Classes

This section describes the data block classes that underlie the **NMath** matrix and vector types.

NOTE—You will rarely need to deal directly with data block objects.

Class Names

The classes that encapsulate blocks of data in **NMath** are named **<Type>DataBlock**, where **<Type>** is **Float**, **Double**, **FloatComplex**, or **DoubleComplex**. (See Chapter 3 for a description of the complex number structures.) Thus:

- The **FloatDataBlock** class represents an array of single-precision floating point numbers.
- The **DoubleDataBlock** class represents an array of double-precision floating point numbers.
- The **FloatComplexDataBlock** class represents an array of single-precision complex numbers.
- The **DoubleComplexDataBlock** class represents an array of double-precision complex numbers.

The data referenced by the **NMath** vector and matrix classes is in the form of an instance of one of the data block classes.

Data Block Properties

Each data block object contains a reference to an array of the appropriate datatype, and an offset into the array. For instance, a **FloatComplexDataBlock** object contains a reference to an array of **FloatComplex** instances.

Think of a data block as encapsulating the concept of a pointer without using unsafe code. The value of an equivalent pointer is the address of the first element of the array, plus the offset.

Data block classes have the following public, *read-only* properties:

- The `Data` property returns the array referenced by the data block.
- The `Offset` property returns the current offset into the array.
- The `Length` property returns the number of elements currently referenced by the data block.

Accessing the Underlying Data

You rarely need to deal directly with data block objects. However, 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 data block object being viewed. As mentioned above, each data block class contains an array and an offset that allows you to extract a pointer to the beginning of the data. For example:

Code Example – C# data block

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

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

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

NOTE—Exercise caution when using raw data pointers.

Vector and matrix classes also provide `ToArray()` methods that return data copied into an array. Thus:

Code Example – C# data block

```
var v = new DoubleVector( "1 2 3 4 5" );
double[] d = v.ToArray();

var A = new DoubleMatrix( "3x3 [1 2 3  4 5 6  7 8 9]" );
double[,] d2 = A.ToArray();
```

Code Example – VB data block

```
Dim V As New DoubleVector("1 2 3 4 5")
Dim D() As Double = V.ToArray()

Dim A As New DoubleMatrix("3x3 [1 2 3  4 5 6  7 8 9]")
Dim D2(,) As Double = A.ToArray()
```

4.2 Slices and Ranges

The most common means of obtaining a different view of a specific block of data in **NMath** is by using **Slice** and **Range** indexing objects. These classes simply provide a way to specify a subset on non-negative integers with constant spacing, which you can then use as an indexing object into matrices and vectors. (See Chapter 5 and Chapter 6 for more information.)

Creating Slices and Ranges

The difference between a **Slice** and a **Range** is only in how you specify the integer subset. You construct a **Slice** object by specifying:

- a beginning index
- the total number of indices
- a step increment, or *stride*

For example, to create a slice for the indices `{ 2, 4, 6, 8, 10 }`, specify a start of `2`, `5` total elements, and a stride of `2`, like so:

Code Example – C# slice

```
var s = new Slice( 2, 5, 2 );
```

Code Example – VB slice

```
Dim S As New Slice(2, 5, 2)
```

You construct a **Range** object by specifying:

- a beginning index

- an ending index
- a stride

Thus, to create a range for the indices `{ 2, 4, 6, 8, 10 }`, specify a starting point of `2`, a stopping point of `10,` and a stride of `2`:

Code Example – C# range

```
var r = new Range( 2, 10, 2 );
```

Code Example – VB range

```
Dim R As New Range(2, 10, 2)
```

Creating Abstract Subsets

Suppose you want to address the elements in a vector `v` from the third element to the last. You *could* do this by creating a **Range** like so:

Code Example – C# range

```
var r = new Range( 2, v.Length - 1, 1 );
```

Code Example – VB range

```
Dim R As New Range(2, v.Length - 1, 1)
```

but this is rather cumbersome. As a convenience, therefore, **NMath** provides the **Position** enumeration which lists different view positions of underlying data. You can use values in the **Position** enumeration in conjunction with ranges and slices to create abstract subsets. The precise meaning of an abstract subset is only determined when an indexing object is applied to a particular matrix or vector. The enumerated values are:

- `Start` indicates the starting position.
- `MidPoint` indicates the midpoint position, rounded down for data structures with an even number of elements.
- `End` indicates the ending position.

For instance, this code creates two ranges that could be used to specify the odd and even elements of a vector:

Code Example – C# range

```
var evenElements = new Range( Position.Start, Position.End, 2 );
var oddElements = new Range( 1, Position.End, 2 );
```

Code Example – VB range

```
Dim EvenElements As New Range(Position.Start, Position.End, 2)
```

```
Dim OddElements As New Range(1, Position.End, 2)
```

The static `All` property on **Slice** and **Range** returns a new object indexing all elements:

Code Example – C# slice

```
Slice allElements = Slice.All;
```

Code Example – VB range

```
Dim AllElements As Slice = Slice.All
```

Modifying Ranges and Slices

You can modify an existing **Slice** or **Range** object using the `Set()` member function. For example:

Code Example – C# range

```
var r = new Range( Position.Start, Position.End, 2 );
r.Set( Position.Start, Position.MidPoint, 1 );
```

Code Example – VB range

```
Dim R As New Range(Position.Start, Position.End, 2)
R.Set(Position.Start, Position.MidPoint, 1)
```

CHAPTER 5.
VECTOR CLASSES

The **NMath** vector classes represent mathematical vectors of a particular datatype. Each class contains a reference to the data block they are viewing (see Chapter 4), along with the parameter values necessary to define their view:

- the number of elements
- a step increment, or *stride*, between elements of the data block

This is generally transparent to you. **NMath** provides indexers to perform the necessary indirection. For example, `v[i]` always returns the *i*th element of vector `v`'s view of the data.

NOTE—Indexing starts at 0.

5.1 Class Names

The classes that encapsulate vectors in **NMath** are named **<Type>Vector**, where **<Type>** is **Float**, **Double**, **FloatComplex**, or **DoubleComplex**. (See Chapter 3 for a description of the complex number classes.) Thus:

- The **FloatVector** class represents a vector of single-precision floating point numbers.
- The **DoubleVector** class represents a vector of double-precision floating point numbers.
- The **FloatComplexVector** class represents a vector of single-precision complex numbers.
- The **DoubleComplexVector** class represents a vector of double-precision complex numbers.

5.2 Creating Vectors

This section describes how to create instances of the vector classes.

Creating Vectors from Numeric Values

You can construct vector objects from numeric values in a variety of ways. All such constructors create a new view of a new data block.

A single passed, non-negative integer creates a vector of that length, with all values initialized to zero. For instance, this creates a vector of floating point values with `10` elements:

Code Example – C# vector

```
var v = new FloatVector( 10 );
```

Code Example – VB vector

```
Dim V As New FloatVector(10)
```

Another constructor enables you to set the initial value of all elements in the vector:

Code Example – C# vector

```
var v = new DoubleVector( 10, 2.0 );
// v[i]==2 for all i

var u =
  new FloatComplexVector( 10, new FloatComplex( 1.0, -2.0 ) );
// u[j] == 1 - 2i for all j
```

Code Example – VB vector

```
Dim V As New DoubleVector(10, 2.0)
' V(i)=2 for all i

Dim U As New FloatComplexVector(10, New FloatComplex(1.0, -2.0))
' U(j) = 1 - 2i for all j
```

Similarly, the vector classes provide a constructor that lets you set the length, the value of the first element, and an amount to increment each successive element in the vector. The *i*th element of the vector thus has the value `initialValue + i * increment`. For example, this creates the vector `[1, 3, 5, 7, 9]`:

Code Example – C# vector

```
var v = new FloatVector( 5, 1, 2 );
```

Code Example – VB vector

```
Dim V As New FloatVector(5, 1, 2)
```

You can also create a vector from an array of values:

Code Example – C# vector

```
double[] dblArray = {1.12, -2.0, 3.88, 1.2, 15.345};
var v = new DoubleVector( dblArray );
```

Code Example – VB vector

```
Dim DblArray() As Double = {1.12, -2.0, 3.88, 1.2, 15.345}
Dim V As New DoubleVector(DblArray)
```

Or a comma-delimited list:

Code Example – C# vector

```
var v = new FloatVector( 3.5, -6.7, 0.0, 3.11, 8.90, 5.0 );
```

Code Example – VB vector

```
Dim V As New FloatVector(3.5, -6.7, 0.0, 3.11, 8.9, 5.0)
```

Complex vector types can also be created from polar coordinates:

Code Example – C# complex vector from polar coordinates

```
var magnitudes = new FloatVector( 1, 2, 3, 6 );
var angles = new FloatVector( 1, 2, 3, -3 );
var v = FloatComplexVector.FromPolar( magnitudes, angles );
```

Code Example – VB complex vector from polar coordinates

```
Dim Magnitudes As New FloatVector( 1, 2, 3, 6 )
Dim Angles as New FloatVector( 1, 2, 3, -3 )
Dim V = FloatComplexVector.FromPolar( magnitudes, angles )
```

Lastly, you can use a random number generator to fill a vector with random values. See Chapter 9 for more information.

Creating Vectors from Strings

You can also construct vectors from a string representation of the form `[ v1 v2 v3 ... ]`. The brackets are optional, and extra whitespace is ignored. Again, these constructors create a new view of a new data block.

For instance:

Code Example – C# vector

```
string s = "4.3 -232   5.344 23.4   -32.43      ";
var v = new DoubleVector( s );

s = "[ (4.3,3.5) (23.4,-234.3) (-21.2,0) ]";
var u = new DoubleComplexVector( s );
```

Code Example – VB vector

```
Dim S As String = "4.3 -232   5.344 23.4   -32.43      "
Dim V As New DoubleVector(S)

S = "[ (4.3,3.5) (23.4,-234.3) (-21.2,0) ]"
Dim U As New DoubleComplexVector(S)
```

An optional second parameter to the constructor accepts values from the `System.Globalization.NumberStyles` enumeration. These styles are used by the `Parse()` methods of the numeric base types. For example:

Code Example – C# vector

```
using System.Globalization;

string s = "$4.52 $4.32 $4.56 $9.94 ($0.04) ($5.00)";
var v = new FloatVector( s,
     NumberStyles.AllowCurrencySymbol |
     NumberStyles.AllowDecimalPoint |
     NumberStyles.AllowParentheses );
```

Code Example – VB vector

```
Imports System.Globalization

Dim S As String = "$4.52 $4.32 $4.56 $9.94 ($0.04) ($5.00)"
Dim V As New FloatVector(s,
  NumberStyles.AllowCurrencySymbol Or
  NumberStyles.AllowDecimalPoint Or
  NumberStyles.AllowParentheses)
```

NOTE—Whitespace, even if set as a group separator, is interpreted as a data separator. Also note that currency representation is based on locale information in System.Globalization.CultureInfo, unless you override that information.

Finally, you can construct a vector from a given text reader. Just position the text reader at the start of a valid text representation of a vector. In this case, the brackets are required, since the text reader reads the stream until a closing bracket is encountered. For instance:

Code Example – C# vector

```
var reader = new StreamReader( "data.txt" );
// ... read until start of vector
var v = new DoubleVector( reader );
```

Code Example – VB vector

```
Dim Reader As New StreamReader("data.txt")
' ... read until start of vector
Dim V As New DoubleVector(Reader)
```

Again, an optional second parameter accept values from the `System.Globalization.NumberStyles` enumeration.

Instead of using a constructor, you can also create a vector from a string representation using the static `Parse()` method. The vector classes provide overloads of the `Parse()` method that accept a string, a string plus number styles, a text reader, and a text reader plus number styles.

Thus:

Code Example – C# vector

```
string s = "$4.52 $4.32 $4.56 $9.94 ($0.04) ($5.00)";
FloatVector v = FloatVector.Parse( s,
  NumberStyles.AllowCurrencySymbol |
  NumberStyles.AllowDecimalPoint |
  NumberStyles.AllowParentheses );
```

Code Example – VB vector

```
Dim S As String = "$4.52 $4.32 $4.56 $9.94 ($0.04) ($5.00)"
Dim V As FloatVector = FloatVector.Parse(s,
  NumberStyles.AllowCurrencySymbol Or
  NumberStyles.AllowDecimalPoint Or
  NumberStyles.AllowParentheses)
```

Conversely, the overridden `ToString()` member function returns a string representation of a vector of the form `[ v1 v2 v3 ... ]`. A variant of the `ToString()` method also accepts a standard .NET numeric format string. For example, the format string `"C"` indicates currency notion:

Code Example – C# vector

```
var v = new DoubleVector( "[ 1.12 8.95 3.95 4.60 ]" );
Console.WriteLine( v.ToString( "C" ) );
```

Code Example – VB vector

```
Dim V As New DoubleVector("[ 1.12 8.95 3.95 4.60 ]")
Console.WriteLine(V.ToString("C"))
```

The `Write()` member function writes a text representation of a vector to a given text writer. Again, a numeric format string is an optional second parameter.

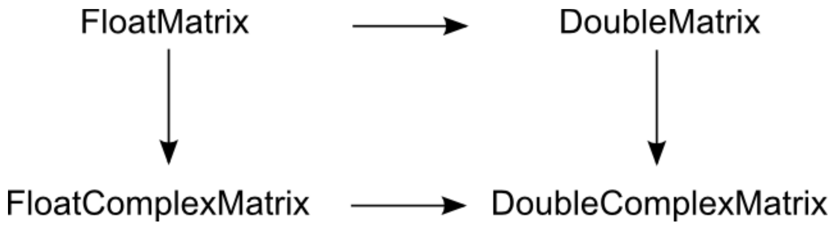
Creating Vectors from ADO.NET Objects

You can create a vector object from an ADO.NET object such as a **DataTable**, an array of **DataRow** objects, a **DataRowCollection**, or a **DataView**. See Chapter 52 for more information.

Implicit Conversion

The implicit conversion operators for the vector classes are shown in Figure 2. An arrow indicates implicit promotion.

Figure 2 – Implicit conversion for vectors



Copying Vectors

The vector classes provide three copy methods:

- `Clone()` returns a deep copy of a vector. Data is copied, so each vector references different data.
- `ShallowCopy()` returns a shallow copy of a vector. Data is not copied. Both vectors reference the same data.
- `DeepenThisCopy()` copies the data viewed by a vector to new data block. This guarantees that there is only one reference to the underlying data, and that this data is in contiguous storage.

For example:

Code Example – C# vector

```
var v = new DoubleVector( "[1 2 3 4 5]" );
DoubleVector u = v.ShallowCopy();

u[0] = 0;   // v[0] == u[0]
u.DeepenThisCopy();
u[1] = 0;   // v[1] != u[1]
```

Code Example – VB vector

```
Dim V As New DoubleVector("[1 2 3 4 5]")
Dim U As DoubleVector = V.ShallowCopy()

U(0) = 0   ' V(0) = U(0)
U.DeepenThisCopy()
U(1) = 0   ' V(1) <> U(1)
```

New Vector Views

A common method of creating vectors in **NMath** is to create a new vector view of data already referenced by another object. This is achieved using **Slice** and **Range** objects, as described in Section 4.2. Here's an example using a **Slice** object to create a new view of a vector's data:

Code Example – C# vector

```
var v = new DoubleVector( 10, 1, 1 );
// v = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]

var first3Elements = new Slice( 0, 3 );
DoubleVector u = v[first3Elements];
```

Code Example – VB vector

```
Dim V As New DoubleVector(10, 1, 1)
' v = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]

Dim First3Elements As New Slice(0, 3)
Dim U As DoubleVector = v(First3Elements)
```

Notice that the vector indexer is overloaded to accept indexing objects, and return a new view of the indexed data.

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

Code Example – C# vector

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

Code Example – VB vector

```
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.

Code Example – C# vector

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

Code Example – VB vector

```
U(2) = 99
V(2) = 99 ' true!
```

Here's another example using a **Range** object:

Code Example – C# vector

```
var v = new DoubleVector( "[1 2 3 4 5 6]" );
DoubleVector everyOther = v[new Range( 0,Position.End,2 )];
```

Code Example – VB vector

```
Dim V As New DoubleVector("[1 2 3 4 5 6]")
Dim EveryOther As DoubleVector = V(New Range(0, Position.End, 2))
```

Methods such as `Row()`, `Column()`, `Diagonal()`, and `Slice()` on the matrix classes also create vector views. See Chapter 6 for more information.

5.3 Value Operations on Vectors

The vector classes have the following read-only properties:

- `Length` gets the number of data elements in a vector.
- `Stride` gets the step between successive elements in the data block that a vector is viewing.
- `DataBlock` gets a reference to the data block that a vector is viewing.

For instance, if `v` is a **DoubleComplexVector** instance:

Code Example – C# vector

```
int length = v.Length;
int stride = v.Stride;
DoubleComplexDataBlock block = v.DataBlock;
```

Code Example – VB vector

```
Dim Length As Integer = V.Length
Dim Stride As Integer = V.Stride
Dim Block As DoubleComplexDataBlock = V.DataBlock
```

NOTE—As described in Section 4.1, use caution when accessing a data block referenced by a vector. Other objects may be viewing the same data.

Accessing and Modifying Vector Values

The vector classes provide standard indexing operators for getting and setting element values. Thus, `v[i]` always returns the *i*th element of vector `v`'s view of the data.

NOTE—Indexing starts at 0.

You can also use the `Set()` member function to set the data elements of a vector to a specified value.

For example, this code changes the contents of `v` to alternate values of `0` and `1`:

Code Example – C# vector

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

var evenElements = new Range( 0, Position.End, 2 );
var oddElements = new Range( 1, Position.End, 2 );

v.Set( evenElements, 0 );
v.Set( oddElements, 1 );
```

Code Example – VB vector

```
Dim V As New FloatVector(10, 0, 1)

Dim EvenElements As New Range(0, Position.End, 2)
Dim OddElements As New Range(1, Position.End, 2)

V.Set(EvenElements, 0)
V.Set(OddElements, 1)
```

NOTE—Any method that returns a vector view of the data referenced by a vector can be used to modify the values of the original vector, since the returned vector and the original vector share the data.

Clearing and Resizing a Vector

The vector classes provide two methods for changing the length of a vector after it has been created:

- `Clear()`resetsthe value of all data elements to zero.
- `Resize()` changes the size of a vector to the specified length, adding zeros or truncating as necessary.
- `ResizeAndClear()` performs the same function as `Resize()`, but also resets the value of all remaining data elements to zero.

Appending to a Vector

You can add new elements to the end of a vector using the `Append()` methods. Thus, this code adds a single element to the end of a vector:

Code Example – C# vector

```
var v = new FloatVector( 10, 0, 0.5F );
float x = 5.5F;
v.Append( x );
```

Code Example – VB vector

```
Dim V As New FloatVector(10, 0, 0.5F)
Dim X As Single = 5.5F
V.Append(X)
```

This code appends another vector to the end of a vector:

Code Example – C# vector

```
var v = new DoubleVector( 10, 0, 1 );
var w = new DoubleVector( 5, 11, 1 );
v.Append( w );
```

Code Example – VB vector

```
Dim V As New DoubleVector(10, 0, 1)
Dim W As New DoubleVector(5, 11, 1)
V.Append(W)
```

Note that a new vector is allocated by the `Append()` methods, and data is copied.

5.4 Logical Operations on Vectors

Operator `==` tests for equality of two vectors, and returns `true` if both vectors have the same dimensions and all values are equal; otherwise, `false`. Following the convention of the .NET Framework, if both objects are `null`, they test equal. The comparison of the values for **DoubleVector** and **DoubleComplexVector** is done using operator `==` for doubles; comparison of the values for **FloatVector**and **FloatComplexVector** is done using operator `==` for floats. Therefore, the values of the vectors must be exactly equal for this method to return `true`. Operator `!=` returns the logical negation of `==`.

The `Equals()` member function also tests for equality. `NaNEquals()` ignores values that are Not-a-Number (NaN).

5.5 Arithmetic Operations on Vectors

NMath provides overloaded arithmetic operators for vectors with their conventional meanings for those .NET languages that support them, and equivalent named methods for those that do not. Table 5 lists the equivalent operators and methods.

Table 5 – Arithmetic operators

Operator	Equivalent Named Method
`+`	`Add()`
`-`	`Subtract()`
`*`	`Multiply()`
`/`	`Divide()`
Unary `-`	`Negate()`
`++`	`Increment()`
`--`	`Decrement()`

Unary negation, increment, and decrement operators are applied to every element in a vector. The `Negate()` method returns a new vector object; `Increment()` and `Decrement()` do not.

All binary operators and equivalent named methods work either with two vectors, or with a vector and a scalar.

NOTE—Vectors must have the same length to be combined using the element-wise operators. Otherwise, a MismatchedSizeException is thrown. (See Chapter 53.)

For example, this C# code uses the overloaded operators:

Code Example – C# vector

```
var v = new FloatVector(5,0,1);  // [0,1,2,3,4]
var u = new FloatVector(5,1,1);  // [1,2,3,4,5]
float scalar = 2;

FloatVector w = v + scalar*u;
```

This Visual Basic code uses the equivalent named methods:

Code Example – VB vector

```
Dim V As New FloatVector(5, 0, 1)
Dim U As New FloatVector(5, 1, 1)
Dim Scalar As Single = 2

Dim W As FloatVector = FloatVector.Add(V,
  FloatVector.Multiply(Scalar, U))
```

NMath also provides overloads of the arithmetic named methods that accept three vector arguments. The third vector holds the result of applying the appropriate operation to the first two vectors. Because no new memory is allocated, efficiency is increased. This is especially useful for repeated operations, such as within loops. For instance, this code adds two vectors and stores the result in a third:

Code Example – C# vector

```
var v = new DoubleVector( "[ 0 1 2 3 4 ]" );
var u = new DoubleVector( 5, 1 );
var w = new DoubleVector( u.Length );

DoubleVector.Add( v, u, w );
DoubleVector.Add( v, u++, w );
DoubleVector.Add( v, v, w );
// Still only three vectors allocated
```

Code Example – VB vector

```
Dim V As New DoubleVector("[ 0 1 2 3 4 ]")
Dim U As New DoubleVector(5, 1)
Dim W As New DoubleVector(U.Length)

DoubleVector.Add(V, U, W)
DoubleVector.Add(V, U.Increment(), W)
DoubleVector.Add(V, V, W)
' Still only three vectors allocated
```

If the three vectors are not all of the same length, a **MismatchedSizeException** is thrown.

Note that the third vector argument can also be the same as one of the first two arguments, in which case it is overwritten with the result:

Code Example – C# vector

```
DoubleVector.Subtract( u, v, v );
```

Code Example – VB vector

```
DoubleVector.Subtract(U, V, V)
```

5.6 Functions of Vectors

NMath provides a variety of functions that take vectors as arguments.

Rounding Functions

Class **NMathFunctions** provides static methods for rounding a vector's elements:

- `Round()` rounds each element of a given vector to the specified number of decimal places.
- `Ceil()` applies the ceiling rounding function to each element of a given vector.
- `Floor()` applies the floor rounding function to each element of a given vector.

For instance, this code converts a vector of dollar amounts to Euros, then rounds to two decimal places:

Code Example – C# vector

```
var v = new DoubleVector( "$4.30 $0.08 ($5.87)",
   NumberStyles.Number | NumberStyles.AllowCurrencySymbol |
   NumberStyles.AllowParentheses );
v = v * 0.9289;  // exchange rate
v = NMathFunctions.Round( v, 2 );
```

Code Example – VB vector

```
Dim V As New DoubleVector("$4.30 $0.08 ($5.87)",
  NumberStyles.Number Or NumberStyles.AllowCurrencySymbol Or
  NumberStyles.AllowParentheses)
V = V * 0.9289  ' exchange rate
V = NMathFunctions.Round(V, 2)
```

Sums, Differences, and Products

Class **NMathFunctions** provides static methods to calculate sums, differences, and products of vector elements:

- `Sum()` returns the sum of the elements in a given vector.
- `AbsSum()` returns the sum of the absolute value of the elements in a given vector. (For complex vectors, this function calculates the sum of the L1 norms of the vector's elements.)
- `CumulativeSum()` returns a vector containing the cumulative sum of the elements in a given vector, such that `u[i] = v[0] + v[1] + ... v[i]`.
- `NaNSum()` returns the sum of the elements in a given vector, ignoring values that are Not-a-Number (NaN). NaN functions are available for real-value vectors only, not complex number vectors.
- `Delta()` returns a vector containing the differences between successive elements in a given vector, such that:

```
u[0] = v[0]
u[i] = v[i] - v[i-1]
```

- `Product()` returns the product of the elements in a given vector.
- `CumulativeProduct()` returns a vector containing the cumulative product of the elements in a given vector.
- `Dot()` returns the vector dot, or inner, product `d` of two vectors, `v` and `w`, where

```
d = v[0]*w[0] + v[1]*w[1]...
```

- `OuterProduct()` creates a matrix containing the outer product of two vectors.
- `Cross()` computes the cross product of two vectors. The vectors must have at least length three, and elements beyond three are ignored for purposes of computing the cross product.

For example:

Code Example – C# vector

```
var v = new FloatVector( "[1 2 3 4 5 6]" );
var u = new FloatVector( v.Length, 1, 1 );

float dp = NMathFunctions.Dot( v, u );
```

Code Example – VB vector

```
Dim V As New FloatVector("[1 2 3 4 5 6]")
Dim U As New FloatVector(V.Length, 1, 1)

Dim DP As Single = NMathFunctions.Dot(V, U)
```

Min/Max Functions

Class **NMathFunctions** provides static min/max finding methods that return the integer index of the element that meets the appropriate criterion:

- `MaxIndex()` returns the index of the element with the greatest value.
- `MinIndex()` returns the index of the element with the smallest value.
- `MaxAbsIndex()` returns the index of the element with the greatest absolute value.
- `MinAbsIndex()` returns the index of the element with the smallest absolute value.

Min/max value methods `MaxValue()`, `MinValue()`, `MaxAbsValue()`, and `MinAbsValue()` return the value of the element that meets the appropriate criterion. The returned type depends on the type of the vector. For instance, the `MaxValue()` method that accepts a **DoubleVector** returns a `double`.

`NaNMax()`, `NaNMin()`, `NaNMaxIndex()`, and `NaNMinIndex()` ignore values that are Not-a-Number (NaN). NaN functions are available for real-value vectors only, not complex number vectors.

Statistical Functions

The static `Mean()` method on **NMathFunctions** returns the mean of a given vector's elements. `Median()` returns the median. If the length of the vector is even, the middle two elements are averaged. `Median()` is available for real-value vectors only, not complex number vectors, because there is no standard ordering for complex numbers.

`Variance()` returns the biased variance of the elements. For instance:

Code Example – C# vector

```
var v = new DoubleVector( "[1 2 3 4 5 6]" );
double mean = NMathFunctions.Mean( v );
double variance = NMathFunctions.Variance( v );
```

Code Example – VB vector

```
Dim V As New DoubleVector("[1 2 3 4 5 6]")
Dim Mean As Double = NMathFunctions.Mean(V)
Dim Variance As Double = NMathFunctions.Variance(V)
```

`SumOfSquares()` returns the sum of the squared deviations from the mean of the elements of a given vector.

`NaNMean()`, `NanMedian()`, `NaNVariance()`, and `NanSumOfSquares()` ignore values that are Not-A-Number (NaN). `NaNCount()` returns the number of NaN values in a vector. NaN functions are available for real-value vectors only, not complex vectors.

Trigonometric Functions

NMath extends standard trigonometric functions `Acos()`, `Asin()`, `Atan()`, `Cos()`, `Cosh()`, `Sin()`, `Sinh()`, `Tan()`, and `Tanh()` to take vector arguments. Class **NMathFunctions** provides these functions as static methods. For example, this code construct a vector whose contents are the cosines of another vector:

Code Example – C# vector

```
var v = new FloatVector( 10, 0, 2 );
FloatVector cosv = NMathFunctions.Cos( v );
```

Code Example – VB vector

```
Dim V As New FloatVector(10, 0, 2)
Dim Cosv As FloatVector = NMathFunctions.Cos(V)
```

The static `Atan2()` method takes two vectors and applies the two-argument arc tangent function to successive pairs of elements.

Transcendental Functions

NMath extends standard transcendental functions `Exp()` and `Log()`, `Log10()` to take vector arguments. Class **NMathFunctions** provides these functions as static methods; each takes a single vector as an argument and return a vector as a result. For instance, this code creates a vector whose elements are the log of another vector’s elements:

Code Example – C# vector

```
var v = new DoubleVector( 10, 0, 5 );
DoubleVector log = NMathFunctions.Log( v );
```

Code Example – VB vector

```
Dim V As New DoubleVector(10, 0, 5)
Dim Log As DoubleVector = NMathFunctions.Log(V)
```

Class **NMathFunctions** also provides the exponential function `Pow()` to raise each element of a vector to a real exponent:

Code Example – C# vector

```
var v = new DoubleVector( 100, 0, 1 );
FloatVector vCubed = NMathFunctions.Pow( v, 3 );
```

Code Example – VB vector

```
Dim V As New DoubleVector(100, 0, 1)
Dim VCubed As FloatVector = NMathFunctions.Pow(V, 3)
```

Absolute Value and Square Root

The static `Abs()` function on class **NMathFunctions** applies the absolute value function to each element of a given vector:

Code Example – C# vector

```
var v = new DoubleVector ( 10, 0, -1 );
DoubleVector abs = NMathFunctions.Abs( v );
```

Code Example – VB vector

```
Dim V As New DoubleVector(10, 0, -1)
Dim ABS As DoubleVector = NMathFunctions.Abs(V)
```

NMath also extends the standard `Sqrt()` function to take a vector argument. Thus, this code creates a vector whose elements are the square root of another vector's elements:

Code Example – C# vector

```
var v = new DoubleVector( 10, 0, 5 );
DoubleVector sqrt = NMathFunctions.Sqrt( v );
```

Code Example – VB vector

```
Dim V As New DoubleVector(10, 0, 5)
Dim SQRT As DoubleVector = NMathFunctions.Sqrt(V)
```

Sorting Functions

The static `Sort()` method on class **NMathFunctions** sorts the elements of a given vector in ascending order using the quicksort algorithm and returns a new vector containing the result:

Code Example – C# vector

```
double[] dblArray = { 1.12, -2.0, 3.88, 1.2, 15.345 };
var v = new DoubleVector( dblArray );

v = NMathFunctions.Sort( v );
```

Code Example – VB vector

```
Dim DblArray() As Double = {1.12, -2.0, 3.88, 1.2, 15.345}
Dim V As New DoubleVector(DblArray)

V = NMathFunctions.Sort(V)
```

NOTE—This method is only available for FloatVector and DoubleVector, since there is no standard ordering for complex numbers.

Any NaN values in the vector are placed at the end of the ordered vector. To order the elements in descending order, `Reverse()` the returned vector:

Code Example – C# vector

```
v = NMathFunctions.Sort( v ).Reverse();
```

Code Example – VB vector

```
V = NMathFunctions.Sort(V).Reverse()
```

Complex Vector Functions

Static methods `Real()` and `Imag()` on class **NMathFunctions** return the real and imaginary part of a vector's elements. If the elements of the given vector are real, `Real()` simply returns the given vector and `Imag()` returns a vector of the same length containing all zeros.

Static methods `Arg()` and `Conj()` on class **NMathFunctions** return the arguments (or phases) and complex conjugates of a vector's elements. If the elements of the given vector are real, both methods simply return the given vector.

5.7 Generic Functions

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

Code Example – C# vector

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

// Construct a delegate for MyFunc
Func<double, double> 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!
```

Code Example – VB vector

```
Dim V As New DoubleVector(10, 0, -1)

' Construct a delegate for MyFunc
Dim MyFuncDelegate As New Func(Of Double, Double)(AddressOf MyFunc)

' Construct a new vector whose values are the result of applying
' MyFunc to the values in vector v. v remains unchanged.
Dim W As DoubleVector = 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.

5.8 Vector Enumeration

NMath vector classes provide standard .NET `GetEnumerator()` methods for returning **IEnumerator** objects. For example:

Code Example – C# vector

```
var v = new FloatVector( 12, -4.3F );
IEnumerator elements = v.GetEnumerator();
```

```
var data = new float[ v.Length ];
int i = 0;
while ( elements.MoveNext() )
{
  data[i++] = (float) elements.Current;
}
```

Code Example – VB vector

```
Dim V As New FloatVector(12, -4.3F)
Dim Elements As IEnumerator = V.GetEnumerator()

Dim Data(V.Length) As Single
Dim I = 0
While Elements.MoveNext()
  I = I + 1
  Data(I) = CType(Elements.Current, Single)
End While
```

Note that the `Current` property on an **IEnumerator** returns the current object in the collection, which must then be cast to the appropriate type. **NMath** also provides custom strongly-typed enumerators: **IFloatEnumerator**, **IDoubleEnumerator**, **IFloatComplexEnumerator**, and **IDoubleComplexEnumerator**. These avoid casting, and are therefore much faster. For instance:

Code Example – C# vector

```
var v = new FloatVector( 12, -4.3F );
IFloatEnumerator elements = v.GetFloatEnumerator();

var data = new float[ v.Length ];
int i = 0;
while ( elements.MoveNext() )
{
  data[i++] = elements.Current;      // No need to cast to float
}
```

Code Example – VB vector

```
Dim V As New FloatVector(12, -4.3F)
Dim Elements As IFloatEnumerator = V.GetFloatEnumerator()

Dim Data(V.Length) As Single
Dim I = 0
While Elements.MoveNext()
  I = I + 1
  Data(I) = elements.Current      ' No need to cast to float
End While
```

CHAPTER 6.
MATRIX CLASSES

The **NMath** matrix classes represent mathematical matrices of a particular datatype. Each class contains a reference to the data block they are viewing (see Chapter 4), along with the parameter values necessary to define their view:

- the number of rows and columns
- the distance between successive row elements, called the *row stride*
- the distance between successive column elements, called the *column stride*

This is generally transparent to you. **NMath** provides indexers to perform the necessary indirection. For example, `A[i,j]` always returns the element in the *i*th row and *j*th column of matrix `A`'s view of the data.

NOTE—Indexing starts at 0.

6.1 Class Names

The classes that encapsulate matrices in **NMath** are named **<Type>Matrix**, where **<Type>** is **Float**, **Double**, **FloatComplex**, or **DoubleComplex**. (See Chapter 3 for a description of the complex number classes.) Thus:

- The **FloatMatrix** class represents a matrix of single-precision floating point numbers.
- The **DoubleMatrix** class represents a matrix of double-precision floating point numbers.
- The **FloatComplexMatrix** class represents a matrix of single-precision complex numbers.
- The **DoubleComplexMatrix** class represents a matrix of double-precision complex numbers.

6.2 Creating Matrices

This section describes how to create instances of the matrix classes.

Creating Matrices from Numeric Values

You can construct matrix objects from numeric values in a variety of ways. All such constructors create a new view of a new data block.

The simplest constructor creates a matrix of the specified dimensions, with all values initialized to zero. For example, this code creates a 4x5 matrix of floating point values:

Code Example – C# matrix

```
var v = new FloatMatrix( 4, 5 );
```

Code Example – VB matrix

```
Dim V As New FloatMatrix(4, 5)
```

Another constructor enables you to set the initial value of all elements in the matrix. This creates a 3x3 matrix of **FloatComplex** instances with all values initialized to `1.0 - 3.0i`:

Code Example – C# matrix

```
var c = new FloatComplex( 1.0, -3.0 );
var A = new FloatComplexMatrix( 3, 3, c );
```

Code Example – VB matrix

```
Dim C As New FloatComplex(1.0F, -3.0F)
Dim A As New FloatComplexMatrix(3, 3, c)
```

Similarly, the matrix classes provide a constructor that lets you specify the dimensions of the matrix, the value of the first element, and an amount to increment each successive element. That is:

Code Example – C# matrix

```
A[i,j] = initialValue + (i+j) * increment
```

Code Example – VB matrix

```
A(I, J) = InitialValue + (I + J) * Increment
```

For instance:

Code Example – C# matrix

```
var A = new DoubleMatrix( 5, 5, 0, 1 );

//      | 0 5 10 15 20 |
//      | 1 6 11 16 21 |
// A =  | 2 7 12 17 22 |
//      | 3 8 13 18 23 |
//      | 4 9 14 19 24 |
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix(5, 5, 0, 1)

'       | 0 5 10 15 20 |
'       | 1 6 11 16 21 |
' A =  | 2 7 12 17 22 |
'       | 3 8 13 18 23 |
'       | 4 9 14 19 24 |
```

You can easily create a matrix from a 2-dimensional array of values. For example:

Code Example – C# matrix

```
float[,] data = new float[10,17];
for ( i = 0; i < 10; ++i )
{
  for ( j = 0; j < 17; ++j )
  {
    data[i,j] = 3.1415*i + j;
  }
}
var A = new FloatMatrix( data );
```

Code Example – VB matrix

```
Dim Data(10, 17) As Single
For I As Integer = 0 To 9
  For J As Integer = 0 To 16
    Data(I, J) = 3.1415 * I + J
  Next
Next
Dim A As New FloatMatrix(Data)
```

You can also create a matrix from a 1-dimensional array of values, but in this case you must also specify the dimensions of the matrix, and whether the given array is laid out in row-major or column-major order. **NMath** provides the **StorageType** enumeration for indicating the storage scheme. For instance:

Code Example – C# matrix

```
double[] data = { 0.0, 2.0,  4.0, 1.0,  3.0, 5.0 };
DoubleMatrix A =
   new DoubleMatrix( 3, 2, data, StorageType.ColumnMajor );

//       |  0.0  1.0  |
// A =  |  2.0  3.0  |
//       |  4.0  5.0  |
```

Code Example – VB matrix

```
Dim Data() As Double = {0.0, 2.0, 4.0, 1.0, 3.0, 5.0}
Dim A As New DoubleMatrix(3, 2, Data, StorageType.ColumnMajor)
```

```
'       |   0.0   1.0   |
' A =  |   2.0   3.0   |
'       |   4.0   5.0   |
```

NOTE—Once in a matrix, all data is stored in the underlying data block in column-major order.

You can also tile a matrix by replicating an existing matrix or vector using the **NMathFunctions** `RepMat()` methods. For example, this code creates a large matrix `B` consisting of an *m*-by-*n* tiling of copies of `A`:

Code Example – C# matrix

```
var A = new DoubleMatrix( 15, 3, -0.4, 0.3 );
int m = 4;
int n = 8;
DoubleMatrix B = NMathFunctions.RepMat( A, m, n );
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix(15, 3, -0.4, 0.3)
Dim M As Integer = 4
Dim N As Integer = 8
Dim B As DoubleMatrix = NMathFunctions.RepMat(A, M, N)
```

Lastly, you can use a random number generator to fill a matrix with random values. See Chapter 9 for more information.

Creating Matrices from Strings

You can also construct matrices from a string representation. The string must contain the number of rows, followed by an optional separator character such as `x`, followed by the number of columns. The matrix values, separated by white space, are then read in row by row. If the sequence of numbers begins with a left bracket `'['`, then the numbers are read until a matching right bracket `']'` is encountered. If no brackets are used, numbers are read until the end of the string. For example:

Code Example – C# matrix

```
var A = new DoubleMatrix( "3x3 [1 2 3 4 5 6 7 8 9]" );
var B =
   new FloatComplexMatrix( "2 2 (1,0) (2,1.2) (3.3,0) (4,3.12)" );
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix("3x3 [1 2 3 4 5 6 7 8 9]")
Dim B As New FloatComplexMatrix(
  "2 2 (1,0) (2,1.2) (3.3,0) (4,3.12)")
```

An optional second parameter accepts values from the `System.Globalization.NumberStyles` enumeration. These styles are used by the `Parse()` methods of the numeric base types. For instance:

Code Example – C# matrix

```
using System.Globalization;

string s = " 2 x 2 [ 1.1e+001 2.2e+000 4.4e+002 8.8e+000 ]";
var A = new DoubleMatrix( s, NumberStyles.Number |
  NumberStyles.AllowExponent );
```

Code Example – VB matrix

```
Imports System.Globalization

Dim S As String = " 2 x 2 [ 1.1e+001 2.2e+000 4.4e+002 8.8e+000 ]"
Dim A As New DoubleMatrix(S, NumberStyles.Number Or
  NumberStyles.AllowExponent)
```

Finally, you can construct a matrix from a given text reader. Just position the text reader at the start of a valid text representation of a matrix. In this case, the brackets are required, since the text reader reads the stream until a closing bracket is encountered.

For example:

Code Example – C# matrix

```
var reader = new StreamReader( "data.txt" );
// Read until the start of the matrix
var A = new FloatMatrix( reader );
```

Code Example – VB matrix

```
Dim Reader As New StreamReader("data.txt")
' Read until the start of the matrix
Dim A As New FloatMatrix(Reader)
```

Again, an optional second parameter accept values from the `System.Globalization.NumberStyles` enumeration.

Instead of using a constructor, you can also create a matrix from a string representation using the static `Parse()` method. The matrix class provide overloads of the `Parse()` method that accept a string, a string plus number styles, a text reader, and a text reader plus number styles. Thus:

Code Example – C# matrix

```
string s = "2x2 [ [1 2 3 4 ]";
DoubleMatrix A = DoubleMatrix.Parse( s );
```

Code Example – VB matrix

```
Dim S As String = "2x2 [ [1 2 3 4 ]"
Dim A As DoubleMatrix = DoubleMatrix.Parse(S)
```

Conversely, the overridden `ToString()` member function returns a string representation of a matrix of the form:

```
[number of rows] x [number of columns] [ matrix values row by row]
```

A variant of the `ToString()` method also accepts a standard .NET numeric format string. For instance, the format string `"C"` indicates currency notion:

Code Example – C# matrix

```
var A = new FloatMatrix( "2x2 [4.523 4.323 4.555 -9.943]" );
Console.WriteLine(A.ToString("C"));
// prints out "2x2 [ $4.52 $4.32 $4.56 ($9.94) ]" in en-US locale
```

Code Example – VB matrix

```
Dim A As New FloatMatrix("2x2 [4.523 4.323 4.555 -9.943]")
Console.WriteLine(A.ToString("C"))
' prints out "2x2 [ $4.52 $4.32 $4.56 ($9.94) ]" in en-US locale
```

The `Write()` member function writes a text representation of a matrix to a given text writer. Again, a numeric format string is an optional second parameter.

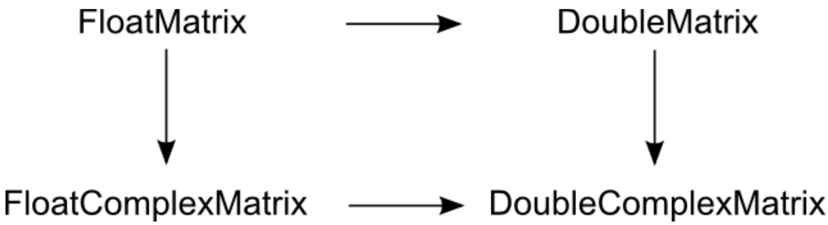
Creating Matrices from ADO.NET Objects

You can create a matrix object from an ADO.NET object such as a **DataTable**, an array of **DataRow** objects, a **DataRowCollection**, or a **DataView**. See Chapter 52 for more information.

Implicit Conversion

The implicit conversion operators for the matrix classes are shown in Figure 3. An arrow indicates implicit promotion.

Figure 3 – Implicit conversion for matrices



Copying Matrices

The matrix classes provide three copy methods:

- `Clone()` returns a deep copy of a matrix. Data is copied, so each matrix references different data.
- `ShallowCopy()` returns a shallow copy of a matrix. Data is not copied. Both matrices reference the same data.
- `DeepenThisCopy()` copies the data viewed by a matrix to new data block. This guarantees that there is only one reference to the underlying data, and that this data is in contiguous storage.

For instance:

Code Example – C# matrix

```
var A = new FloatMatrix( 4, 5, 1.0 );
FloatMatrix B = A.ShallowCopy();

B[0,0] = 0;    // A[0,0] == B[0,0]
B.DeepenThisCopy();
B[0,1] = 0;    // A[0,1] != B[0,1]
```

Code Example – VB matrix

```
Dim A As New FloatMatrix(4, 5, 1.0)
Dim B As FloatMatrix = A.ShallowCopy()

B(0, 0) = 0    ' A[0,0] == B[0,0]
B.DeepenThisCopy()
B(0, 1) = 0    ' A[0,1] != B[0,1]
```

Matrix Views

Another way to create matrices in **NMath** is to create a new matrix view of data already referenced by another matrix. This is achieved using **Slice** and **Range** objects, as described in Section 4.2. Here's an example using a **Range** object to create a new matrix view of the top left corner of a matrix:

Code Example – C# matrix

```
var A = new DoubleMatrix( 8, 8 );
var topLeft = new Range( 0, 3 );
DoubleMatrix AtopLeft = A[ topLeft, topLeft ];
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix(8, 8)
Dim TopLeft As New Range(0, 3)
Dim ATopLeft As DoubleMatrix = A(TopLeft, TopLeft)
```

Notice that the matrix indexer is overloaded to accept indexing objects, and return a new view of the indexed data.

6.3 Value Operations on Matrices

The matrix classes have the following read-only properties:

- `Cols` gets the number of columns in a matrix.
- `ColStride` gets the step increment between successive elements in a column.
- `Rows` gets the number of rows in a matrix.
- `RowStride` gets the step increment between successive elements in a column.
- `DataBlock` gets a reference to the data block that a matrix is viewing.

For example, if `A` is a **FloatComplexMatrix** instance:

Code Example – C# matrix

```
int cols = A.Cols;
int rows = A.Rows;
FloatComplexDataBlock block = A.DataBlock;
```

Code Example – VB matrix

```
Dim Cols As Integer = A.Cols
Dim Rows As Integer = A.Rows
Dim Block As FloatComplexDataBlock = A.DataBlock
```

NOTE—As described in Section 4.1, use caution when accessing a data block referenced by a matrix. Other objects may be viewing the same data.

Accessing and Modifying Matrix Values

The matrix classes provide standard indexers for getting and setting element value at a specified row and column position in a matrix. Thus, `A[i,j]` always returns the element in the *i*th row and *j*th column of matrix `A`'s view of the data.

NOTE—Indexing starts at 0.

Thus, this code sets the value in the lower right corner of the matrix to zero:

Code Example – C# matrix

```
var A = new DoubleMatrix( "2x2 [1 2 3 4]" );
A[1,1] = 0;
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix("2x2 [1 2 3 4]")
A(1, 1) = 0
```

The matrix indexer is also overloaded to accept **Range** and **Slice** indexing objects. For instance:

Code Example – C# matrix

```
var A = new DoubleMatrix( 5, 5, 2);
var B = new DoubleMatrix( 2, 2, 1);
var s = new Slice( 0, 2 );

A[s,s] = B
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix(5, 5, 2)
Dim B As New DoubleMatrix(2, 2, 1)
Dim S As New Slice(0, 2)

A(S, S) = B
```

You can also use the `Set()` member function to set the data elements of a matrix to a specified value. For instance, this code sets values in the last two columns of matrix `A` to zero:

Code Example – C# matrix

```
int rows = 5, cols = 5;
var A = new DoubleMatrix( rows, cols, 0, 1 );

var col = new Slice( 3, 2 );
Slice row = Slice.All;
A.Set( col, row, 0 );
```

Code Example – VB matrix

```
Dim Rows As Integer = 5
Dim Cols As Integer = 5
Dim A As New DoubleMatrix(Rows, Cols, 0, 1)

Dim Col As New Slice(3, 2)
Dim Row As Slice = Slice.All
A.Set(Col, Row, 0)
```

You can replace either slice with an integer value indicating a particular row or column. Thus, this code changes the values in the first column of `A` to `-1`:

Code Example – C# matrix

```
A.Set( 1, Slice.All, -1);
```

Code Example – VB matrix

```
A.Set(1, Slice.All, -1)
```

NOTE—Any method that returns a vector view of the data referenced by a matrix can be used to modify the values of matrix, since the returned vector and the matrix share the data. See Section 6.6.

Clearing and Resizing a Matrix

The matrix classes provide two methods for changing the size of a matrix after it has been created:

- `Clear()` resets the value of all data elements to zero.
- `Resize()` changes the size of a matrix to the specified number of rows and columns, adding zeros or truncating as necessary.
- `ResizeAndClear()` performs the same function as `Resize()`, but also resets the value of all remaining data elements to zero.

6.4 Logical Operations on Matrices

Operator `==` tests for equality of two matrices, and returns `true` if both matrices have the same dimensions and all values are equal; otherwise, `false`. Following the convention of the .NET Framework, if both objects are `null`, they test equal. The comparison of the values for **DoubleMatrix** and **DoubleComplexMatrix** is done using operator `==` for doubles; comparison of the values for **FloatMatrix** and **FloatComplexMatrix** is done using operator `==` for floats. Therefore, the values of the matrices must be exactly equal for this method to return `true`. Operator `!=` returns the logical negation of `==`.

The `Equals()` member function also tests for equality. `NaNEquals()` ignores values that are Not-a-Number (NaN).

6.5 Arithmetic Operations on Matrices

NMath provides overloaded arithmetic operators for matrices with their conventional meanings for those .NET languages that support them, and equivalent named methods for those that do not. Table 6 lists the equivalent operators and methods.

Table 6 – Arithmetic operators

Operator	Equivalent Named Method
`+`	`Add()`
`-`	`Subtract()`
`*`	`Multiply()`
`/`	`Divide()`
Unary `-`	`Negate()`
`++`	`Increment()`
`--`	`Decrement()`

Unary negation, increment, and decrement operators are applied to every element in a matrix. The `Negate()` method returns a new matrix object; `Increment()` and `Decrement()` do not.

All binary operators and equivalent named methods work either with two matrices, or with a matrix and a scalar.

NOTE—Matrices must have the same dimensions to be combined using the element-wise operators. Otherwise, a MismatchedSizeException is thrown. (See Chapter 53.)

For example, this C# code uses the overloaded operators:

Code Example – C# matrix

```
int rows = 3, cols = 3;

var A =
   new DoubleComplexMatrix( rows, cols, new DoubleComplex(1,0) );
var B =
   new DoubleComplexMatrix( rows, cols, new DoubleComplex(0,1) );
var s = new DoubleComplex( 2, 0 );

DoubleComplexMatrix result = A + s*B;
```

This Visual Basic code uses the equivalent named methods:

Code Example – VB matrix

```
Dim rows As Integer = 3
Dim cols As Integer = 3

Dim A As _
   New DoubleComplexMatrix(rows, cols, New DoubleComplex(1, 0))
Dim B As _
   New DoubleComplexMatrix(rows, cols, New DoubleComplex(0, 1))
Dim s As New DoubleComplex(2, 0)

Dim result As DoubleComplexMatrix = _
   DoubleComplexMatrix.Add(A, DoubleComplexMatrix.Multiply(s, B))
```

NMath also provides overloads of the arithmetic named methods that accept three matrix arguments. The third matrix holds the result of applying the appropriate operation to the first two matrices. Because no new memory is allocated, efficiency is increased. This is especially useful for repeated operations, such as within loops. For instance, this code multiplies two matrices and stores the result in a third:

Code Example – C# matrix

```
int rows = size;
int cols = size;
var A = new DoubleMatrix( rows, cols, 0, 1);
var B = new DoubleMatrix( rows, cols, 1, 1 );
var C = new DoubleMatrix( rows, cols );

FloatMatrix.Multiply( A, B, C );
FloatMatrix.Multiply( A--, B, C );
FloatMatrix.Multiply( B, B, C );
// Still only three matrices allocated
```

Code Example – VB matrix

```
Dim Rows As Integer = Size
Dim Cols As Integer = Size
Dim A As New DoubleMatrix(Rows, Cols, 0, 1)
Dim B As New DoubleMatrix(Rows, Cols, 1, 1)
Dim C As New DoubleMatrix(Rows, Cols)

FloatMatrix.Multiply(A, B, C)
FloatMatrix.Multiply(A.Decrement(), B, C)
FloatMatrix.Multiply(B, B, C)
' Still only three matrices allocated
```

If the three matrices do not have the same dimensions, a **MismatchedSizeException** is thrown.

6.6 Vector Views

A variety of methods are providing for returning vector views of the data referenced by a matrix. The returned vector and the matrix share the data, so care must be exercised when modifying values. If after constructing a different view of an object's data you want your own private view that you can modify without affecting any other objects, simply invoke the `DeepenThisCopy()` method on the vector:

Code Example – C# matrix

```
var A = new DoubleMatrix( 8, 8, 1, 1 );
DoubleVector v = A.Diagonal();

v.DeepenThisCopy();
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix(8, 8, 1.0, 1.0)
Dim V As DoubleVector = A.Diagonal()

V.DeepenThisCopy()
```

Row and Column Views

Member functions `Row()` and `Column()` return vector views of a specified row or column. For instance:

Code Example – C# matrix

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

DoubleVector row1 = A.Row( 1 );
DoubleVector col0 = A.Col( 0 );
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix("3x3 [1 2 3  4 5 6  7 8 9]")

Dim Row1 As DoubleVector = A.Row(1)
Dim Col0 As DoubleVector = A.Col(0)
```

Diagonal Views

The `Diagonal()` member function returns a vector view of a diagonal of a matrix. If no diagonal is specified, a vector view of the main diagonal is returned. For example, this code increments every element along the main diagonal:

Code Example – C# matrix

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

Code Example – VB matrix

```
Dim A As New FloatMatrix(5, 8)
A.Diagonal().Increment()
```

Arbitrary Slices

The `Slice()` member function returns a vector view of an arbitrary slice of a matrix. The parameters are:

- the starting row
- the starting column
- the number of elements
- the row stride
- the column stride

The slice begins at the starting row and column, and extends for the number of elements. The increment between successive elements in the vector is row stride rows and column stride columns. For example, this code returns a view of the diagonal from the bottom left corner to the top right of a 3x3 matrix:

Code Example – C# matrix

```
var A = new DoubleMatrix( "3x3 [1 2 3  4 5 6  7 8 9]" );
DoubleVector v = A.Slice( 2, 0, 3, -1, 1 );
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix("3x3 [1 2 3  4 5 6  7 8 9]")
Dim V As DoubleVector = A.Slice(2, 0, 3, -1, 1)
```

6.7 Functions of Matrices

NMath provides a variety of functions that take matrices as arguments.

Matrix Transposition

The matrix classes provide `Transpose()` member functions for calculating the transpose of a matrix: `B[i,k] = A[k,i]`. Class **NMathFunctions** also provides a static `Transpose()` method that returns the transpose of a matrix. For instance:

Code Example – C# matrix

```
var A = new FloatComplexMatrix( 5, 5, 1, 1 );
FloatComplexMatrix B = A.Transpose();
FloatComplexMatrix C = NMathFunctions.Transpose(A);
// B == C
```

Code Example – VB matrix

```
Dim A As New FloatComplexMatrix(5, 5, 1.0F, 1.0F)
Dim B As FloatComplexMatrix = A.Transpose()
Dim C As FloatComplexMatrix = NMathFunctions.Transpose(A)
' B == C
```

In both cases, the matrix returned is a new view of the same data. `Transpose()` just swaps the number of rows and the number of columns, as well as the row strides and column strides. No data is copied.

Matrix Norms

The matrix classes provide member functions `OneNorm()` to compute the 1-norm (or largest column sum) of a matrix, `InfinityNorm()` to compute the infinitynorm (or largest row sum) of a matrix, and `FrobeniusNorm()` to compute the Frobenius norm. For instance:

Code Example – C# matrix

```
var A = new DoubleMatrix( "3x3 [1 2 3  4 5 6  7 8 9]" );
double d1 = A.OneNorm();
double d2 = A.InfinityNorm();
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix("3x3 [1 2 3  4 5 6  7 8 9]")
Dim D1 As Double = A.OneNorm()
Dim D2 As Double = A.InfinityNorm()
```

Matrix Products

Class **NMathFunctions** provides the static `Product()` method for calculating the matrix product of two matrices. For example:

Code Example – C# matrix

```
var A = new FloatMatrix( "3x3 [1 2 3  4 5 6  7 8 9]" );
var B = new FloatMatrix( 3, 3, 1, 1 );
FloatMatrix C = NMathFunctions.Product( A, B );
```

Code Example – VB matrix

```
Dim A As New FloatMatrix("3x3 [1 2 3  4 5 6  7 8 9]")
Dim B As New FloatMatrix(3, 3, 1.0F, 1.0F)
Dim C As FloatMatrix = NMathFunctions.Product(A, B)
```

Transpose operations to be performed on the operands of a matrix-matrix multiply operation are specified using a value from the `NMathFunctions.ProductTransposeOption` enum:

- `TransposeNone` does not transpose either matrix before multiplying.
- `TransposeBoth` transposes both operands before multiplying.
- `TransposeFirst` transposes only the first operand before multiplying.
- `TransposeSecond` transposes only the second operand before multiplying.
- `ConjTransposeBoth` takes the conjucate transpose of both operands before multiplying.
- `ConjTransposeFirst` takes the conjugate transpose only of the first operand before multiplying.
- `ConjTransposeSecond` takes the conjugate transpose only of the second operand before multiplying.

Thus, this code calculates the inner product of the transpose of `A` with `B`:

Code Example – C# matrix

```
var A = new FloatMatrix( "3x3 [1 2 3  4 5 6  7 8 9]" );
var B = new FloatMatrix( 3, 3, 1, 1 );
FloatMatrix C = NMathFunctions.Product( A, B,
  ProductTransposeOption.TransposeFirst );
```

Code Example – VB matrix

```
Dim A As New FloatMatrix("3x3 [1 2 3  4 5 6  7 8 9]")
Dim B As New FloatMatrix(3, 3, 1.0F, 1.0F)
Dim C As FloatMatrix = NMathFunctions.Product(A, B,
  ProductTransposeOption.TransposeFirst)
```

Additional overloads of the `Product()` method calculate the inner product of a matrix and a scalar:

Code Example – C# matrix

```
var A = new DoubleMatrix( "3x3 [1 2 3  4 5 6  7 8 9]" );
var v = new DoubleVector( "[3 2 1]" );
DoubleVector u = NMathFunctions.Product( A, v );
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix("3x3 [1 2 3  4 5 6  7 8 9]")
Dim V As New DoubleVector("[3 2 1]")
Dim U As DoubleVector = NMathFunctions.Product(A, V)
```

Overloads are also provided which place the result of multiplying the first two operands into a third argument, rather than allocating new memory for the result:

Code Example – C# matrix

```
NMathFunctions.Product( A, B, C,
  ProductTransposeOption.TransposeBoth );
```

Code Example – VB matrix

```
NMathFunctions.Product(A, B, C,
  ProductTransposeOption.TransposeBoth)
```

Matrix Inverse and Pseudoinverse

Class **NMathFunctions** provides the static `Inverse()` method for calculating the inverse of a matrix:

Code Example – C# matrix

```
var A = new FloatMatrix( "3x3 [1 2 3  4 5 6  7 8 9]" );
FloatMatrix AInv = NMathFunctions.Inverse( A );
```

Code Example – VB matrix

```
Dim A As New FloatMatrix("3x3 [1 2 3  4 5 6  7 8 9]")
Dim AInv As FloatMatrix = NMathFunctions.Inverse(A)
```

The standard inverse fails if the matrix is singular or not square.

The *pseudoinverse* A^+ is a generalization of the inverse, and exists for any n x m matrix, where $n \geq m$:

$$A^+ = (A^T A)^{-1} A^T$$

NMathFunctions provides the static `Pseudoinverse()` method:

Code Example – C# matrix

```
FloatMatrix APseudoInv = NMathFunctions.Pseudoinverse( A );
```

Code Example – VB matrix

```
Dim APseudoInv As FloatMatrix = NMathFunctions.PseudoInverse(A)
```

To test the quality of the pseudoinverse, you can check the condition number of $A^T A$:

Code Example – C# matrix

```
float cond = NMathFunctions.ConditionNumber(
  NMathFunctions.TransposeProduct( A, A ), NormType.OneNorm );
if (cond > 0.000001)
{
  // good
}
```

Code Example – VB matrix

```
Dim Cond As Single = NMathFunctions.ConditionNumber(
  NMathFunctions.TransposeProduct(A, A), NormType.OneNorm)
If Cond > 0.000001 Then
  ' good
End If
```

NOTE—The best way to compute the pseudoinverse is to use singular value decomposition. Method MatrixFunctions.Pseudoinverse() implements this method.

Rounding Functions

Class **NMathFunctions** provides static methods for rounding the elements of a matrix:

- `Round()` rounds each element of a given matrix to the specified number of decimal places.

- `Ceil()` applies the ceiling rounding function to each element of a given matrix.
- `Floor()` applies the floor rounding function to each element of a given matrix.

Sums and Differences

The static `Sum()` method on **NMathFunctions** accepts a matrix and returns a vector containing the sums of the elements in each column. To sum the rows, simply `Transpose()` the matrix first.

For example:

Code Example – C# matrix

```
var A = new DoubleMatrix( 5, 8, 1, 1 );

DoubleVector AColSums = NMathFunctions.Sum( A );
DoubleVector ARowSums = NMathFunctions.Sum( A.Transpose() );
A.Transpose()  // return A to original view
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix(5, 8, 1.0, 1.0)

Dim AColSums As DoubleVector = NMathFunctions.Sum(A)
Dim ARowSums As DoubleVector = NMathFunctions.Sum(A.Transpose())
A.Transpose()  ' return A to original view
```

`Transpose()` just swaps the number of rows and the number of columns, as well as the row strides and column strides. No data is copied.

`NaNSum()` ignores values that are Not-A-Number (NaN).

NOTE—NaN functions are available for real-value matrices only, not complex number matrices.

The static `Delta()` method on **NMathFunctions** returns a new matrix with the same dimensions as a given matrix, whose values are the result of applying the vector delta function to each column of the matrix. The vector delta computes the differences between successive elements in a given vector, such that:

```
u[0] = v[0]
u[i] = v[i] - v[i-1]
```

Applied to a matrix, `Delta()` returns a new matrix such that:

```
B[0,j] = A[0,j]
B[i,j] = A[i,j] - A[i-1,j]
```

Again, to apply the `Delta()` function to rows rather than columns, just transpose the matrix first.

Min/Max Functions

Class **NMathFunctions** provides static min/max finding methods that return a vector containing the value of the element in each column that meets the appropriate criterion:

- `Max()` returns a vector containing the greatest values in each column.
- `Min()` returns a vector containing the smallest values in each column.
- `NaNMax()` returns a vector containing the greatest values in each column, ignoring values that are Not-a-Number (NaN).
- `NaNMin()` returns a vector containing the smallest values in each column.

NOTE—NaN functions are available for real-value matrices only, not complex number matrices.

To apply these functions to the rows of a matrix, simply `Transpose()` the matrix first.

Statistical Functions

The static `Mean()`, `Median()`, `Variance()`, and `SumOfSquares()` methods on **NMathFunctions** are overloaded to accept a matrix and return a vector containing the result of applying the appropriate function to each column in the matrix:

Code Example – C# matrix

```
var A = new FloatMatrix( 5, 5, 0, 2 );
FloatVector means = NMathFunctions.Mean( A );
FloatVector medians = NMathFunctions.Median( A );
FloatVector variances = NMathFunctions.Variance( A );
```

Code Example – VB matrix

```
Dim A As New FloatMatrix(5, 5, 0.0F, 2.0F)
Dim Means As FloatVector = NMathFunctions.Mean(A)
Dim Medians As FloatVector = NMathFunctions.Median(A)
Dim Variances As FloatVector = NMathFunctions.Variance(A)
```

`NaNMean()`, `NaNMedian()`, `NaNVariance()`, and `NaNSumOfSquares()` ignore values that are Not-A-Number (NaN). `NaNCount()` returns the number of NaN values in each column. NaN functions are available for real-value matrices only, not complex matrices.

To apply these functions to the rows of a matrix, simply `Transpose()` the matrix first.

Trigonometric Functions

NMath extends standard trigonometric functions `Acos()`, `Asin()`, `Atan()`, `Cos()`, `Cosh()`, `Sin()`, `Sinh()`, `Tan()`, and `Tanh()` to take matrix arguments. Class **NMathFunctions** provides these functions as static methods. For instance, this code construct a matrix whose contents are the sines of another matrix:

Code Example – C# matrix

```
var A = new FloatMatrix( 10, 10, 0, Math.Pi/4 );
FloatMatrix cosA = NMathFunctions.Cos( A );
```

Code Example – VB matrix

```
Dim A As New FloatMatrix(10, 10, 0.0F, Math.PI / 4.0F)
Dim CosA As FloatMatrix = NMathFunctions.Cos(A)
```

The static `Atan2()` method takes two matrices and applies the two-argument arc tangent function to corresponding pairs of elements.

Transcendental Functions

NMath extends standard transcendental functions `Exp()`, `Log()`, `Log10()`, and `Sqrt()` to take matrix arguments. Class **NMathFunctions** provides these functions as static methods; each takes a single matrix as an argument and return a matrix as a result. For example, this code creates a matrix whose elements are the square root of the elements in another matrix:

Code Example – C# matrix

```
var A = new DoubleMatrix( 3, 3, 1, 1 );
DoubleMatrix sqrt = NMathFunctions.Sqrt( A );
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix(3, 3, 1.0, 1.0)
Dim Sqrt As DoubleMatrix = NMathFunctions.Sqrt(A)
```

Function `Expm()` on **NMathFunctions** raises the constant *e* to a given matrix power, using a scaling and squaring method based upon Pade approximation. This is different than method `Exp()` which exponentiates each element of a matrix independently.

Class **NMathFunctions** also provides the exponential function `Pow()` to raise each element of a matrix to a real exponent.

Code Example – C# matrix

```
var A = new DoubleMatrix( "2x2 [1 2 3 4]" );
DoubleMatrix cubed = NMathFunctions.Pow( A, 3 );
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix("2x2 [1 2 3 4]")
Dim Cubed As DoubleMatrix = NMathFunctions.Pow(A, 3)
```

Absolute Value and Square Root

The static `Abs()` function on class **NMathFunctions** applies the absolute value function to each element of a given matrix:

Code Example – C# matrix

```
var A = new DoubleMatrix( 10, 10, 0, -1 );
DoubleMatrix abs = NMathFunctions.Abs( A );
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix(10, 10, 0.0, -1.0)
Dim Abs As DoubleMatrix = NMathFunctions.Abs(A)
```

NMath also extends the standard `Sqrt()` function to take a matrix argument. Thus, this code creates a matrix whose elements are the square root of another matrix's elements:

Code Example – C# matrix

```
var A = new FloatMatrix( 10, 10, 1, 2 );
FloatMatrix sqrt = NMathFunctions.Sqrt( A );
```

Code Example – VB matrix

```
Dim A As New FloatMatrix(10, 10, 1.0F, 2.0F)
Dim Sqrt As FloatMatrix = NMathFunctions.Sqrt(A)
```

Sorting Functions

The static `SortByColumn()` method on class **NMathFunctions** sorts the rows of a matrix by the values in a specified column. For instance, this code sorts matrix `A` by values in the first column:

Code Example – C# matrix

```
var A = new FloatMatrix( 20, 20, 0, 1 );
A = NMathFunctions.SortByColumn( A, 0 );
```

Code Example – VB matrix

```
Dim A As New FloatMatrix(20, 20, 0.0F, 1.0F)
A = NMathFunctions.SortByColumn(A, 0)
```

Complex Matrix Functions

Static methods `Real()` and `Imag()` on class **NMathFunctions** return the real and imaginary part of the elements of a matrix. If the elements of the given matrix are real, `Real()` simply returns the given matrix and `Imag()` returns a matrix of the same dimensions containing all zeros.

Static methods `Arg()` and `Conj()` on class **NMathFunctions** return the arguments (or phases) and complex conjugates of the elements of a matrix. If the elements of the given matrix are real, both methods simply return the given matrix.

For instance:

Code Example – C# matrix

```
DoubleComplexMatrix A =
 new DoubleComplexMatrix( "2x2 [(1,-1) (2,-.5) (2.2,1.1) (7,9)]" );

DoubleComplexMatrix AConj = NMathFunctions.Conj( A );
// AConj = 2x2 [(1,1) (2,0.5) (2.2,-1.1) (7,-9)]

// Now use the Imag method to create a real matrix containing
// the imaginary parts of AConj.
DoubleMatrix AConjImag = NMathFunctions.Imag( AConj );
```

Code Example – VB matrix

```
Dim A As New DoubleComplexMatrix(
  "2x2 [(1,-1) (2,-.5) (2.2,1.1) (7,9)]")

Dim AConj As DoubleComplexMatrix = NMathFunctions.Conj(A)
' AConj = 2x2 [(1,1) (2,0.5) (2.2,-1.1) (7,-9)]

' Now use the Imag method to create a real matrix containing
' the imaginary parts of AConj.
Dim AConjImag As DoubleMatrix = NMathFunctions.Imag(AConj)
```

6.8 Generic Functions

NMath provides generic functions that apply a given function delegate to every element in a matrix, or to every column in a matrix.

Applying Elementwise Functions

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

Code Example – C# matrix

```
var A = new DoubleMatrix( 5, 5, 0, Math.Pi/4 );
var MyFuncDelegate = new Func<double, double>( MyFunc );
DoubleMatrix B = A.Apply( MyFuncDelegate );
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix(5, 5, 0.0, Math.PI / 4.0)
Dim MyFuncDelegate As New Func(Of Double, Double)(MyFunc)
Dim B As DoubleMatrix = A.Apply(MyFuncDelegate)
```

Applying Columnwise Functions

NMath 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`:

Code Example – C# matrix

```
var A = new FloatMatrix( 5, 5, 0, Math.Pi/4 );
var MyFuncDelegate = new Func<FloatVector, float>( MyFunc );
FloatVector v = A.ApplyColumns( MyFuncDelegate );
```

Code Example – VB matrix

```
Dim A As New FloatMatrix(5, 5, 0.0F, Math.PI / 4.0F)
Dim MyFuncDelegate As New Func(Of FloatVector, Single)(MyFunc)
Dim V As FloatVector = A.ApplyColumns(MyFuncDelegate)
```

To apply a function to the rows of matrix, just `Transpose()` the matrix first. `Transpose()` simply swaps the number of rows and the number of columns, as well as the row strides and column strides. No data is copied, so it's a relatively cheap operation. For instance:

Code Example – C# matrix

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

Code Example – VB matrix

```
Dim V As FloatVector = A.Transpose().ApplyColumns(MyFuncDelegate)
A.Transpose()    ' return A to original view
```

6.9 Matrix Enumeration

NMath matrix classes provide standard .NET `GetEnumerator()` methods for returning **IEnumerator** objects. For example:

Code Example – C# matrix

```
int rows = 13, cols = 3;
var A = new DoubleMatrix( rows, cols, 0, .25 );
IEnumerator elements = A.GetEnumerator();

var data = new double[rows*cols];
i = 0;
while ( elements.MoveNext() )
{
  data[i++] = (double) elements.Current;
}
```

Code Example – VB matrix

```
Dim Rows As Integer = 13
Dim Cols As Integer = 3
Dim A As New DoubleMatrix(Rows, Cols, 0.0, 0.25)
Dim Elements As IEnumerator = A.GetEnumerator()

Dim Data(Rows * Cols) As Double

Dim I As Integer = 0
While Elements.MoveNext()
  I += 1
  Data(I) = CType(Elements.Current, Double)
End While
```

Note that the `Current` property on an **IEnumerator** returns the current object in the collection, which must then be cast to the appropriate type. **NMath** also provides custom strongly-typed enumerators: **IFloatEnumerator**, **IDoubleEnumerator**, **IFloatComplexEnumerator**, and **IDoubleComplexEnumerator**. These avoid casting, and are therefore much faster.

For instance:

Code Example – C# matrix

```
int rows = 13, cols = 3;
var A = new DoubleMatrix( rows, cols, 0, .25 );
IDoubleEnumerator elements = A.GetDoubleEnumerator();

var data = new double[rows*cols];
i = 0;
while ( elements.MoveNext() )
{
  data[i++] = elements.Current;       // No need to cast to double
}
```

Code Example – VB matrix

```
Dim Rows As Integer = 13
Dim Cols As Integer = 3
Dim A As New DoubleMatrix(Rows, Cols, 0.0, 0.25)
Dim Elements As IDoubleEnumerator = A.GetDoubleEnumerator()

Dim Data(Rows * Cols) As Double
Dim I As Integer = 0
While Elements.MoveNext()
  I += 1
  Data(I) = Elements.Current ' No need to cast to Double
End While
```

CHAPTER 7. SOLUTIONS OF LINEAR SYSTEMS

NMath provides classes for computing and storing the LU factorization for a matrix.

LU factorization is a procedure for decomposing a matrix into a product of a *lower* triangular matrix and an *upper* triangular matrix. Given a matrix `A`, an LU factorization class factors `A` as follows:

```
PA = LU
```

where `P` is a permutation matrix, `L` is a lower triangular matrix with ones on the diagonal, and `U` is an upper triangular matrix.

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.

NMath also provides several static functions for solving linear systems, and for computing determinants, inverses, and condition numbers.

7.1 Class Names

The classes that compute and store LU factorizations in **NMath** are named **<Type>LUFact**, where **<Type>** is **Float**, **Double**, **FloatComplex**, or **DoubleComplex**. (See Chapter 3 for a description of the complex number classes.) Thus:

- The **FloatLUFact** class represents the LU factorization of a matrix of single-precision floating point numbers.
- The **DoubleLUFact** class represents the LU factorization of a matrix of double-precision floating point numbers.
- The **FloatComplexLUFact** class represents the LU factorization of a matrix of single-precision complex numbers.
- The **DoubleComplexLUFact** class represents the LU factorization of a matrix of double-precision complex numbers.

7.2 Creating LU Factorizations

You can create an instance of an LU factorization class by supplying the constructor with a matrix to factor. Thus:

Code Example – C# LU factorization

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

Code Example – VB LU factorization

```
Dim A As New DoubleComplexMatrix(5, 5, 1, 1)
Dim LU As New DoubleComplexLUFact(A)
```

You can also use an existing instance to factor other matrices with the provided `Factor()` method. For instance:

Code Example – C# LU factorization

```
var A = new FloatMatrix( n, n, 1, 1.62F );
var lu = new FloatLUFact( A );

B = new FloatVector( n, -1.2F, 1.78F );
lu.Factor( B );
```

Code Example – VB LU factorization

```
Dim A As New FloatMatrix(N, N, 1, 1.62F)
Dim LU As New FloatLUFact(A)

Dim B As New FloatVector(N, -1.2F, 1.78F)
LU.Factor(B)
```

The read-only `IsGood` property gets a boolean value that is `true` if the matrix factorization succeeded and the factorization may be used to solve equations, compute determinants, inverses, and so on. Otherwise, it returns `false`. For example:

Code Example – C# LU factorization

```
if ( lu.IsGood )
{
  // Do something here...
}
```

Code Example – VB LU factorization

```
If LU.IsGood Then
  ' Do something here...
End If
```

Other read-only properties provide information about the matrix used to construct an LU factorization:

- `Cols` gets the number of columns of the factored matrix.
- `Rows` gets the number of rows of the factored matrix.
- `IsSingular` returns `true` if the matrix was singular; otherwise, `false`.

7.3 Using LU Factorizations

Once an LU factorization is constructed from a matrix (see Section 7.2), it can be reused to solve for different right hand sides, to compute inverses, to compute condition numbers, and so on.

Component Matrices

Read-only properties provide access to the component matrices of the LU factorization:

- `P` gets the permutation matrix.
- `L` gets the lower triangular matrix.
- `U` gets the upper triangular matrix.
- `Pivots` gets an array of pivot indices, where row `i` was interchanged with `Pivots[i]`.

Solving for Right-Hand Sides

You can use an LU factorization to solve for right-hand sides using the `Solve()` method. For instance, this code solves for one right-hand side.:

Code Example – C# LU factorization

```
var A = new DoubleMatrix( "3x3 [2 1 1 4 1 0 -2 2 1]" );
var lu = new DoubleLUFact( A );

var v = new DoubleVector( "[8 11 3]" );
DoubleVector x = lu.Solve( v );
```

Code Example – VB LU factorization

```
Dim A As New DoubleMatrix("3x3 [2 1 1 4 1 0 -2 2 1]")
Dim LU As New DoubleLUFact(A)
```

```
Dim V As New DoubleVector("[8 11 3]")
Dim X As DoubleVector = LU.Solve(V)
```

The returned vector `x` is the solution to the linear system `Ax = v`. Note that the length of vector `v` must be equal to the number of rows in the factored matrix `A` or a **MismatchedSizeException** is thrown. (See Section 53.1.)

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

Code Example – C# LU factorization

```
var A = new FloatMatrix( "3x3 [2 1 1  4 1 0 -2 2 1]" );
var lu = new FloatLUFact( A );

var B = new FloatMatrix( "3x2[8 3  11 11  3 8]" );
FloatMatrix X = fact.Solve( B );
```

Code Example – VB LU factorization

```
Dim A As New FloatMatrix("3x3 [2 1 1  4 1 0 -2 2 1]")
Dim LU As New FloatLUFact(A)

Dim B As New FloatMatrix("3x2[8 3  11 11  3 8]")
Dim X As FloatMatrix = Fact.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`.

`SolveInPlace()` methods are also provided which place the solution in the given vector or matrix, without allocating new memory. The given right-hand side data must have unit stride.

Computing Inverses, Determinants, and Condition Numbers

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

Code Example – C# LU factorization

```
var A = new FloatMatrix( "3x3 [2 1 1  4 1 0 -2 2 1]" );
var lu = new FloatLUFact( A );

FloatMatrix AInv = lu.Inverse();
float ADet = lu.Determinant();
```

Code Example – VB LU factorization

```
Dim A As New FloatMatrix("3x3 [2 1 1  4 1 0 -2 2 1]")
Dim LU As New FloatLUFact(A)
```

```
Dim AInv As FloatMatrix = LU.Inverse()
Dim ADet As Single = LU.Determinant()
```

The `ConditionNumber()` method computes the condition number in a specified norm type. The condition number of a matrix `A` is:

```
kappa = ||A|| ||AInv||
```

where `AInv` is the inverse of the matrix `A`.

NOTE—The ConditionNumber() method returns the reciprocal of the condition number, rho, where rho = 1/kappa.

The provided `NormType` enumeration contains values for specifying the matrix norm. You can also choose to estimate the condition number, which is faster but less accurate, or to compute it directly. For small matrices, the results are usually the same. Thus, this code estimates the condition number in the infinity-norm:

Code Example – C# LU factorization

```
var A = new DoubleMatrix( "3x3 [2 1 1   4 3 3   8 7 9 ]" );
var lu = new DoubleLUFact( A );

double AEstimatedConditionNum =
  lu.ConditionNumber( NormType.InfinityNorm, true );
```

Code Example – VB LU factorization

```
Dim A As New DoubleMatrix("3x3 [2 1 1   4 3 3   8 7 9 ]")
Dim LU As New DoubleLUFact(A)

Dim AEstimatedConditionNum As Double =
  LU.ConditionNumber(NormType.InfinityNorm, True)
```

This code computes the condition number directly in the 1-norm:

Code Example – C# LU factorization

```
double AComputedConditonNum =
   lu.ConditionNumber( NormType.OneNorm, false );
```

Code Example – VB LU factorization

```
Dim AComputedConditonNum As Double =
  LU.ConditionNumber(NormType.OneNorm, False)
```

7.4 Static Methods

As a convenience, **NMath** provides static methods on class **NMathFunctions** for solving linear systems, and for computing determinants, inverses, and condition numbers. All methods accept a matrix.

The following static methods are provided:

- `NMathFunctions.Solve()` solves linear systems for single or multiple right-hand sides.
- `NMathFunctions.Inverse()` computes the inverse of a given matrix.
- `NMathFunctions.Determinant()` computes the determinant of a given matrix.
- `NMathFunctions.EstimateConditionNumber()` estimates the condition number of a given matrix in the specified norm type.
- `NMathFunctions.ConditionNumber()` directly computes the condition number of a given matrix in the specified norm type.

For instance:

Code Example – C# LU factorization

```
var A = new DoubleMatrix( "3x3 [2 1 1  4 1 0 -2 2 1]" );

var b = new DoubleVector( "[8 11 3]" );
DoubleVector x = NMathFunctions.Solve( A, b );

var B = new DoubleMatrix( "3x2[8 3  11 11  3 8]" );
DoubleMatrix X = NMathFunctions.Solve( A, B );

DoubleMatrix AInv = NMathFunctions.Inverse( A );
double ADet = NMathFunctions.Determinant( A );
double ACond =
   NMathFunctions.ConditionNumber( A, NormType.InfinityNorm );
```

Code Example – VB LU factorization

```
Dim A As New DoubleMatrix("3x3 [2 1 1  4 1 0 -2 2 1]")

Dim B As New DoubleVector("[8 11 3]")
Dim X As DoubleVector = NMathFunctions.Solve(A, B)

Dim B As New DoubleMatrix("3x2[8 3  11 11  3 8]")
Dim X As DoubleMatrix = NMathFunctions.Solve(A, B)

Dim AInv As DoubleMatrix = NMathFunctions.Inverse(A)
Dim ADet As Double = NMathFunctions.Determinant(A)
```

```
Dim ACond As Double =
  NMathFunctions.ConditionNumber(A, NormType.InfinityNorm)
```

Note that an an LU factorization instance is created with each call to `NMathFunctions.Solve()`. If you are calling `Solve()` repeatedly (inside a loop, for example), and the coefficient matrix is not changing between calls, this is more efficient:

Code Example – C# LU factorization

```
var fact = new DoubleLUFact( A, false );
...
fact.Solve( B );
```

Code Example – VB LU factorization

```
Dim Fact As New DoubleLUFact(A, False)
...
Fact.Solve(B)
```

CHAPTER 8. LEAST SQUARES

NMath provides classes for computing the minimum-norm solution to a linear system $Ax = y$. In a linear model, a quantity y depends on one or more independent variables $a_1, a_2, ..., a_n$ such that $y = x_0 + x_1 a_1 + ... + x_n a_n$. (Parameter x_0 is called the *intercept parameter*.) The goal of a least squares problem is to solve for the best values of $x_0, x_1, ..., x_n$.

Several observations of the independent values a_i are recorded, along with the corresponding values of the dependent variable y. If m observations are performed, and for the *i*th observation we denote the values of the independent variables a_{i1}, $a_{i2}, ..., a_{in}$ and the corresponding dependent value of y as y_i, then we form the linear system $Ax = y$, where $A = (a_{ij})$ and $y = (y_i)$. The general least squares solution is the value of x that minimizes $||Ax - y||$. The *nonnegative* least squares solution is the value of x subject to the constraint that each element of x is nonnegative.

Note that if the model contains a non-zero intercept parameter, then the first column of A is all ones.

The **NMath** least squares classes use a complete orthogonal factorization of A to compute the solution. Matrix A is rectangular, and may be rank deficient.

8.1 Class Names

The classes that compute general least squares solutions in **NMath** are named **<Type>LeastSquares**, where **<Type>** is **Float**, **Double**, **FloatComplex**, or **DoubleComplex**. (See Chapter 3 for a description of the complex number classes.) Thus:

- The **FloatLeastSquares** class computes the least squares solution to the linear system $Ax = y$, where A is a **FloatMatrix** of independent observations, and y is a **FloatVector** of corresponding values for the dependent variable.
- The **DoubleLeastSquares** class computes the least squares solution to the linear system $Ax = y$, where A is a **DoubleMatrix** of independent

observations, and `y` is a **DoubleVector** of corresponding values for the dependent variable.

- The **FloatComplexLeastSquares** class computes the least squares solution to the linear system `Ax = y`, where `A` is a **FloatComplexMatrix** of independent observations, and `y` is a **FloatComplexVector** of corresponding values for the dependent variable.
- The **DoubleComplexLeastSquares** class computes the least squares solution to the linear system `Ax = y`, where `A` is a **DoubleComplexMatrix** of independent observations, and `y` is a **DoubleComplexVector** of corresponding values for the dependent variable.

The classes that compute nonnegative least squares solutions in **NMath** are named **<Type>NonnegativeLeastSquares**, where **<Type>** is **Float** or **Double—FloatNonnegativeLeastSquares** and **DoubleNonnegativeLeastSquares**.

8.2 Creating Least Squares Solutions

Least squares solutions to the linear system `Ax = y` are constructed from a rectangular matrix `A` and a vector of values `y`. For instance:

Code Example – C# least squares

```
var A =
   new DoubleMatrix( "4x2[1.0 20.0 1.0 30.0 40.0 1.0 50.0 1.0]" );
var y = new DoubleVector( "[.446 .601 .786 .928]" );

var lsq = new DoubleLeastSquares( A, y );
```

Code Example – VB least squares

```
Dim A =
  New DoubleMatrix("4x2[1.0 20.0 1.0 30.0 40.0 1.0 50.0 1.0]")
Dim Y = New DoubleVector("[.446 .601 .786 .928]")
Dim LSQ = New DoubleLeastSquares(A, Y)
```

An optional Boolean parameter to the constructor can be used to add an intercept parameter to the model. If `true`, a column of ones is prepended to a deep copy of matrix `A` before solving for the least squares solution.

NOTE—The input matrix A is not changed.

For example:

Code Example – C# least squares

```
var lsq = new FloatComplexLeastSquares ( A, y, true );
```

Code Example – VB least squares

```
Dim LSQ As New FloatComplexLeastSquares(A, Y, True)
```

Finally, for advanced users, you can specify a non-default tolerance to be used in computing the effective rank. The effective rank of `A` is determined by treating as zero those singular values that are less than the tolerance times the largest singular value.

Thus:

Code Example – C# least squares

```
double tolerance = 1e-5;
var lsq =
   new DoubleComplexLeastSquares( A, y, false, tolerance );
```

Code Example – VB least squares

```
Dim Tolerance As Double = "1e-5"
Dim LQS As New DoubleComplexLeastSquares(A, Y, False, Tolerance)
```

NOTE—For details of the effective rank computation, see the documentation for LAPACK routines sgelsy(), dgelsy(), zgelsy(), and cgelsy().

8.3 Using Least Squares Solutions

Once constructed, an **NMath** least squares class provides read-only properties to access the least squares solution to the linear system `Ax = y`:

- `X` gets the least squares solution.
- `Yhat` gets the predicted value `yHat = Ax`, where `x` is the calculated solution.
- `Residuals` gets the vector of residuals `r` where $r_i = y_i - yHat_i$.
- `ResidualSumOfSquares` gets the residual sum of squares $(y_0 - yHat_0)^2 + (y_1 - yHat_1)^2 + ... + (y_{m-1} - yHat_{m-1})^2$.
- `Rank` gets the effective rank of the matrix `A`.
- `Tolerance` gets the tolerance used to compute the effective rank of the matrix `A`.

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:

Code Example – C# least squares

```
var A = new DoubleMatrix( "5x1[20.0  30.0  40.0  50.0  60.0]" );
var y = new DoubleVector( "[.446 .601 .786 .928 .950]" );
var 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 );
Console.WriteLine( "Residual Sum of Squares (RSS) = {0}",
   lsq.ResidualSumOfSquares );
```

Code Example – VB least squares

```
Dim A As New DoubleMatrix("5x1[20.0  30.0  40.0  50.0  60.0]")
Dim Y As New DoubleVector("[.446 .601 .786 .928 .950]")
Dim LSQ As 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)
Console.WriteLine("Residual Sum of Squares (RSS) = {0}",
  LSQ.ResidualSumOfSquares)
```

8.4 Nonnegative Least Squares Solutions

Classes **FloatNonnegativeLeastSquares** and **DoubleNonnegativeLeastSquares** find nonnegative least squares solutions—that is, the value of `x` that minimizes `||Ax - y||` subject to the constraint that each element of the vector `x` is nonnegative.

The interface is the same as for the general least squares classes (Section 8.2 and Section 8.3), with the addition of a `RankDeficiencyDetected` property. If a rank deficiency is detected while solving an unconstrained least squares problem during the nonnegative least squares iterative algorithm, this property returns `true`.

CHAPTER 9. RANDOM NUMBER GENERATORS

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

NMath provides two sets of random number generators:

- *Scalar* random number generators, which generate random deviates one at a time, via the `Next()` method. All **NMath** scalar generators inherit from the abstract base class **RandomNumberGenerator**, providing a common interface.
- *Vectorized* random number generators, which yield a stream of random numbers. Vectorized random number generators generally outperform scalar generators in computations requiring multiple deviates. All **NMath** scalar generators implement the **IRandomNumberDistribution** interface, and use a **RandomNumberStream**.

This chapter describes how to use the random number generator classes.

9.1 Scalar Random Number Generators

NMath provides scalar generator classes that return random deviates from a variety of probability distributions.

Table 7 – Scalar Random Number Generators

Class	Description
RandGenUniform	Uniform distribution.
RandGenBeta	Beta distribution.
RandGenBinomial	Binomial distribution.
RandGenExponential	Exponential distribution.
RandGenGamma	Gamma distribution.
RandGenGeometric	Geometric distribution.

Table 7 – Scalar Random Number Generators

Class	Description
RandGenJohnson	Johnson distribution.
RandGenLogNormal	Log-normal distribution.
RandGenNormal	Normal distribution.
RandGenPareto	Pareto distribution.
RandGenPoisson	Poisson distribution.
RandGenTriangular	Triangular distribution.
RandGenWeibull	Weibull distribution.

Underlying Uniform Generators

All **NMath** scalar 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. Thus, the statistical properties and performance of the generators largely depend on the statistical properties and performance of the underlying random number generator.

By default, all scalar 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.

If you have your own uniform random number generator that you wish to use, all **NMath** random number generators provide constructor overloads that accept a `RandomNumberGenerator.UniformRandomNumber` function delegate. The function must generate uniform deviates in the range zero to one, and return a `double`.

For example, this code creates a delegate object from the method `System.Random.NextDouble()`, then constructs a binomial random number generator that uses this delegate:

Code Example – C# random number generators

```
var sysRand = new Random();
var uniformDeviates =
   new RandomNumberGenerator.UniformRandomNumber(
       sysRand.NextDouble );

int trials = 2000;
double prob = .002;
var binRand =
   new RandGenBinomial( trials, prob, uniformDeviates );
```

Code Example – VB random number generators

```
Dim SysRand As New Random()
Dim UniformDeviates As New
  RandomNumberGenerator.UniformRandomNumber(
    AddressOf SysRand.NextDouble)

Dim Trials As Integer = 2000
Dim Prob As Double = 0.002
Dim BinRand As New RandGenBinomial(Trials, Prob, UniformDeviates)
```

All generators inherit a `UniformDeviateMethod` property from **RandNumberGenerator** for accessing and modify the underlying delegate method. For example, this code changes the delegate used by `binRand`:

Code Example – C# random number generators

```
var mt = new RandGenMTwist( );
binRand.UniformDeviateMethod =
   new RandomNumberGenerator.UniformRandomNumber( mt.NextDouble );
```

Code Example – VB random number generators

```
Dim MT As New RandGenMTwist()
BinRand.UniformDeviateMethod =
  New RandomNumberGenerator.UniformRandomNumber(AddressOf
    MT.NextDouble)
```

Generating Random Numbers

All **NMath** generators provide a `Next()` method that returns a random deviate as a `double`, except for **RandGenBinomial** and **RandGenPoisson** that return an `int`. For example, this code prints out `100` random deviates from a normal distribution with mean of `-12.9` and variance of `2.066`:

Code Example – C# random number generators

```
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() );
}
```

Code Example – VB random number generators

```
Dim Mean As Double = -12.9
Dim Variance As Double = 2.066
Dim Gen As New RandGenNormal(Mean, Variance)

For I As Integer = 0 To 99
  Console.WriteLine(Gen.Next())
Next
```

The base class **RandomNumberGenerator** also provides the abstract method `NextDouble()`, which is equivalent to calling `Next()`. This is a common method for applications that require polymorphic random number generation across the different generators, but also incurs the extra overhead of a virtual function call.

The `Fill()` method enables you to fill an array of `float`, `double`, **FloatComplex**, or **DoubleComplex** with random values. Thus:

Code Example – C# random number generators

```
var array1 = new double[ 100 ];
var array2 = new FloatComplex[ 100 ];

var gen = new RandGenPoisson();
gen.Fill( array1 );
gen.Fill( array2 );
```

Code Example – VB random number generators

```
Dim Array1(100) As Double
Dim Array2(100) As FloatComplex

Dim Gen As New RandGenPoisson()
Gen.Fill(Array1)
Gen.Fill(Array2)
```

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

Code Example – C# random number generators

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

Code Example – VB random number generators

```
Dim Gen As New RandGenUniform(0, 100)
Dim V As New DoubleVector(10, Gen)
Dim A As New DoubleComplexMatrix(25, 25, Gen)
```

Random Seeds

As described above, all **NMath** random number generators, regardless of the distribution, use an underlying uniform random number generator to generate random deviates in the range `(0,1)`, then derive from the deviate a random number from the appropriate probability distribution. Thus, the seed that determines the pseudorandom sequence is associated with the underlying uniform generator, *not* with the wrapping generator.

All **NMath** random number generator classes have `Reset()` and `Reset(int)` methods that attempt to reset the underlying uniform generator with the time of day, for the no argument reset, or the given seed, for the integer argument version. These methods return `true` if the reset was successful and `false` if it was not. The reset methods succeed if the following conditions are met:

1. The uniform generator delegate is an instance method; that is, the `Target` property of the **Delegate** class returns a non-null reference.
2. The object reference thus obtained has a method named `Initialize()` that returns void and takes no arguments, for the `Reset()` method, or a single integer argument for the `Reset(int)` method.

For example, this code attempts to generate two identical sequences by explicitly setting and resetting the seed:

Code Example – C# random number generators

```
int seed = 0x124;
var mt = new RandGenMTwist( seed );
var uniformDeviates =
   new RandomNumberGenerator.UniformRandomNumber( mt.NextDouble );

var gen = new RandGenNormal( 50, 5, uniformDeviates );
var randomSequence1 = new DoubleVector( 100, gen );
```

```
if ( gen.Reset(seed) ) {
   var randomSequence2 = new DoubleVector( 100, gen );
}
else {
   Console.WriteLine( "Could not reset generator" );
}
```

Code Example – VB random number generators

```
Dim Seed As Integer = &H124
Dim MT As New RandGenMTwist(Seed)
Dim UniformDeviates As New
 RandomNumberGenerator.UniformRandomNumber(AddressOf MT.NextDouble)

Dim Gen As New RandGenNormal(50, 5, UniformDeviates)
Dim RandomSequence1 As New DoubleVector(100, Gen)

If Gen.Reset(Seed) Then
  Dim RandomSequence2 As New DoubleVector(100, Gen)
Else
  Console.WriteLine("Could not reset generator")
End If
```

9.2 Vectorized Random Number Generators

Unlike scalar-type generators, whose output is a successive random number (Section 9.1), vectorized generators produce a vector of *n* successive random numbers from a given distribution. Vectorized generators typically outperform scalar generators because the overhead expense of a function call is comparable to the total time required for computation.

NMath provides vectorized distribution classes for many continuous (Table 8) and discrete (Table 9) distributions.

Table 8 – Continuous Distributions

Class	Description
DoubleRandomBetaDistribution **FloatRandomBetaDistribution**	Beta distribution.
DoubleRandomCauchyDistribution **FloatRandomCauchyDistribution**	Cauchy distribution.

Table 8 – Continuous Distributions

Class	Description
DoubleRandomExponentialDistribution **FloatRandomExponentialDistribution**	Exponential distribution
DoubleRandomGammaDistribution **FloatRandomGammaDistribution**	Gamma distribution.
DoubleRandomGaussianDistribution **FloatRandomGaussianDistribution**	Gaussian distribution.
DoubleRandomGumbelDistribution **FloatRandomGumbelDistribution**	Gumbel distribution.
DoubleRandomLaplaceDistribution **FloatRandomLaplaceDistribution**	Laplace distribution.
DoubleRandomLogNormalDistribution **FloatRandomLogNormalDistribution**	Log-normal distribution.
DoubleRandomRayleighDistribution **FloatRandomRayleighDistribution**	Rayleigh distribution.
DoubleRandomUniformDistribution **FloatRandomUniformDistribution**	Uniform distribution.
DoubleRandomWeibullDistribution **FloatRandomWeibullDistribution**	Weibull distribution.

Table 9 – Discrete Distributions

Class	Description
IntRandomBernoulliDistribution	Bernoulli distribution.
IntRandomBinomialDistribution	Binomial distribution.
IntRandomGeometricDistribution	Geometric Distribution
IntRandomHypergeometricDistribution	Hypergeometric distribution.
IntRandomNegativeBinomialDistribution	Negative Binomial distribution.
IntRandomPoissonDistribution	Poisson distribution.

Table 9 – Discrete Distributions

Class	Description
IntRandomPoissonVaryingMeanDistribution	Possion distribution with varying mean.
IntRandomUniformDistribution	Uniform distribution.
IntRandomUniformBitsDistribution	Integer values with uniform bit distribution.

Distribution objects are constructed from the relevant distribution parameters. For example:

Code Example – C# random number generators

```
double mean = 1.0;
double sigma = 1.0;
DoubleRandomGaussianDistribution dist =
    new DoubleRandomGaussianDistribution(mean, sigma);
```

Code Example – VB random number generators

```
Dim Mean = 1.0
Dim Sigma = 1.0
Dim Dist As New DoubleRandomGaussianDistribution(Mean, Sigma)
```

Generating Random Numbers

Class **RandomNumberStream** encapsulates a vectorized random number generator which yields a stream of random numbers.

A stream is constructed from an optional seed, and an optional enumerated value specifying which algorithm to use for generating random numbers uniformly distributed in the interval `[0, 1]`.

Code Example – C# random number generators

```
int seed = 0x345;
var stream = new RandomNumberStream(seed,
  RandomNumberStream.BasicRandGenType.MersenneTwister);
```

Code Example – VB random number generators

```
Dim Seed As Integer = &H345
Dim Stream As New RandomNumberStream(Seed,
  RandomNumberStream.BasicRandGenType.MersenneTwister
```

You can use a stream and distribution to fill an array:

Code Example – C# random number generators

```
int n = 100;
int start = 0;
var a = new double[n];
dist.Fill(stream, a, start, n);
```

Code Example – VB random number generators

```
Dim N As Integer = 100
Dim Start As Integer = 0
Dim A(N) As Double
Dist.Fill(Stream, A, Start, N)
```

Or to fill a new vector or matrix:

Code Example – C# random number generators

```
var v = new DoubleVector(n, stream, dist);
```

Code Example – VB random number generators

```
Dim V As New DoubleVector(N, Stream, Dist)
```

Successive Random Numbers

If you want the performance of a vectorized random number generator, but still need to access the random deviates sequentially, **NMath** provides class **RandomNumbers**, which uses a stream to buffer the random numbers internally.

For instance:

Code Example – C# random number generators

```
int bufferSize = 100;
RandomNumbers<double, DoubleRandomGaussianDistribution> rnd =
  new RandomNumbers<double, DoubleRandomGaussianDistribution>(seed,
    dist, bufferSize);

for (int i = 0; i < 10; i++)
{
  Console.WriteLine("Next() = {0}", rnd.Next());
}
```

Code Example – VB random number generators

```
Dim BufferSize As Integer = 100
Dim RND As New RandomNumbers(Of Double,
  DoubleRandomGaussianDistribution)(Seed, Dist, BufferSize)

For I As Integer = 0 To 9
  Console.WriteLine("Next() = {0}", RND.Next())
Next
```

Independent Streams

NMath provides classes for generating several independent streams of random numbers using two methods:

- In the *leapfrog* method, the independent sequences are created by splitting the original sequence into *k* disjoint subsets, where *k* is the number of independent streams, is such a way that the first stream generates the random numbers x_1, x_{k+1}, x_{2k+1}, x_{3k+1},..., the second stream generates the numbers x_2, x_{k+2}, x_{2k+2}, x_{3k+2},..., and, finally, the k*th* stream would generate x_k, x_{2k}, x_{3k}... Class **LeapfrogRandomStreams** uses the leapfrog method.
- In the *skip-ahead*, or *block-splitting*, method, the independent sequences are created by splitting the original sequence into k non-overlapping blocks, where *k* is the number of independent streams. Each stream generates numbers only from its corresponding block. Class **SkipAheadRandomStreams** uses the skip-ahead method.

For example, this code creates `10` streams of length `100` using the skip-ahead method:

Code Example – C# random number generators

```
int seed = 0x124;
RandomNumberStream.BasicRandGenType genType =
    RandomNumberStream.BasicRandGenType.MultiplicativeCongruent31;
int nstreams = 10;
int streamLen = 100;
SkipAheadRandomStreams gen =
    new SkipAheadRandomStreams(seed, genType, nstreams, streamLen);
```

Code Example – VB random number generators

```
Dim Seed = &H124
Dim GenType =
  RandomNumberStream.BasicRandGenType.MultiplicativeCongruent31
Dim NStreams = 10
Dim StreamLen = 100
Dim Gen As New SkipAheadRandomStreams(Seed, GenType, NStreams,
  StreamLen)
```

You can use a single distribution to fill an array or matrix:

Code Example – C# random number generators

```
var dist = new DoubleRandomUniformDistribution();
var A = new DoubleMatrix(streamLen, nstreams);
gen.Fill(dist, A);
```

Code Example – VB random number generators

```
Dim Dist As New DoubleRandomUniformDistribution()
Dim A As New DoubleMatrix(StreamLen, NStreams)
Gen.Fill(Dist, A)
```

Or to create a new matrix:

Code Example – C# random number generators

```
var dist = new DoubleRandomLogNormalDistribution();
DoubleMatrix B = gen.Next(dist);
```

Code Example – VB random number generators

```
Dim Dist As New DoubleRandomLogNormalDistribution()
Dim B As DoubleMatrix = Gen.Next(Dist)
```

You can also use an array of distributions, one per stream:

Code Example – C# random number generators

```
nstreams = 3;
var intDists = new IRandomNumberDistribution<double>[nstreams];
intDists[0] = new DoubleRandomUniformDistribution();
intDists[1] = new DoubleRandomBetaDistribution();
intDists[2] = new DoubleRandomCauchyDistribution();
var gen = new SkipAheadRandomStreams(seed, genType, nstreams,
  streamLen);
DoubleMatrix C = gen.Next(intDists);
```

Code Example – VB random number generators

```
nstreams = 3
Dim IntDists(NStreams) As IRandomNumberDistribution(Of Double)
IntDists(0) = New DoubleRandomUniformDistribution()
IntDists(1) = New DoubleRandomBetaDistribution()
IntDists(2) = New DoubleRandomCauchyDistribution()
Dim Gen = New SkipAheadRandomStreams(Seed, GenType, NStreams,
  StreamLen)
Dim C As DoubleMatrix = Gen.Next(IntDists)
```

Quasirandom Numbers

NMath provides classes for generating sequences of *quasirandom* points. A quasirandom sequence is a sequence of n-tuples that fills *n*-space more uniformly than uncorrelated random points. **NiederreiterQuasiRandomGenerator** generates quasirandom numbers using the Niederreiter method; **SobolQuasiRandomGenerator** uses the Sobol method.

For example:

Code Example – C# quasirandom numbers

```
int dim = 3;
var nqrg = new NiederreiterQuasiRandomGenerator(dim);
```

Code Example – VB quasirandom numbers

```
Dim Dimensions As Integer = 3
Dim NQRG As New NiederreiterQuasiRandomGenerator(Dimensions)
```

You can fill an existing matrix or array. (The points are the columns of the matrix, so the number of rows in the given matrix must be equal to the `Dimension` of the quasirandom number generator.)

Code Example – C# quasirandom numbers

```
int numPts = 5000;
var A = new DoubleMatrix(nqrg.Dimension, numPts);
nqrg.Fill(A);
```

Code Example – VB quasirandom numbers

```
Dim NumPts As Integer = 5000
Dim A As New DoubleMatrix(NQRG.Dimension, NumPts)
NQRG.Fill(A)
```

Or create a new matrix:

Code Example – C# quasirandom numbers

```
DoubleMatrix B = nqrg.Next(
  new DoubleRandomUniformDistribution(), numPts);
```

Code Example – VB quasirandom numbers

```
Dim B As DoubleMatrix = NQRG.Next(
  New DoubleRandomUniformDistribution(), NumPts)
```

The quasirandom numbers will follow the given distribution.

CHAPTER 10.
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. This chapter describes how to use the FFT, convolution, and correlation classes.

10.1 Fast Fourier Transforms

Fast Fourier Transforms (FFTs) are efficient algorithms for calculating the discrete fourier transform (DFT) and its inverse. **NMath** provides classes for performing FFTs on real and complex 1D and 2D data.

FFT Classes

The classes that perform FFTs in **NMath** are named in the form **<Type><Direction><Dimensionality>FFT**, where

- **<Type>** is **Float**, **Double**, **FloatComplex**, or **DoubleComplex** based on the precision of the data and the forward domain of the FFT, either real or complex.
- **<Direction>** is **Forward** for calculating the DFT, and **Backward** for calculating its inverse.
- **<Dimensionality>** is **1D** or **2D**, depending on the dimensionality of the signal data.

For example, class **DoubleForward2DFFT** performs the forward DFT on 2D double-precision real signal data. Class **FloatComplexBackward1DFFT** represents the backward DFT of a 1D single-precision complex signal vector.

This set of classes elegantly supports all common 1D and 2D FFT computations in a robust, easy to use, object-oriented interface.

Creating FFT Instances

FFT instances are constructed by specifying the size of the signal data. For example, this code constructs a **DoubleForward1DFFT** for a signal vector of length `1024`:

Code Example – C# FFT

```
var fft = new DoubleForward1DFFT( 1024 );
```

Code Example – VB FFT

```
Dim FFT As New DoubleForward1DFFT(1024)
```

This creates a **DoubleComplexBackward2DFFT** for a `500 x 500` data matrix:

Code Example – C# FFT

```
var fft = new DoubleComplexBackward2DFFT( 500, 500 );
```

Code Example – VB FFT

```
Dim FFT As New DoubleComplexBackward2DFFT(500, 500)
```

FFT instances can also be created by copying the configuration from another FFT instance. For example:

Code Example – C# FFT

```
var fft2 = new FloatForward1DFFT( fft1 );
```

Code Example – VB FFT

```
Dim FFT2 As New FloatForward1DFFT(FFT1)
```

An **NMathFormatException** is raised if the given FFT is not of a compatible precision, domain, and dimensionality. You can, however, create a forward FFT from a backward FFT instance, and *vice versa*.

Scale Factors

FFT classes provide properties for setting the scale factor of the FFT:

- Forward FFT classes provide a `ForwardScaleFactor` property.
- Backward FFT classes provide a `BackwardScaleFactor` property.

The default scale factor is `1.0`. This code sets the scale factor on a **DoubleForward1DFFT** instance to `2.0`:

Code Example – C# FFT

```
var fft = new DoubleForward1DFFT( 1024 );
fft.ForwardScaleFactor = 2.0;
```

Code Example – VB FFT

```
Dim FFT As New DoubleForward1DFFT(1024)
FFT.ForwardScaleFactor = 2.0
```

As a convenience, backward FFT classes also provide a `SetScaleFactorByLength()` method which sets the scale factor to the inverse of the signal length. If the forward FFT scale factor is `1.0`, using this backward scale factor guarantees that `backwardFFT(forwardFFT(signal)) = signal`. Note that MATLAB uses this scale factor by default.

Computing FFTs

FFTs can be computed either *in place*, overwriting the input data, or with the result placed in a separate, pre-allocated data structure passed by reference.

The `FFTInPlace()` method computes the FFT in place, while the `FFT()` method places the result in a second data structure. For example, this code compute an FFT in place:

Code Example – C# FFT

```
var data = new DoubleVector( 1024, new RandGenUniform() );
var fft = new DoubleForward1DFFT( 1024 );
fft.FFTInPlace( data );
```

Code Example – VB FFT

```
Dim data As New DoubleVector(1024, New RandGenUniform())
Dim FFT As New DoubleForward1DFFT(1024)
FFT.FFTInPlace(data)
```

This code places the result in a second data structure:

Code Example – C# FFT

```
var data = new FloatMatrix( 5, 5, new RandGenUniform() );
var result = new FloatMatrix( 5, 5 );
var fft = new FloatForward2DFFT( 5, 5 );
fft.FFT( data, ref result );
```

Code Example – VB FFT

```
Dim Data As New FloatMatrix(5, 5, New RandGenUniform())
Dim Result As New FloatMatrix(5, 5)
Dim FFT As New FloatForward2DFFT(5, 5)
FFT.FFT(data, Result);
```

Data can be supplied either using **NMath** vector and matrix types, or using arrays. For **NMath** types, an offset into the data can be specified on the vector or matrix instance. For arrays, a separate integer offset may be passed to the FFT methods.

NOTE—In general, the FFT classes require that all input signal data be in contiguous (packed) storage—that is, have positive or negative unit stride. More complex memory layouts can be handled with class DoubleGeneral1DFFT (see "Strided Signals" below).

Unpacking Real Results

Results from computing an FFT on real signal data are returned in MKL `Pack` format[1], a compact representation of a complex conjugate-symmetric sequence. The result is the same size as the original signal data.

For convenience, *reader* classes are provided for unpacking the results. The FFT instance used to generated the result can be queried for the appropriate reader using the `GetSignalReader()` method. This guarantees that the correct packed signal reader is constructed.

For example:

Code Example – C# FFT

```
var data = new DoubleVector( 1024, new RandGenUniform() );
var fft = new DoubleForward1DFFT( 1024 );
fft.FFTInPlace( data );
DoubleSymmetricSignalReader reader = fft.GetSignalReader( data );
```

Code Example – VB FFT

```
Dim Data As New DoubleVector(1024, New RandGenUniform())
Dim FFT As New DoubleForward1DFFT(1024)
FFT.FFTInPlace(Data)
Dim Reader As DoubleSymmetricSignalReader =
FFT.GetSignalReader(Data)
```

Readers provide random access to any element in the pack FFT result:

Code Example – C# FFT

```
DoubleComplex thirdelement = reader[2];
```

Code Example – VB FFT

```
Dim ThirdElement As DoubleComplex = Reader(2)
```

Reader classes also provide methods for unpacking the entire result:

- The *full* unpack methods—such as `UnpackFullToArray()` and `UnpackFullToMatrix()`—build the unpacked signal representation of the entire packed complex symmetric signal.
- The *symmetric half* unpack methods—such as `UnpackSymmetricHalftoArray()` and

[1]"Packed Formats", *Intel Math Kernel Library Reference Manual*, September 2007, pp. 2554-2559.

`UnpackSymmetricHalfToMatrix()`—build the unpacked signal representation of the symmetric leading half of the packed signal.

For instance, this code unpacks the entire signal:

Code Example – C# FFT

```
DoubleSymmetric2DSignalReader reader =
  fft.GetSignalReader( ref result );

DoubleComplexMatrix unpacked = reader.UnpackFullToMatrix();
```

Code Example – VB FFT

```
Dim Reader As DoubleSymmetric2DSignalReader =
  FFT.GetSingalReader(Result)

Dim Unpacked As DoubleComplexMatrix = reader.UnpackFullToMatrix()
```

NOTE—Complex FFTs do not create packed results. The result is already the same size as the signal data.

Inverting Real Results

Results from computing an FFT on real signal data are returned in symmetric complex conjugate form, and NMath provides special classes for inverting this data back to the real domain.

For example, this code computes a forward FFT on real 1D signal data:

Code Example – C# FFT

```
var data = new DoubleVector( "[ 1 2 2 1 ]" );
var result = new DoubleVector( 4) ;

var fft = new DoubleForward1DFFT( 4 );
fft.FFT( data, ref result );
```

Code Example – VB FFT

```
Dim Data As New DoubleVector("[ 1 2 2 1 ]")
Dim Result As New DoubleVector(4)

Dim FFT As New DoubleForward1DFFT(4)
FFT.FFT(data, Result)
```

This code uses class **DoubleSymmetricBackward1DFFT** to invert the result:

Code Example – C# FFT

```
var reverse = new DoubleVector( 4 );

var rfft = new DoubleSymmetricBackward1DFFT( 4 );
rfft.SetScaleFactorByLength();
rfft.FFT( result, ref reverse );
```

Code Example – VB FFT

```
Dim Reverse As New DoubleVector(4)

Dim RFFT As New DoubleSymmetricBackward1DFFT(4)
RFFT.SetScaleFactorByLength()
RFFT.FFT(Result, Reverse)
```

Symmetric backward FFT classes, such as **DoubleSymmetricBackward1DFFT**, exploit the complex conjugate symmetry of the forward FFT result. The scaling is necessary for `reverse` to match `data`. (See "Scale Factors"above.)

Strided Signals

In general, the FFT classes require that all input signal data be in contiguous (packed) storage—that is, have unit stride. When working with strided signals, the FFT must be configured separately, and then used to create an advanced general FFT instance.

NOTE—Strided signals are supported for 1D signals only.

For example, suppose we have the following signal data, and wish to perform an FFT on the subset of the data specified by an offset of `3` and a stride of `2`:

Code Example – C# FFT

```
double[] data =
  { 94423, -341, 42343, 1, -1, 2, -1, 2, -1, 1, -85, 22 };
```

Code Example – VB FFT

```
Dim Data() As Double =
  {94423.0, -341.0, 42343.0, 1.0, -1.0, 2.0, -1.0, 2.0, -1.0, 1.0,
  -85.0, 22.0}
```

The desired subset has a length of `4`. To perform an FFT on this subset:

1. Build an **FFTConfiguration** instance which describes the FFT to be computed, including the stride and offset:

Code Example – C# FFT

```
int dimension = 1;
int length = 4;
var config = new FFTConfiguration(
  FFTDirection.FORWARD,
  FFTPrecision.DOUBLE,
  FFTDomain.REAL,
  dimension,
  length );
configcomplex.DataOffset = 3;
configcomplex.DataStride = 2;
configcomplex.InPlace = true;
```

Code Example – VB FFT

```
Dim Dimension As Integer = 1
Dim Length As Integer = 4
Dim Config As New FFTConfiguration(
  FFTDirection.FORWARD,
  FFTPrecision.DOUBLE,
  FFTDomain.REAL,
  Dimension,
  Length)
ConfigComplex.DataOffset = 3
ConfigComplex.DataStride = 2
ConfigComplex.InPlace = True
```

2. Build a **DoubleGeneral1DFFT** instance from the configuration:

Code Example – C# FFT

```
var fft = new DoubleGeneral1DFFT( ref config );
```

Code Example – VB FFT

```
Dim FFT As New DoubleGeneral1DFFT(Config)
```

3. Create an array to hold the result, and compute the FFT:

Code Example – C# FFT

```
var result = new double[4];
fft.FFT( signal, ref result );
```

Code Example – VB FFT

```
Dim Result(4) As Double
FFT.FFT(Signal, Result)
```

Class **DoubleGeneral1DFFT** is intended for advanced users. If the provided configuration does not correctly match the layout and type of the input signal data, exceptions and erroneous outputs will result.

10.2 Convolution and Correlation

Convolution is used to linearly filter a signal The convolution *z(n)* of two discrete input sequences *x(n)* and *y(n)* is defined as:

$$z(k) = \sum_{i} x(j)y(k-j)$$

Mathematically, the two convolved vectors, *x* and *y*, can be interchanged without changing the convolution result, *z*. In practice, however, one vector, called the convolution *kernel*, is often much shorter than the other and is typically used in many convolution operations against different data sets. The kernel can be thought of as a moving window scanned across the data vector. The output value is the weighted sum of the data within the window multiplied by the kernel. Where necessary, the sum is computed by padding the edges of the data with zeros. If the data is of length *m* and the kernel is of length *n*, then the output is of length *m+n-1*.

Correlation is used to characterize the statistical similarity between two signals. The operation is very similar to convolution, in that correlation uses two signals to produce a third signal, called the *cross-correlation*, or, if a signal is correlated with itself, the *autocorrelation*. The correlation is defined as:

$$z(k) = \sum_{i} x(j)y(k+j)$$

NMath provides classes for performing linear convolutions on real and complex 1D data. The API is

Convolution and Correlation Classes

The classes that perform 1D convolution and correlation in **NMath** are named **<Type>1DConvolution** and **<Type>1DCorrelation**, respectively, where **<Type>** is **Float**, **Double**, **FloatComplex**, or **DoubleComplex**. For example, class **Double1DConvolution** performs convolutions of two 1D sequences of double-precision floating point values.

Creating Convolution and Correlation Instances

Convolution and correlation instances are constructed by specifying the kernel and the length of the data vector. For example, this code constructs a **Double1DConvolution** for a kernel of length `5`, representing a moving average, and data vector of length `1024`:

Code Example – C# FFT

```
var kernel = new DoubleVector( ".2 .2 .2 .2 .2" );
int dataLength = 1024;
Double1DConvolution conv =
  new Double1DConvolution( kernel, dataLength );
```

Code Example – VB FFT

```
Dim Kernel As New DoubleVector(".2 .2 .2 .2 .2")
Dim DataLength = 1024
Dim Conv As New Double1DConvolution(Kernel, DataLength)
```

The kernel can be supplied either using an **NMath** vector or an array. For an **NMath** vector, a kernel offset and stride can be specified on the vector instance. For an array, a separate integer kernel offset and stride may be passed to the constructor:

Code Example – C# FFT

```
var kernel = new DoubleVector( "-1 .2 -1 .2 -1 .2" );
int kernelOffset = 1;
int kernelStride = 2;
int dataLength = 1024;
var corr = new Double1DCorrelation( kernel, kernelOffset,
  kernelStride, dataLength );
```

Code Example – VB FFT

```
Dim Kernel As New DoubleVector("-1 .2 -1 .2 -1 .2")
Dim KernelOffset = 1
Dim KernelStride = 2
Dim DataLength = 1024
Dim Corr As New Double1DCorrelation(Kernel, KernelOffset,
  KernelStride, DataLength)
```

Convolution and Correlation Properties

Once constructed, an **NMath** convolution or correlation object provides the following read-only properties:

- `KernelLength` gets the length of the kernel.
- `DataLength` gets the expected convolution or correlation data length.

- `Length` gets the length of the output convolution or correlation. The output length equals `DataLength + KernelLength - 1`.

Computing Convolutions and Correlations

The `Convolve()` method computes the convolution and the `Correlate()` method computes the correlation between the stored kernel, and a given data vector. For example:

Code Example – C# FFT

```
var data = new FloatVector( 500, new RandGenUniform() );
FloatVector result = corr.Correlate( data );
```

Code Example – VB FFT

```
Dim Data As New FloatVector(500, New RandGenUniform())
Dim Result As FloatVector = Corr.Correlate(Data)
```

An **InvalidArgumentException** is raised if the length of the given data does not match the data length previously specified in the constructor.

If you are performing multiple convolutions or correlations using the same object—within a loop, for example—you can reuse the same pre-allocated vector to hold the result:

Code Example – C# FFT

```
var data = new FloatVector( 500, new RandGenUniform() );
var result = new FloatVector( corr.Length );
corr.Correlate( data, ref result );
```

Code Example – VB FFT

```
Dim Data As New FloatVector(500, New RandGenUniform())
Dim Result As New FloatVector(Corr.Length)
Corr.Correlate(Data, Result)
```

Windowing Options

The `Convolve()` and `Correlate()` methods compute the full result, with length `DataLength + KernelLength - 1`. Boundary values, where the kernel partially overlaps the data, are computed by padding the edges of the data with zeros. The `TrimConvolution()` and `TrimCorrelation()` methods creates a clipped view into a given result, using the specified `Windowing` option:

- `Windowing.Unwindowed` (the default) retrieves the full result.
- `Windowing.CenterWindow` clips the result to the length of the data, shifted to the center.

- `Windowing.FullKernelOverlap` returns the data portion that entirely overlaps the kernel.

For instance:

Code Example – C# FFT

```
DoubleVector result = conv.Convolve( data );
DoubleVector trimmed = conv.TrimConvolution( result,
  CorrelationBase.Windowing.FullKernelOverlap );
```

Code Example – VB FFT

```
Dim Result As DoubleVector = Conv.Convolve(Data)
Dim Trimmed As DoubleVector = Conv.TrimConvolution(result,
  CorrelationBase.Windowing.FullKernelOverlap)
```

No data is copied. The returned vector is a view into the same data referenced by the given result.

CHAPTER 11. DISCRETE WAVELET TRANSFORMS

A *wavelet* is a wave-like oscillation, which integrates to zero and is well-localized in time. A *discrete wavelet transform* (DWT) is any wavelet transform for which the wavelets are discretely sampled. DWT captures both frequency and location information, an important advantage over FFT (Chapter 10).

DWTs have found engineering applications in computer vision, pattern recognition, signal filtering and perhaps most widely in signal and image compression. In 2000 the ISO JPEG committee proposed a new JPEG2000 image compression standard that is based on the wavelet transform using two Daubechies wavelets. This standard made the relatively new image decomposition algorithm ubiquitous on desktops around the world.

NMath provides classes for performing DWT 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.

11.1 Creating Wavelets

NMath provides classes for creating wavelet objects: **FloatWavelet** and **DoubleWavelet**. Each derives from an abstract **Wavelet** base class.

Wavelets are constructed by specifying the wavelet family, using a value from the `Wavelet.Wavelets` enum. Fives types of built-in wavelets are supported: Harr, Daubechies, Least Asymmetric, Best Localized, and Coiflet. Built-in wavelets are identified by short name: the first letter abbreviates the wavelet family name, and the number that follows indicates the wavelet length. For example, this code builds a single-precision Coiflet wavelet of length `4`:

Code Example – C# Wavelet

```
var wavelet = new FloatWavelet( Wavelet.Wavelets.C4 );
```

Code Example – VB Wavelet

```
Dim WaveletInstance As New FloatWavelet(Wavelet.Wavelets.C4)
```

Custom wavelets can also be created by passing in the wavelet's low and high pass decimation filter values. The wavelet class then imposes the wavelet's symmetry properties to compute the reconstruction filters.

For example, this code builds a custom reverse bi-orthogonal wavelet:

Code Example – C# Custom Wavelet

```
var low = new double[] {0.0, 0.0, 0.7071068, 0.7071068, 0.0, 0.0};
var high = new double[] {0.0883883, 0.0883883, -0.7071068,
                         0.7071068, -0.0883883, -0.0883883};
var wavelet = new DoubleWavelet( low, high );
```

Code Example – VB Custom Wavelet

```
Dim Low = New Double() {0.0, 0.0, 0.7071068, 0.7071068, 0.0, 0.0}
Dim High = New Double() {0.0883883, 0.0883883, -0.7071068,
                         0.7071068, -0.0883883, -0.0883883}
Dim WaveletInstance As New DoubleWavelet(Low, High)
```

After creating a wavelet object, you can access various properties of the wavelet:

- `FamilyName` gets the wavelet family name (long-form), or `Custom` in the case of a custom wavelet.
- `ShortName` gets the wavelet name abbreviation.
- `Length` gets the length of the wavelet.
- `HighDecFilter` gets the high-pass decimation filter values.
- `LowDecFilter` gets the low-pass decimation filter values.
- `HighRecFilter` gets the high-pass reconstruction filter values.
- `LowRecFilter` gets the low-pass reconstruction filter values.

11.2 Computing Discrete Wavelet Transforms

As with Fourier analysis (Chapter 10), there are three basic steps to filtering signals using wavelets:

- Decompose the signal using the DWT.
- Filter the signal in the wavelet space using thresholding.
- Invert the filtered signal to reconstruct the original, now filtered signal, using the inverse DWT.

The filtering of signals using wavelets is based on the idea that as the DWT decomposes the signal into details and approximation parts, at some scale the details contain mostly insignificant noise and can be removed or zeroed out using thresholding without affecting the signal.

In **NMath**, classes **FloatDWT** and **DoubleDWT** perform discrete wavelet transforms. Both derive from the **DiscreteWaveletTransform** abstract base class. DWT classes support both single step forward and reverse DWTs and multilevel signal deconstruction and reconstruction.

Instances of DWT types are constructed from signal data and a wavelet instance (Section 11.1). For example:

Code Example – C# DWT

```
var data = new DoubleVector( 26, new RandGenNormal( 1.0, 1.0 ) );
var wavelet = new DoubleWavelet( Wavelet.Wavelets.D2 );
var dwt = new DoubleDWT( data.DataBlock.Data, wavelet );
```

Code Example – VB DWT

```
Dim Data As New DoubleVector(26, New RandGenNormal(1.0, 1.0))
Dim WaveletInstance As New DoubleWavelet(Wavelet.Wavelets.D2)
Dim DWT As New DoubleDWT(Data.DataBlock.Data, waveletInstance)
```

An edge management mode can also be specified using values from the `DiscreteWaveletTransform.WaveletMode` enum. The default value is `WaveletMode.PeriodicPadding`.

Single Step DWT

For convenience, DWT classes provide `DWT()` and `IDWT()` methods for performing single-step forward and reverse DWTs. For example, this code performs a single-step deconstruction and reconstruction.

Code Example – C# Single-Step DWT

```
// Decompose signal with DWT
double[] approx;
double[] details;
dwt.DWT( data.DataBlock.Data, out approx, out details );

// Rebuild the signal
double[] signal = dwt.IDWT( approx, details );
```

Code Example – VB Single-Step DWT

```
' Decompose signal with DWT
Dim Approx() As Double
Dim Details() As Double
DWT.DWT(Data.DataBlock.Data, Approx, Details)
```

```
' Rebuild the signal
Dim Signal As Double() = DWT.IDWT(Approx, Details)
```

Multilevel DWT

The `Decompose()` method performs a multilevel discrete wavelet decomposition at a specified level. For instance:

Code Example – C# Multilevel DWT

```
dwt.Decompose( 5 );
```

Code Example – VB Multilevel DWT

```
DWT.Decompose(5)
```

`MaximumDecompLevel()` provides the maximum number of DWT decompositions possible based on the signal and wavelet lengths. `CurrentDecompLevel()` provides the current maximum level to which this signal has been decomposed.

The `Reconstruct()` method performs a multilevel discrete wavelet reconstruction at a specified level. A signal decomposition must be first completed. If no level is specified, a complete reconstruction is performed. For example, this code rebuilds the signal to level `2`:

Code Example – C# Multilevel DWT

```
double[] reconstructedData2 = dwt.Reconstruct( 2 );
```

Code Example – VB Multilevel DWT

```
Dim ReconstructedData2() As Double = DWT.Reconstruct(2)
```

This code rebuilds the signal to level `1`—the original (filtered) signal.

Code Example – C# Multilevel DWT

```
double[] reconstructedData1 = dwt.Reconstruct();
```

Code Example – VB Multilevel DWT

```
Dim ReconstructedData1() As Double = DWT.Reconstruct()
```

Accessing the Coefficients

After a signal decomposition is completed, the coefficient vectors can be accessed. The `WaveletCoefficients()` method takes the wavelet coefficient type, either details or approximation, and the detail level desired, starting with level `1` and continuing to the maximum level of decomposition completed (similar to MATLAB's `wrcoef` function). Depending on the length of the wavelet and signal vector the approximations may have an extra element at the end of the vector due to the IDWT.

Code Example – C# Wavelet Coefficients

```
var approx = dwt.WaveletCoefficients(
  DiscreteWaveletTransform.WaveletCoefficientType.Details, 2 );
```

Code Example – VB Wavelet Coefficients

```
Dim Approx() As Double = DWT.WaveletCoefficients(
  DiscreteWaveletTransform.WaveletCoefficientType.Details, 2)
```

Threshold Calculations

`ComputeThreshold()` finds a single threshold for a given thresholding method and decomposition level. Four different thresholding methods are supported: Universal, UniversalMAD, Sure, and Hybrid (also known as SureShrink).

For example, this code computes the Universal threshold at level 1:

Code Example – C# Wavelet Threshold Calculation

```
double lambdaU = dwt.ComputeThreshold(
  DiscreteWaveletTransform.ThresholdMethod.Universal, 1 );
```

Code Example – VB Wavelet Threshold Calculation

```
Dim LambdaU As Double = DWT.ComputeThreshold(
  DiscreteWaveletTransform.ThresholdMethod.Universal, 1)
```

Thresholding

NMath supports details thresholding at any level.

`ThresholdAllLevels()` thresholds all levels of detail in the current signal decomposition. The method accepts a thresholding policy from the `DiscreteWaveletTransform.ThresholdPolicy` enum, and a vector of threshold values, with the first value applied to level `1`, the second applied to level `2`, and so on. The length of the threshold vector must be at least the depth of the current decomposition as indicated by `CurrentDecompLevel()`.

For example, this code thresholds all detail levels using the same threshold with a Soft policy:

Code Example – C# Wavelet Thresholding

```
dwt.ThresholdAllLevels(
  DiscreteWaveletTransform.ThresholdPolicy.Soft,
  new double[] { lambdaU, lambdaU, lambdaU, lambdaU, lambdaU } );
```

Code Example – VB Wavelet Thresholding

```
DWT.ThresholdAllLevels(
  DiscreteWaveletTransform.ThresholdPolicy.Soft,
  New Double() {LambdaU, LambdaU, LambdaU, LambdaU, LambdaU})
```

`ThresholdLevel()` thresholds the specified details level in the current signal decomposition.

CHAPTER 12. HISTOGRAMS

In **NMath**, instances of the **Histogram** class construct and maintain a histogram of input data. Input data is sorted into bins, and a count is kept of how many data points fall into each bin.

12.1 Creating Histograms

The **Histogram** class provides various methods for defining the bins into which input data will be sorted. For example, you can create a histogram with a specified number of equal-sized bins spanning specified maximum and minimum values. Thus, this code creates a histogram with `10` equal-sized bins spanning `0.0` to `100.0`:

Code Example – C# histogram

```
var hist = new Histogram( 10, 0.0, 100.0 );
```

Code Example – VB histogram

```
Dim Hist As New Histogram(10, 0.0, 100.0)
```

The first `n-1` bins are closed with respect to the lower bound, but open with respect to the upper bound. For instance, in the histogram created above, the first bin includes `0.0` but excludes `10.0`, the second bin includes `10.0` but excludes `20.0`, and so forth. The final bin is closed with respect to both upper and lower bounds. Thus, in the code above, the last bin includes both `90.0` and `100.0`.

If you do not wish to create equal-sized bins, you can create a **Histogram** from a vector of bin boundaries. Bin boundaries must be strictly monotonically increasing; that is, `binBoundares[i]` must be strictly less than `binBoundaries[i+1]` for each `i`. For example, this constructs a histogram with `3` unequal-sized bins spanning `0.0` to `100.0`:

Code Example – C# histogram

```
var v = new DoubleVector( “0.0 25.0 75.0 100.0” );
var hist = new Histogram( v );
```

Code Example – VB histogram

```
Dim V As New DoubleVector("0.0 25.0 75.0 100.0")
Dim Hist As New Histogram(V)
```

Again, the first `n-1` bins are closed with respect to the lower bound, but open with respect to the upper bound. The final bin is closed with respect to both upper and lower bounds.

Finally, for complete control, you can create a **Histogram** from an array of **Interval** objects. An **Interval** represents a numeric interval with inclusive or exclusive lower and upper bounds. The **Interval** constructor accepts a lower and upper bound, plus a value from the **Interval.Type** enumeration indicated whether the interval is open or closed with respect to each boundary. Thus:

Code Example – C# histogram

```
// (0,10)
var i1 = new Interval(  0, 10, Interval.Type.OpenOpen );

// [0,10)
var i1 = new Interval(  0, 10, Interval.Type.ClosedOpen );

// (0,10]
var i1 = new Interval(  0, 10, Interval.Type.OpenClosed );

// [0,10]
var i1 = new Interval(  0, 10, Interval.Type.ClosedClosed );
```

Code Example – VB histogram

```
' (0,10)
Dim I1 As New Interval(0, 10, Interval.Type.OpenOpen)

' [0,10)
Dim I1 As New Interval(0, 10, Interval.Type.ClosedOpen)

' (0,10]
Dim I1 As New Interval(0, 10, Interval.Type.OpenClosed)

' [0,10]
Dim I1 As New Interval(0, 10, Interval.Type.ClosedClosed)
```

A **Histogram** can be created from an array of **Interval** objects. The intervals must be continuous and non-overlapping.

12.2 Adding Data to Histograms

The provided `AddData()` method adds a vector of data to a **Histogram**. The histogram bin count containing each given data point is updated. For example, this code constructs a vector of `100` random numbers from a normal distribution and adds the data to **Histogram** `hist`.

Code Example – C# histogram

```
double mean = 70.0;
double variance = 10.0;
var rng = new RandGenNormal( mean, variance );
var v = new DoubleVector( 100, rng );

hist.AddData( v );
```

Code Example – VB histogram

```
Dim Mean As Double = 70.0
Dim Variance As Double = 10.0
Dim RNG As New RandGenNormal(Mean, Variance)
Dim V As New DoubleVector(100, RNG)

Hist.AddData(V)
```

As a convenience, the **Histogram** class also provides a constructor that accepts the number of bins and a vector data. The constructed bins are of equal size and scaled with the maximum and minimum data. The counts in the histogram are initialized with the contents of the given vector. Thus:

Code Example – C# histogram

```
var hist = new Histogram( 20, v );
```

Code Example – VB histogram

```
Dim Hist As New Histogram(20, V)
```

Lastly, you can add a single data point to a histogram using an overload of the `AddData()` method that accepts a double:

Code Example – C# histogram

```
double d = 5.34;
hist.AddData( d );
```

Code Example – VB histogram

```
Dim D As Double = 5.34
Hist.AddData(D)
```

12.3 Value Operations of Histograms

The **Histogram** class has the following read-only properties:

- `Bins` gets the bin boundaries as an array of **Interval** objects.
- `Counts` gets the counts for each bin as an array of integers.

- `NumBins` gets the number of bins in the histogram.
- `NumSmaller` gets the number of data points that were smaller than the smallest bin boundary.
- `NumLarger` gets the number of data points that were larger than the largest bin boundary.
- `Total` gets the total number of data points added to the histogram.

Similarly, the `Count()` member function gets the bin count for a given bin. `Reset()` resets all bin counts (and `NumSmaller` and `NumLarger`) to zero; the number of bins and the bin boundaries remain unchanged.

`PDF()` computes the probability density function (PDF) for a specified value or bin, and `CDF()` computes the cumulative distribution function (CDF).

12.4 Displaying Histograms

The **Histogram** class provides two methods for displaying a histogram textually. The `ToString()` member function returns a formatted string representation of a histogram. If the bin boundaries are `b0, b1, b2,...,bn-1`, and the counts for these bins are `c1, c2,...,cn`, respectively, then `ToString()` returns a string with the following format:

```
[b0,b1)   :   c1
[b1,b2)   :   c2
[b2,b3)   :   c3
.
.
.
[bn-2,bn-1]: cn
```

The provided `StemLeaf()` method formats the contents of a histogram into a simple ASCII stem-leaf diagram with the following form:

```
[b0,b1):      *****c1
[b1,b2):      **********c2
[b2,b3):      ***************c3
.
.
.
[bn-2,bn-1]: *****cn
```

The number of asterisks represents the count for that bin minus one.

CHAPTER 13. CALCULUS

NMath provides classes for encapsulating functions of one variable, $f(x)$. Once constructed, function objects enable you to:

- evaluate a function at a given x-value or vector of x-values;
- integrate a function over a given interval;
- compute the derivative of a function at a given x-value;
- manipulate functions algebraically.

This chapter describes how to create and manipulate function objects.

13.1 Encapsulating Functions

Class **OneVariableFunction** encapsulates an arbitrary function, and works with other numerical classes to approximate integrals and derivatives.

NOTE—Class Polynomial extends OneVariableFunction, and provides exact methods for integration and differentiation of polynomials, as well as various convenience functions for creating and manipulating polynomials. This is the preferred class to use if your function is a polynomial. See Section 13.4 for more information.

Creating a Function of One Variable

A **OneVariableFunction** is constructed from a `Func<double, double>`, a function delegate that takes a single double parameter and returns a double.

For example, suppose you wish to encapsulate this function:

Code Example – C# calculus

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

Code Example – VB calculus

```
Function MyFunction(X As Double) As Double
  Return Math.Sin(X) + Math.Pow(X, 3) / Math.PI
End Function
```

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

Code Example – C# calculus

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

Code Example – VB calculus

```
Dim D As New Func(Of Double, Double)(AddressOf MyFunction)
```

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

Code Example – C# calculus

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

Code Example – VB calculus

```
Dim F As New OneVariableFunction(D)
```

A `Func<double, double>` delegate is also implicitly converted to a **OneVariableFunction**. Thus:

Code Example – C# calculus

```
OneVariableFunction f = d;
```

Code Example – VB calculus

```
OneVariableFunction f = d;
```

Properties of Functions

A **OneVariableFunction** object has the following properties:

- `Function` gets the encapsulated function delegate.
- `Integrator` gets and sets the integration object associated with the function (see Section 13.2).
- `Differentiator` gets and sets the differentiation object associated with the function (see Section 13.3).

Evaluating Functions

The `Evaluate()` method on **OneVariableFunction** evaluates a function at a given *x*-value. For instance, if `f` is a **OneVariableFunction**:

Code Example – C# calculus

```
double y = f.Evaluate( Math.PI );
```

Code Example – VB calculus

```
Dim Y As Double = F.Evaluate(Math.PI)
```

`Evaluate()` also accepts a vector of x-values, and returns a vector of y-values, such that `y[i] = f( x[i] )`. Thus, this code evaluates `f` at `100` points between `0` and `1`:

Code Example – C# calculus

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

Code Example – VB calculus

```
Dim X As New DoubleVector(100, 0, 1.0 / 100.0)
Dim Y As DoubleVector = F.Evaluate(X)
```

Finally, `Evaluate()` accepts another **OneVariableFunction**, and returns a new function encapsulating the composite. For example, if `f` encapsulates the function $f(x) = \sin(x)$ and `g` encapsulates $g(x) = \sqrt{x+1}$, you can create a new function that encapsulates $f(g(x)) = \sin(\sqrt{x+1})$ like so:

Code Example – C# calculus

```
OneVariableFunction composite = f.Evaluate( g );
```

Code Example – VB calculus

```
Dim Composite As OneVariableFunction = F.Evaluate(g)
```

Algebraic Manipulation of Functions

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. Table 10 lists the equivalent operators and methods.

Table 10 – Arithmetic operators

Operator	Equivalent Named Method
`+`	`Add()`
`-`	`Subtract()`
`*`	`Multiply()`

Table 10 – Arithmetic operators

Operator	Equivalent Named Method
`/`	`Divide()`
Unary `-`	`Negate()`

All binary operators and equivalent named methods work either with two functions, or with a function and a scalar. For example, this C# code uses the overloaded operators:

Code Example – C# calculus

```
OneVariableFunction g = f/2;
OneVariableFunction sum = f + g;
OneVariableFunction neg = -f;
```

This Visual Basic code uses the equivalent named methods:

Code Example – VB calculus

```
Dim G As OneVariableFunction = OneVariableFunction.Divide(F, 2)
Dim Sum As OneVariableFunction = OneVariableFunction.Add(F, g)
Dim Neg As OneVariableFunction = OneVariableFunction.Negate(F)
```

Finally, as a convenience, **NMathFunctions** provides a `Pow()` method that raises a function to a scalar power:

Code Example – C# calculus

```
OneVariableFunction g = NMathFunctions.Pow( f, 3.5 );
```

Code Example – VB calculus

```
Dim G As OneVariableFunction = NMathFunctions.Pow(F, 3.5)
```

13.2 Numerical Integration

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

NOTE—Class Polynomial provides a method for constructing the exact antiderivative of a polynomial. See Section 13.4 for more information.

Computing Integrals

The `Integrate()` method on **OneVariableFunction** (Section 13.1) computes the integral of a function over a given interval. For example, if `f` is a **OneVariableFunction**, this code integrates `f` over the interval `-1` to `1`:

Code Example – C# calculus

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

Code Example – VB calculus

```
Dim Integral As Double = F.Integrate(-1, 1)
```

NOTE—NMath does not directly support improper intervals; that is, it must be possible to evaluate the function at both the lower and upper bounds, and at any point in between (no singularities).

To perform integration, every **OneVariableFunction** has an **IIntegrator** object associated with it. **NMath** integration classes such as **RombergIntegrator** and **GaussKronrodIntegrator** implement the **IIntegrator** interface. The default integrator for a **OneVariableFunction** is an instance of **RombergIntegrator**, which may be changed using the `Integrator` property. Thus:

Code Example – C# calculus

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

Code Example – VB calculus

```
F.Integrator = New GaussKronrodIntegrator()
Dim Integral As Double = F.Integrate(0, Math.PI)
```

You can also change the default **IIntegrator** associated with all instances of **OneVariableFunction** using the static `DefaultIntegrator` property. For instance:

Code Example – C# calculus

```
OneVariableFunction.DefaultIntegrator =
  new GaussKronrodIntegrator();

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

double integral = f.Integrate( 0, 1 );  // uses Gauss-Kronrod
```

Code Example – VB calculus

```
OneVariableFunction.DefaultIntegrator = New
GaussKronrodIntegrator()
```

```
Dim D As New Func(Of Double, Double)(AddressOf MyFunction)
Dim F As New OneVariableFunction(D)

Dim Integral As Double = f.Integrate(0, 1) ' uses Gauss-Kronrod
```

Romberg Integration

In general, the class of methods known as *Newton-Cotes formulas* estimate the integral of a function over a given interval by dividing the interval into 2^k panels, where *k* is called the *order*, estimating the integral within each panel, then summing the estimates. For instance, the *trapezoidal rule* approximates the function in each panel by a straight line between the end points. *Simpson's rule* approximates the function in two adjacent panels by a quadratic function connecting the two outer points and the common midpoint. Higher-level methods are obtained by interpolating higher degree polynomial segments.

Because all methods evaluate the function at the same set of points, higher-level approximations can be derived from lower-level approximations. For example, it can be shown that a *k*th-order Simpson's rule approximation can be derived from two trapezoidal rule approximations of order *k* and *k*-1. Similarly, a *Boole's rule* approximation, which fits third-degree polynomials through the points associated with four-panel partitions of the interval, can be derived from two Simpson's rule approximations of order *k* and *k*-1. In this way, all higher level approximations can be derived from a series of trapezoidal rule approximations.

This iterated application of trapezoidal rule approximations is known as *Romberg integration*. Romberg integration is a very powerful method for quickly and accurately integrating smooth functions.

In **NMath**, instances of class **RombergIntegrator** compute successive Romberg approximations of increasing order until the estimated error in the approximation is less than a specified error tolerance, or until the maximum order is reached. The default error tolerance is `1e-8`, and the default maximum order is `20`.

To perform integration, every **OneVariableFunction** has an **IIntegrator** object associated with it, which is used by the `Integrate()` method to compute integrals. The default **IIntegrator** for a **OneVariableFunction** is an instance of **RombergIntegrator**. For example, assuming `f` is a **OneVariableFunction**, this code uses the default **RombergIntegrator** to integrate over the interval `-1` to `1`:

Code Example – C# calculus

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

Code Example – VB calculus

```
Dim Estimate As Double = f.Integrate(-1, 1)
```

The underlying **IIntegrator** can be accessed using the `Integrator` property.

In some cases, you may wish to create a **RombergIntegrator** yourself. This gives you more control over the integration process, and allows you to reuse a customized integrator to integrate several functions, or one function over several intervals. Thus, this code instantiates a **RombergIntegrator**, uses the provided `Tolerance` property to change the error tolerance and the `MaximumOrder` property to change the maximum order, then calls the `Integrate()` method on **RombergIntegrator** to integrate functions `f` and `g`:

Code Example – C# calculus

```
var rom = new RombergIntegrator();
rom.Tolerance = 1e-6;
rom.MaximumOrder = 16;
double integralF = rom.Integrate( f, -1, 1);
double integralG = rom.Integrate( g, 0, 2 * Math.PI );
```

Code Example – VB calculus

```
Dim Rom As New RombergIntegrator()
Rom.Tolerance = "1e-6"
Rom.MaximumOrder = 16
Dim IntegralF As Double = Rom.Integrate(f, -1, 1)
Dim IntegralG As Double = Rom.Integrate(g, 0, 2 * Math.PI)
```

To compute a Romberg estimate of a specific order, *k*, you can also set the `MaximumOrder` to `k` and the `Tolerance` to a negative value. This code configures the **RombergIntegrator** to compute an 8*th*-order approximation:

Code Example – C# calculus

```
var rom = new RombergIntegrator();
rom.Tolerance = -1;
rom.MaximumOrder = 8;
double estimate = rom.Integrate( f, -1, 1);
```

Code Example – VB calculus

```
Dim Rom = New RombergIntegrator()
Rom.Tolerance = -1
Rom.MaximumOrder = 8
Dim Estimate As Double = Rom.Integrate(f, -1, 1)
```

After computing an estimate, a **RombergIntegrator** holds a record of the iteration process. Read-only properties are provided for accessing this information:

- `RombergEstimate` gets the Romberg estimate for the integral, as returned by the `Integrate()` method.
- `RombergErrorEstimate` gets an estimate of the error in the Romberg estimate of the integral just computed.
- `ToleranceMet` returns `true` if the estimate of the error in the Romberg approximation just computed is less than or equal to the tolerance;

otherwise, `false`. (Integration ends either when the estimated error in the approximation is less than tolerance, or when the maximum order is reached.)

- `Order` gets the order of the Romberg approximation just computed.
- `TrapeziodEstimate` gets the estimate for the integral yielded by the compound trapeziod rule where the number of panels is equal to the order of the Romberg estimate.
- `SimpsonEstimate` gets the estimate for the integral yielded by the compound Simpson's rule where the number of panels is equal to the order of the Romberg estimate. (Note: Returns `0` if `Order = 0`.)
- `Tableau` gets the entire **DoubleMatrix** of successive approximations computed while computing a Romberg estimate. The rows are the order of approximation. The columns are the level of approximation. The first column contains the trapezoidal approximations, the second column the Simpson's rule approximations, the third column the Boole's rule approximations, and so on, up to the `Order` of the approximation just computed.

Thus, this code retrieves the Boole's rule approximation:

Code Example – C# calculus

```
var rom = new RombergIntegrator();
double integral = rom.Integrate( f, 0, 1);
int order = rom.Order;
DoubleMatrix tableau = rom.Tableau;

double boole;
if ( order >= 2 )
{
  boole = tableau[ order, 2 ];
}
```

Code Example – VB calculus

```
Dim Rom As New RombergIntegrator()
Dim Integral As Double = Rom.Integrate(f, 0, 1)
Dim Order As Integer = Rom.Order
Dim Tableau As DoubleMatrix = Rom.Tableau

Dim Boole As Double
If Order >= 2 Then
  Boole = Tableau(Order, 2)
End If
```

Gauss-Kronrod Integration

Gaussian integration estimates an integral by evaluating the function at *non-equally spaced* points over the interval. The method attempts to pick optimal points at which to evaluate the function, and furthermore to weight the contribution of each point. *Gauss-Kronrod* integration is an adaptive Gaussian quadrature method in which the function is evaluated at special points known as *Kronrod points*. The Gauss-Kronrod method is especially suited for non-singular oscillating integrands.

NMath includes Gauss-Kronrod classes for different numbers of Kronrod points ($2n + 1$, beginning with a Gauss 10-point rule):

- **GaussKronrod21Integrator** approximates integrals using the Gauss 10-point and the Kronrod 21-point rule.
- **GaussKronrod43Integrator** approximates integrals using the Gauss 21-point and the Kronrod 43-point rule.
- **GaussKronrod87Integrator** approximates integrals using the Gauss 43-point and the Kronrod 87-point rule.

Finally, the automatic **GaussKronrodIntegrator** class uses Gauss-Kronrod rules with increasing number of points. Approximation ends when the relative error is less than the tolerance scaled by the integration result, or when the maximum number of points is reached. The default error tolerance is `1e-7`; the default maximum number of points is `87`. Unless you have reason to believe in advance that a particular Gauss-Kronrod rule is optimal for your function, it is recommended that you use the automatic integrator.

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

Code Example – C# calculus

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

Code Example – VB calculus

```
F.Integrator = New GaussKronrodIntegrator()
Dim Integral As Double = F.Integrate(0, Math.PI)
```

This code specifically uses the **GaussKronrod43Integrator**, rather than the automatic **GaussKronrodIntegrator**:

Code Example – C# calculus

```
f.Integrator = new GaussKronrod43Integrator();
double integral = f.Integrate( -1, 1 );
```

Code Example – VB calculus

```
F.Integrator = New GaussKronrod43Integrator()
```

```
Dim Integral As Double = F.Integrate(-1, 1)
```

In some cases you may wish to create a Gauss-Kronrod integrator yourself. This gives you more control over the integration process, and allows you to reuse a customized integrator to integrate several functions, or one function over several intervals. Thus, this code instantiates a **GaussKronrodIntegrator**, uses the provided `Tolerance` property to change the error tolerance, then calls the `Integrate()` method on **GaussKronrodIntegrator** to integrate functions `f` and `g`:

Code Example – C# calculus

```
var gk = new GaussKronrodIntegrator();
gk.Tolerance = 1e-6;
double integralF = gk.Integrate( f, -1, 1);
double integralG = gk.Integrate( g, 0, 2 * Math.PI );
```

Code Example – VB calculus

```
Dim GK As New GaussKronrodIntegrator()
GK.Tolerance = "1e-6"
Dim IntegralF As Double = GK.Integrate(f, -1, 1)
Dim IntegralG As Double = GK.Integrate(g, 0, 2 * Math.PI)
```

Read-only properties are provided for accessing information about an integral approximation, once it has been computed:

- `RelativeErrorEstimate` gets an estimate of the relative error for the integral approximation.
- `ToleranceMet` gets a boolean value indicating whether or not the relative error for the integral approximation is less than the tolerance scaled by the integration result.
- `PreviousEstimate` gets the integral approximation calculated using the previous rule—for example, the Gauss 10-point rule for a **GaussKronrod21Integrator**, the Kronrod 21-point rule for a **GaussKronrod43Integrator**, and so forth.

For instance, this code checks whether the error tolerance was met before proceeding:

Code Example – C# calculus

```
var gk = new GaussKronrodIntegrator();
gk.Tolerance = 1e-6;
double integral = gk.Integrate( f, -1, 1 );

if ( gk.ToleranceMet )
{
  // Do something here...
}
```

Code Example – VB calculus

```
Dim GK As New GaussKronrodIntegrator()
GK.Tolerance = "1e-6"
Dim Integral As Double = GK.Integrate(f, -1, 1)

If GK.ToleranceMet Then
  ' Do something here...
End If
```

13.3 Differentiation

The `Differentiate()` method on **OneVariableFunction** (Section 13.1) computes the derivative of a function at a given *x*-value. For example, if `f` is **OneVariableFunction**, this code estimates the derivative at `0`:

Code Example – C# calculus

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

Code Example – VB calculus

```
Dim D As Double = F.Differentiate(0)
```

NOTE—Class Polynomial provides a method for constructing the exact derivative of a polynomial. See Section 13.4 for more information.

To perform differentiation, every **OneVariableFunction** has an **IDifferentiator** object associated with it. **NMath** provides class **RiddersDifferentiator**, which computes the derivative of a given function at a given *x*-value by Ridders' method of polynomial extrapolation, and implements the **IDifferentiator** interface.

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.

The default **IDifferentiator** for a **OneVariableFunction** is an instance of **RiddersDifferentiator**. To achieve more control over how differentiation is performed, you can instantiate your own **RiddersDifferentiator**. For instance, this code uses the `Tolerance` property to set the error tolerance to a non-default value, and the `MaximumOrder` property to set the maximum order, then calls the `Differentiate()` method to differentiate function `f` at π:

Code Example – C# calculus

```
var ridders = new RiddersDifferentiator();
ridders.Tolerance = 1e-6;
ridders.MaximumOrder = 20;
double d = ridders.Differentiate( f, Math.PI );
```

Code Example – VB calculus

```
Dim Ridders As New RiddersDifferentiator()
Ridders.Tolerance = "1e-6"
Ridders.MaximumOrder = 20
Dim D As Double = Ridders.Differentiate(F, Math.PI)
```

Setting the error tolerance to a value less than zero ensures that the Ridders differentiation is of the maximum order:

Code Example – C# calculus

```
var ridders = new RiddersDifferentiator();
ridders.Tolerance = -1;
double d = ridders.Differentiate( f, 1 );
```

Code Example – VB calculus

```
Dim Ridders As New RiddersDifferentiator()
Ridders.Tolerance = -1
Dim D As Double = Ridders.Differentiate(F, 1)
```

Read-only properties are provided for accessing information about a derivative approximation, once it has been computed:

- `ErrorEstimate` gets an estimate of the error of the derivative just computed.
- `Order` gets the order of the final polynomial extrapolation.
- `ToleranceMet` gets a boolean value indicating whether or not the error estimate for the derivative approximation is less than the tolerance.
- `Tableau` gets a matrix of successive approximations produced while computing the derivative. Successive columns in the matrix contain higher orders of extrapolation; successive rows decreasing step size.

For instance, this code checks whether the error tolerance was met before proceeding:

Code Example – C# calculus

```
var ridders = new RiddersDifferentiator();
double d = ridders.Differentiate( f, Math.PI );

if ( ridders.ToleranceMet ) {
  // Do something here...
}
```

Code Example – VB calculus

```
Dim Ridders As New RiddersDifferentiator()
Dim D As Double = ridders.Differentiate(F, Math.PI)
```

```
If Ridders.ToleranceMet Then
  ' Do something here...
End If
```

13.4 Polynomials

Class **Polynomial** extends **OneVariableFunction** (Section 13.1). Rather than encapsulating an arbitrary function delegate, **Polynomial** represents a polynomial by its coefficients, arranged in ascending order—that is, a vector $a_0, a_1, \ldots, a_n$ such that:

$$f(x) = a_0x^0 + a_1x^1 + \ldots + a_nx^n$$

Thus, the polynomial $5x^4 - 2x^2 + x + 3$ is represented as a **DoubleVector** of length `5` with elements `"3 1 -2 0 5"`.

Creating Polynomials

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:

Code Example – C# polynomials

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

Code Example – VB polynomials

```
Dim Coef As New DoubleVector("1 0 2")     ' 2x^2 + 1
Dim P As 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. For example, this code interpolates the polynomial $2x^2 - x + 5$ through the points `(1,6)`, `(2,11)`, and `(3,20)`:

Code Example – C# polynomials

```
var x = new DoubleVector( "1 2 3");
var y = new DoubleVector( "6 11 20" );
var p = new Polynomial( x, y );
```

Code Example – VB polynomials

```
Dim X As New DoubleVector("1 2 3")
Dim Y As New DoubleVector("6 11 20")
Dim P As New Polynomial(X, Y)
```

You can also construct a **Polynomial** instance from a vector of *x*-values and a **OneVariableFunction** evaluated at each *x*:

Code Example – C# polynomials

```
var f = new Func<double, double>( myFunction );
var x = new DoubleVector( 10, 1, 1 );
var p = new Polynomial( x, f );
```

Code Example – VB polynomials

```
Dim F As New Func(Of Double, Double)(AddressOf myFunction)
Dim X As New DoubleVector(10, 1, 1)
Dim P As New Polynomial(X, F)
```

Properties of Polynomials

Class **Polynomial** inherits `Function`, `Integrator`, and `Differentiator` properties from **OneVariableFunction** (Section 13.1). Additionally, **Polynomial** provides these properties:

- `Coeff` gets and sets the vector of coefficients.
- `Degree` gets the degree of the polynomial.

The degree is the order of the highest non-zero coefficient. Therefore, the degree may be less than the length of the underlying coefficient vector, as returned for example by `Coeff.Length`. The `Reduce()` method is provided for removing trailing zeros from the coefficient vector.

Evaluating Polynomials

Class **Polynomial** inherits the `Evaluate()` method from **OneVariableFunction**. This method evaluates a polynomial at a given *x*-value, or vector of *x*-values. Thus:

Code Example – C# polynomials

```
var coeff = new DoubleVector( "6 -1 5 0 3 -2" );
var p = new Polynomial( coeff );

double y = p.Evaluate( 1.25 );
```

Code Example – VB polynomials

```
Dim Coeff As New DoubleVector("6 -1 5 0 3 -2")
Dim P As New Polynomial(Coeff)

Dim Y As Double = P.Evaluate(1.25)
```

Algebraic Manipulation of Polynomials

Because a **Polynomial** *is-a* **OneVariableFunction**, all of the overloaded arithmetic operators and equivalent named methods described in Section 13.1 accept polynomials. For example, this code adds a **Polymomial** to a **OneVariableFunction** to create a new **OneVariableFunction**:

Code Example – C# polynomials

```
var coeff = new DoubleVector( "1 4 -1 1 2 -3" );
var p = new Polynomial( coeff );

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

OneVariableFunction g = p + f;
```

Code Example – VB polynomials

```
Dim Coeff As New DoubleVector("1 4 -1 1 2 -3")
Dim P As New Polynomial(Coeff)

Dim D As New Func(Of Double, Double)(AddressOf MyFunction)
Dim F As New OneVariableFunction(D)

Dim G As Func(Of Double, Double) = P + F
```

Additionally, class **Polynomial** provides overloads of the arithmetic operators and named methods. These operators and methods work either with two polynomials, or with a polynomial and a scalar. They operate directly on the underlying vector(s) of coefficients, and therefore return instances of **Polynomial**. For example:

Code Example – C# polynomials

```
var coeff = new DoubleVector( "-11 3 1 1 0 -1 2" );
var p = new Polynomial( coeff );
Polynomial p2 = p/2;
Polynomial p3 = p + p2;
```

Code Example – VB polynomials

```
Dim Coeff As New DoubleVector("-11 3 1 1 0 -1 2")
Dim P As New Polynomial(Coeff)
Dim P2 As Polynomial = P / 2.0
Dim P3 As Polynomial = P + P2
```

NOTE—You can divide one Polynomial by another, but the result is a OneVariableFunction rather than a Polynomial, since the quotient is a rational function, and not necessarily a polynomial.

Integration

Class **Polynomial** inherits the `Integrate()` method from **OneVariableFunction** (Section 13.2), 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:

Code Example – C# polynomials

```
var p = new Polynomial( new DoubleVector( "5 3 0 2" ) );
double integral = p.Integrate( -1, 1 );
Polynomial i = p.AntiDerivative();
```

Code Example – VB polynomials

```
Dim P As New Polynomial(New DoubleVector("5 3 0 2"))
Dim Integral As Double = P.Integrate(-1, 1)
Dim I As Polynomial = P.AntiDerivative()
```

In constructing the antiderivative, the constant of integration is assumed to be zero.

Each **Polynomial** object has a **PolynomialIntegrator** associated with it, which implements the **IIntegrator** interface. 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.

NOTE—You can, of course, set the IIntegrator associated with a Polynomial to a non-default value, such as a Romberg numerical integrator. But since this would only compute an approximation of the integral, there would be little point.

Differentiation

Class **Polynomial** inherits both `Differentiate()` and `Derivative()` methods from **OneVariableFunction** (Section 13.3). `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:

Code Example – C# polynomials

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

Code Example – VB polynomials

```
Dim Coeff As New DoubleVector("1 -2 3")
```

```
Dim P As New Polynomial(Doeff)
Dim Der As Polynomial = P.Derivative()  ' Der.Coeff = "-2 6"
```

Each **Polynomial** object has a **PolynomialDifferentiator** associated with it, which implements the **IDifferentiator** interface. Because the derivative of a polynomial can be easily constructed, **PolynomialDifferentiator** simply constructs the first derivative and evaluates it at the given *x*-value. This gives the exact derivative.

NOTE—You can, of course, set the Differentiator associated with a polynomial to a non-default value, such as a Ridders numerical differentiator. But again, as this would only compute an approximation of the derivative, there would be little point.

13.5 Function Interpolation

Abstract class **TabulatedFunction** extends **OneVariableFunction** (Section 13.1). Rather than encapsulating an arbitrary function delegate, **TabulatedFunction** holds paired vectors of known *x*- and *y*-values. The function can be evaluated at arbitrary points using derived *function interpolation classes*. As a **OneVariableFunction**, a **TabulatedFunction** can be manipulated algebraically. Numerical integrals and derivatives can also be computed.

A **TabulatedFunction** type is constructed from paired vectors of known *x*- and *y*-values. The values for *x* must be in strictly increasing order. Class **TabulatedFunction** inherits `Function`, `Integrator`, and `Differentiator` properties from **OneVariableFunction** (Section 13.1). Additionally, **TabulatedFunction** provides these properties:

- `X` gets the vector of x-values represented by the function.
- `Y` gets the vector of y-values represented by the function.
- `NumberOfTabulatedValues` gets the number of tabulated values.

The `X` and `Y` properties return a copy of the tabulated data. Therefore, modifying the returned vectors does not change the **TabulatedFunction**.

To change the tabulated values represented by a **TabulatedFunction**, use the `SetTabulatedValues()` method. Provided `GetX()`, `SetX()`, `GetY()`, and `SetY()` methods also enable you to get and set individual tabulated values, or a range of values.

Class **TabulatedFunction** inherits the `Evaluate()` method from **OneVariableFunction**. This method evaluates the interpolated function at a given *x*-value, or vector of *x*-values.

Linear Spline Interpolation

Class **LinearSpline** extends **TabulatedFunction** and represents a function whose values are determined by linear interpolation between tabulated values. For example:

Code Example – C# linear spline interpolation

```
var xValues = new DoubleVector(10, 0, 1);
DoubleVector yValues = xValues * xValues;
var ls = new LinearSpline( xValues, yValues );
double yInterpolated = ls.Evaluate( 3.5 );
```

Code Example – VB linear spline interpolation

```
Dim XValues As New DoubleVector(10, 0, 1)
Dim YValues = XValues * XValues
Dim LS As New LinearSpline(XValues, YValues)
Dim YInterpolated = LS.Evaluate(3.5)
```

Evaluating *x*-values outside the range of tabulated values returns the last know *y*-value. In the example above, `ls.Evaluate( 9.5 ) == ls.Evaluate( 9 )`.

Cubic Spline Interpolation

Abstract class **CubicSpline** extends **TabulatedFunction** and represents a function whose values are determined by cubic spline interpolation between the tabulated values. **NMath** provides two concrete implementations of **CubicSpline**: **NaturalCubicSpline** and **ClampedCubicSpline**. The *natural cubic spline* is a cubic spline where the second derivative of the interpolating function is required to be zero at the left and right endpoints. The *clamped cubic spline* is a cubic spline where the first derivative of the interpolating function is specified at the left and right endpoints.

For example, this code creates a **NaturalCubicSpline** to resample at a fixed sampling interval a cubic spline fit constructed from data with a variable sampling interval:

Code Example – C# cubic spline interpolation

```
var x = new DoubleVector( “1.0 1.3 1.4 1.8 2.0”);
var y = new DoubleVector( “2.4 4.6 4.7 2.3 1.0” );

var s = new NaturalCubicSpline( x, y );

var xx = new DoubleVector( “1.0 1.25 1.5 1.75 2.0”);
DoubleVector yy = s.Evaluate( xx );
```

Code Example – VB cubic spline interpolation

```
Dim X As New DoubleVector("1.0 1.3 1.4 1.8 2.0")
```

```
Dim Y As New DoubleVector("2.4 4.6 4.7 2.3 1.0")

Dim S As New NaturalCubicSpline(X, Y)

Dim XX As New DoubleVector("1.0 1.25 1.5 1.75 2.0")
Dim YY As DoubleVector = S.Evaluate(XX)
```

This code creates a **ClampedCubicSpline** that enforces endslopes of zero for the cubic spline fit:

Code Example – C# cubic spline interpolation

```
var s = new ClampedCubicSpline( x, y, 0, 0 );
```

Code Example – VB cubic spline interpolation

```
Dim S As New ClampedCubicSpline(X, Y, 0, 0)
```

Class **ClampedCubicSpline** provides `LeftEndSlope` and `RightEndSlope` properties for getting and setting the clamped values, and method `SetEndSlopes()` for modifying them together.

Evaluating *x*-values outside the range of tabulated values in a **NaturalCubicSpline** returns the last know *y*-value. In a **ClampedCubicSpline**, the last fitted cubic is used, or a linear extrapolation is performed in the case of only `2` or `3` tabulated vaules.

Smooth Splines

Class **SmoothCubicSpline** derives from **TabulatedFunction**. The API is the same as for other cubic spline classes, with the addition of a smoothing factor, `P`. The smoothing factor takes values in the range `0 <= p <= 1`, where `0` results in zero curvature (linear interpolation), and `1` results to a conventional cubic spline.

Creating Your Own Interpolation Classes

The **NMath** interpolation class framework is easily extensible. To create your own interpolation class, simply extend **TabulatedFunction**. Specify a delegate function of type `Func<double, double>` for the instance variable `function` in the base class **OneVariableFunction**. This delegate computes and returns values for arbitrary *x*-values.

In addition, deriving classes may override the virtual method `ProcessTabulatedValues()`. This method is invoked by **TabulatedFunction** instances whenever the tabulated values are changed.

CHAPTER 14.
SIGNAL PROCESSING

NMath provides classes for processing 1D signal data, including

- filtering, using **MovingWindowFilter** or **SavitzkyGolayFilter**;
- peak finding, using **PeakFinderSavitzkyGolay** or **PeakFinderRuleBased**.

This chapter describes how to create and manipulate signal processing objects.

14.1 Moving Window Filtering

Class **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. The neighboring points are determined by the number of points to the left, *nL*, and the number of points to the right, *nR*:

$$g(i) = \sum_{n=-nL}^{nR} c(n)f(i+n)$$

MovingWindowFilter extends class **CorrelationFilter** which provides basic correlation services.

Creating Moving Window Filter Objects

A **MovingWindowFilter** instance is constructed from the number of points to the left and right of the input point, and the coefficients of the linear combination.

For example, this code constructs an asymmetric moving window filter of length `5`:

Code Example – C# signal filtering

```
int numberLeft = 1;
int numberRight = 3;
var filterCoefficients = new DoubleVector(5, 0.20);
var filter = new MovingWindowFilter( numberLeft, numberRight,
  filterCoefficients );
```

Code Example – VB signal filtering

```
Dim NumberLeft = 1
```

```
Dim NumberRight = 3
Dim FilterCoefficients As New DoubleVector(5, 0.2)
Dim Filter As New MovingWindowFilter(NumberLeft, NumberRight,
  FilterCoefficients)
```

An **InvalidArgumentException** is raised if the length of the coefficient vector is not equal to `numberLeft + numberRight + 1`.

Static class methods are provided for generating coefficient vectors of three common types:

- `MovingAverageCoefficients()` constructs a coefficient vector that implements a moving average filter.
- `ExponentiallyWeightedMovingAverageCoefficients()` constructs a coefficient vector of exponentially weighted moving average (EWMA) coefficients of the specified length. As the number of EWMA coefficients increases, the filter captures at most `%86.47` of the total weight due to the finite length of the filter. The filter length n and the exponential weight α are related by $\alpha = 2/(n+1)$.
- `SavitzkyGolayCoefficients()` constructs a coefficient vector that implements a Savitzky-Golay smoothing filter (also known as least-squares, or DIgital Smoothing POlynomial, DISPO). The filter coefficients are chosen such that the filtered point is the value of an approximating polynomial of the specified order, typically quadratic or quartic. The polynomial is fit using a least squares algorithm.

For example, the following code constructs a moving average filter to replace each input data point with the average of it's value and the surrounding points:

Code Example – C# signal filtering

```
int numberLeft = 4;
int numberRight = 5;
DoubleVector filterCoefficients =
  MovingWindowFilter.MovingAverageCoefficients( numberLeft,
    numberRight );
var filter = new MovingWindowFilter( numberLeft, numberRight,
  filterCoefficients );
```

Code Example – VB signal filtering

```
Dim NumberLeft = 4
Dim NumberRight = 5
Dim FilterCoefficients As DoubleVector =
  MovingWindowFilter.MovingAverageCoefficients(NumberLeft,
    NumberRight)
Dim Filter As New MovingWindowFilter(NumberLeft, NumberRight,
  FilterCoefficients)
```

This code creates a Savitzky-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:

Code Example – C# signal filtering

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

Code Example – VB signal filtering

```
Dim NumberLeft = 3
Dim NumberRight = 3
Dim Degree = 4
Dim FilterCoefficients As DoubleVector =
  MovingWindowFilter.SavitzkyGolayCoefficients(NumberLeft,
    NumberRight, Degree)
Dim Filter As New MovingWindowFilter(NumberLeft, NumberRight,
  FilterCoefficients)
```

This code creates an exponential moving average filter of length `18`:

Code Example – C# signal filtering

```
int n = 18;
DoubleVector filterCoefficients =
 MovingWindowFilter.ExponentiallyWeightedMovingAverageCoefficients(
n );

var EWMAfilter = new MovingWindowFilter( 0,
  coef.filterCoefficients - 1, filterCoefficients );
```

Code Example – VB signal filtering

```
Dim N = 18
Dim FilterCoefficients As DoubleVector =

 MovingWindowFilter.ExponentiallyWeightedMovingAverageCoefficients(
  N)

Dim EWMAfilter As New MovingWindowFilter(0,
  Coef.FilterCoefficients - 1, filterCoefficients)
```

After construction, the `SetFilterParameters()` method can be used to reset the filter parameters on a filter instance:

Code Example – C# signal filtering

```
filter.SetFilterParameters( numberLeft, numberRight,
  filterCoefficients );
```

Code Example – VB signal filtering

```
Filter.SetFilterParameters(NumberLeft, NumberRight,
  FilterCoefficients)
```

Moving Window Filter Properties

Once constructed, a **MovingWindowFilter** object provides the following read-only properties:

- `NumberLeft` gets the number of points to the left for the filter window.
- `NumberRight` gets the number of points to the right for the filter window.
- `WindowWidth` gets the width of the moving window (equal to `NumberLeft + NumberRight + 1`).
- `NumberOfCoefficients` gets the number of filter coefficients (equal to `WindowWidth`).
- `Coefficients` gets the vector of filter coefficients.

Filtering Data

The `Filter()` method on **MovingWindowFilter** applies a filter to a given data set using the specified boundary option.

The `MovingWindowFilter.BoundaryOption` enumeration specifies options for handling the boundaries in a moving window filter, where the filter does not complete overlap with the data:

- `BoundaryOption.PadWithZeros` adds `NumberLeft` zeros to the beginning of the data to be filtered and `NumberRight` zeros to end.
- `BoundaryOption.DoNotFilterBoundaryPoints` specifies that the first `NumberLeft` and the last `NumberRight` data will not be filtered.

For example, the following code constructs a noisy cosine signal, and then filters the data:

Code Example – C# signal filtering

```
var rng = new RandGenNormal();
var noisySignal = new DoubleVector( length );
for ( int i = 0; i < length; i++ )
```

```
{
  noisySignal[i] = Math.Cos( .2*i ) + rng.Next();
}

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

Code Example – VB signal filtering

```
Dim RNG As New RandGenNormal()
Dim NoisySignal As New DoubleVector(Length)
For I As Integer = 0 To Length - 1
  NoisySignal(I) = Math.Cos(0.2 * I) + RNG.Next()
Next

Dim FilteredSignal As DoubleVector = Filter.Filter(NoisySignal,
  MovingWindowFilter.BoundaryOption.PadWithZeros)
```

14.2 Savitzky-Golay Filtering

Class **SavitzkyGolayFilter** is a correlation filter specialized for filtering with Savitzky-Golay coefficients. Unlike **MovingWindowFilter** (Section 14.1), **SavitzkyGolayFilter** has additional boundary options for better edge continuity.

SavitzkyGolayFilter uses class **SavitzkyGolay** to generate the Savitzky-Golay filter coefficients for smoothing data, or computing smoothed derivatives, and extends class **CorrelationFilter** which provides basic correlation services.

Creating Savitzky-Golay Filter Objects

A **SavitzkyGolayFilter** instance is constructed from the number of points to the left and right of the input point, and the degree of polynomial used to fit data. Either the data or a derivative of the data can be smoothed.

For example, this code builds a Savitzky-Golay filter with a window width of `7`, and a `4`th degree smoothing polynomial:

Code Example – C# Savitzky-Golay

```
int numberLeft = 3;
int numberRight = 3;
int degree = 4;
SavitzkyGolayFilter sgf =
  new SavitzkyGolayFilter(numberLeft, numberRight, degree);
```

Code Example – VB Savitzky-Golay

```
Dim NumberLeft = 3
```

```
Dim NumberRight = 3
Dim Degree = 4
Dim SGF As New SavitzkyGolayFilter(NumberLeft, NumberRight, Degree)
```

This code creates a Savitzky-Golay filter for smoothing the first derivative using a 5th degree polynomial:

Code Example – C# Savitsky-Golay

```
int numberLeft = 3;
int numberRight = 3;
int degree = 5;
int derivativeOrder = 1;

var sgf = new SavitzkyGolayFilter(numberLeft,
  numberRight, degree, derivativeOrder);
```

Code Example – VB Savitzky-Golay

```
Dim NumberLeft = 3
Dim NumberRight = 3
Dim Degree = 5
Dim DerivativeOrder = 1

Dim SGF As New SavitzkyGolayFilter(NumberLeft, NumberRight, Degree,
  DerivativeOrder)
```

Savitzky-Golay Filter Properties

Once constructed, a **SavitzkyGolayFilter** object provides the following read-only properties:

- `NumberLeft` gets the number of points to the left for the filter window.
- `NumberRight` gets the number of points to the right for the filter window.
- `WindowWidth` gets the width of the moving window (equal to `NumberLeft + NumberRight + 1`).

Filtering Data

The `Filter()` method on **SavitzkyGolayFilter** applies a filter to a given data set:

Code Example – C# Savitsky-Golay

```
DoubleVector filteredSignal = filter.Filter( noisySignal );
```

Code Example – VB Savitzky-Golay

```
Dim FilteredSignal As DoubleVector = Filter.Filter(NoisySignal)
```

A boundary option may also be specified using the `SavitzkyGolayFilter.SavitzyGolayBoundaryOption` enumeration, which provides options for handling the boundaries in a Savitzky-Golay filter, where the filter does not completely overlap with the data:

- `SavitzyGolayBoundaryOption.PadWithZeros` adds `NumberLeft` zeros to the beginning of the data to be filtered and `NumberRight` zeros to end.
- `SavitzyGolayBoundaryOption.DoNotFilterBoundaryPoints` specifies that the first `NumberLeft` and the last `NumberRight` data will not be filtered.
- `SavitzyGolayBoundaryOption.ShiftFilterCenter` (the default) uses the Savitzky-Golay smoothing of the same order of the filter to smooth the boundaries. The filter width, and polynomial order is kept fixed, while the filter centerpoint is shifted toward the boundaries.
- `SavitzyGolayBoundaryOption.ShrinkFilterWidth` uses the Savitzky-Golay smoothing of the same order of the filter to smooth the ends points. The polynomial order is kept fix, and the filter width is shrunk as the filter center approaches the data bounday.

For instance:

Code Example – C# Savitsky-Golay

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

Code Example – VB Savitzky-Golay

```
Dim FilteredSignal As DoubleVector = Filter.Filter(NoisySignal,
  SavitzyGolayBoundaryOption.PadWithZeros)
```

14.3 Savitzky-Golay Peak Finding

Class **PeakFinderSavitzkyGolay** uses smooth Savitzky-Golay derivatives to find peaks in data. A *peak* is defined as a smoothed derivative zero crossing.

PeakFinderSavitzkyGolay extends **PeakFinderBase**, the abstract base class for all peak finding algorithms, and an enumerable collection of all found peaks.

Creating Savitzky-Golay Peak Finders

A **PeakFinderSavitzkyGolay** instance is constructed from a vector of data, a window width, and the degree of polynomial used to fit the data. For instance, this code builds a data set from a `sinc()` function, then constructs a peak finder with a width of `6`, and `4`th degree smoothing polynomial:

Code Example – C# peak finding

```
var x = new DoubleVector(5000, 0.01, 0.1);
DoubleVector data = NMathFunctions.Sin(x) / x;
PeakFinderSavitzkyGolay pf =
  new PeakFinderSavitzkyGolay(data, 6, 4);
```

Code Example – VB peak finding

```
Dim X As New DoubleVector(5000, 0.01, 0.1)
Dim Data As DoubleVector = NMathFunctions.Sin(X) / X
Dim PF As New PeakFinderSavitzkyGolay(Data, 6, 4)
```

The constructor parameters must satisfy the following rules:

- The window width must be less than the length of the data.
- The polynomial degree must be less than the window width.

Typically, the degree of the smoothing polynomial is between `3` and `5`.

Savitzky-Golay Peak Finder Results

Once you've constructed a **PeakFinderSavitzkyGolay** object, the `LocatePeaks()` method finds all peak abscissae and their smoothed ordinates in current data set:

Code Example – C# peak finding

```
pf.LocatePeaks();
```

Code Example – VB peak finding

```
PF.LocatePeaks()
```

The provided indexer on **PeakFinderSavitzkyGolay** gets each peak as an instance of struct **Extrema**. Property `NumberPeaks` gets the total number of peaks found. For example, this code dump all peaks to the console:

Code Example – C# peak finding

```
for (int i = 0; i < pf.NumberPeaks; i++)
{
  Extrema peak = pf[i];
  Console.WriteLine("Found peak at = ({0},{1})", peak.X, peak.Y);
}
```

Code Example – VB peak finding

```
For I As Integer = 0 To PF.NumberPeaks - 1
  Dim Peak As Extrema = PF(I)
  Console.WriteLine("Found peak at = ({0},{1})", Peak.X, Peak.Y)
Next
```

Advanced Savitzky-Golay Peak Finder Properties

Additional properties on **PeakFinderSavitzkyGolay** control the set of peaks that are found by the `LocatePeaks()` method:

- `SlopeSelectivity` gets and sets the slope selectivity. The selectivity of the peak finder can be reduced by increasing the `SlopeSelectivity`. If `SlopeSelectivity` is set to `0` (default), all found peaks are reported.
- `AbscisaInterval` gets and sets the abscissa interval for the data. This is used to scale the derivatives to the correct units. For proper scaling of the peak abscissa locations, set `AbscissaInterval` to the data sample interval.
- `RootFindingTolerance` gets and sets the error tolerance for the underlying **RiddersRootFinder**. The default is `0.00001`.

For instance:

Code Example – C# peak finding

```
pf.AbscissaInterval = 0.1;
pf.SlopeSelectivity = 0;
pf.LocatePeaks();
```

Code Example – VB peak finding

```
PF.AbscissaInterval = 0.1
PF.SlopeSelectivity = 0
PF.LocatePeaks()
```

14.4 Rule-Based Peak Finding

Class **PeakFinderRuleBased** finds peaks subject to rules about peak height and peak separation. A peak is defined as a point which is higher that both neighbors or infinity. Non-infinite end points are excluded as a peak. This class is analogous to MATLAB's `findpeaks()` function.

Creating Rule-Based Peak Finders

A **PeakFinderRuleBased** instance is constructed from a vector of data.

Code Example – C# rule-based peak finding

```
var x = new DoubleVector(5000, 0.01, 0.1);
DoubleVector data = NMathFunctions.Sin(x) / x;
var pf = new PeakFinderRuleBased( data );
```

Code Example – VB rule-based peak finding

```
Dim X As New DoubleVector(5000, 0.01, 0.1)
Dim Data As DoubleVector = NMathFunctions.Sin(X) / X
Dim PF As New PeakFinderRuleBased(Data)
```

Adding Rules

Peak finding rule types are specified with the **PeakFinderRuleBased.Rules** enumeration.

- `Rules.MinHeight`

  Removes peaks that have an amplitude less than a specified amount.

- `Rules.Threshold`

  Find peaks that are at least a specified amount higher than their neighboring samples.

Rules are added using the `AddRule()` method, and removed using `RemoveRule()`. Only one rule of each type is allowed. After updating the rule list, call either `LocatePeaks()` or `LocatePeakIndices()` to update the peak inventory.

For example, this rule finds all peaks with an amplitude greater than `1.5`.

Code Example – C# rule-based peak finding

```
pf.AddRule( PeakFinderRuleBased.Rules.MinHeight, 1.5 );
pf.LocatePeaks();
```

Code Example – VB rule-based peak finding

```
PF.AddRule( PeakFinderRuleBased.Rules.MinHeight, 1.5 );
PF.LocatePeaks();
```

If a `Rules.MinHeight` rule was already specified, it is removed before adding the new rule.

Rule-Based Peak Finder Results

The provided indexer on **PeakFinderRuleBased** gets each peak as an instance of struct **Extrema**. Property `NumberPeaks` gets the total number of peaks found. For example, this code dump all peaks to the console:

Code Example – C# peak finding

```
for (int i = 0; i < pf.NumberPeaks; i++)
{
  Extrema peak = pf[i];
  Console.WriteLine("Found peak at = ({0},{1})", peak.X, peak.Y);
```

```
}
```

Code Example – VB peak finding

```
For I As Integer = 0 To PF.NumberPeaks - 1
  Dim Peak As Extrema = PF(I)
  Console.WriteLine("Found peak at = ({0},{1})", Peak.X, Peak.Y)

Next
```

CHAPTER 15.
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.

15.1 Special Functions

Class **SpecialFunctions** provides the special functions shown in Table 1.

Table 11 – NMath Special Functions

Function	Description
`Airy()`	The Airy and Bairy functions are the two solutions of the differential equation `y''(x) = xy.`
`BesselI0()`	Modified Bessel function of the first kind, order zero.
`BesselI1()`	Modified Bessel function of the first kind, first order.
`BesselIv()`	Modified Bessel function of the first kind, non-integer order.
`BesselJ0()`	Bessel function of the first kind, order zero.
`BesselJ1()`	Bessel function of the first kind, first order.
`BesselJn()`	Bessel function of the first kind, arbitrary integer order.
`BesselJv()`	Bessel function of the first kind, non-integer order.
`BesselK0()`	Modified Bessel function of the second kind, order zero.

Table 11 – NMath Special Functions

Function	Description
`BesselK1()`	Modified Bessel function of the second kind, order one.
`BesselKn()`	Modified Bessel function of the second kind, arbitrary integer order.
`BesselY0()`	Bessel function of the second kind, order zero.
`BesselY1()`	Bessel function of the second kind, order one.
`BesselYn()`	Bessel function of the second kind of integer order.
`BesselYv()`	Bessel function of the second kind, non-integer order.
`Beta()`	The beta function (also known as the Eulerian integral of the first kind): `Beta(a, b) = Gamma(a) * Gamma(b) / Gamma(a+b)`.
`Binomial()`	The binomial coefficient (n choose k)—the number of ways of picking k unordered outcomes from n possibilities.
`BinomialLn()`	The natural log of the binomial coefficient.
`Cn()`	Jacobian elliptic function *Cn()* for real, pure imaginary, or complex arguments.
`Digamma()`	The digamma, or psi, function, defined as `Gamma'(z)/Gamma(z)`.
`Ei()`	Exponential integral.
`EllipJ()`	The real valued Jacobi elliptic functions.
`EllipticE()`	The complete elliptic integral of the second kind.
`EllipticF()`	The incomplete elliptic integral of the first kind.
`EllipticK()`	The complete elliptic integral, *K(m)*, of the first kind.
`EulerGamma`	A constant, also known as the Euler-Macheroni constant. Famously, rationality unknown.

Table 11 – NMath Special Functions

Function	**Description**
`Factorial()`	Factorial. The number of ways that *n* objects can be permuted.
`FactorialLn()`	The natural log factorial of *n*, `ln(n!)`.
`Gamma()`	The gamma function.
`GammaLn()`	The natural log of the gamma function.
`GammaReciprocal()`	The reciprocal of the gamma function.
`HarmonicNumber()`	The harmonic number, *Hn*, is a truncated sum of the harmonic series.
`Hypergeometric1F1()`	The confluent hypergeometric series of the first kind.
`Hypergeometric2F1()`	The Gauss or generalized hypergeometric function.
`IncompleteBeta()`	The incomplete beta function().
`IncompleteGamma()`	The incomplete gamma integral.
`IncompleteGammaComplement()`	The complemented incomplete gamma integral.
`PolyLogarithm()`	The polylogarithm, *Li_n(x)*.
`Sn()`	The Jacobian elliptic function *Sn()* for real, pure imaginary, or complex arguments.
`Zeta()`	The Riemann zeta function.

Using these special functions in your code is easy.

Code Example – C# special functions

```
// 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 );
```

Code Example – VB special functions

```
' Compute the Jacobi function Sn() with a complex argument.
Dim Complex As New DoubleComplex( 0.1, 3.3 )
Dim SN as DoubleComplex = SpecialFunctions( Complex, 0.3 )
```

```
' Compute the elliptic integral, K(m)
Dim EI as Double = SpecialFunctions.EllipticK( 0.432 )
```

PART III - MATRIX ANALYSIS

CHAPTER 16.
MATRIX FUNCTIONS

The `CenterSpace.NMath.Core` namespace provides the following matrix and linear algebra functionality:

- 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.
- Classes for solving symmetric, Hermitian, and nonsymmetric eigenvalue problems.

To avoid using fully qualified names, preface your code with an appropriate namespace statement:

Code Example – C#

```
using CenterSpace.NMath.Core;
```

Code Example – VB

```
imports CenterSpace.NMath.Core
```

CHAPTER 17.
STRUCTURED SPARSE MATRIX TYPES

NMath provides a wide variety of structured sparse matrix types, including triangular, symmetric, Hermitian, banded, tridiagonal, symmetric banded, and Hermitian banded.

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.

This chapter describes the **NMath** structured sparse matrix types, and the storage schemes they use. See Chapter 19 for general sparse matrix classes.

17.1 Lower Triangular Matrices

A *lower triangular* matrix is a square matrix with all elements above the main diagonal equal to zero. That is, $a_{ij} = 0$ for $i < j$. For example, this is a 4 x 4 lower triangular matrix:

$$\begin{bmatrix} 3 & 0 & 0 & 0 \\ 2 & -1 & 0 & 0 \\ 1 & -2 & 2 & 0 \\ 1 & 3 & 4 & 2 \end{bmatrix}$$

Lower triangular matrices often arise at an intermediate stage in solving systems of equations and inverting matrices.

NMath provides lower triangular matrix classes for four datatypes: single- and double-precision floating point numbers, and single- and double-precision complex numbers. The classnames are **FloatLowerTriMatrix**, **DoubleLowerTriMatrix**, **FloatComplexLowerTriMatrix**, and **DoubleComplexLowerTriMatrix**.

For efficiency, zero elements above the main diagonal are not stored. Instead, matrix values are stored in a vector row by row. For example, the following 5 x 5 lower triangular matrix:

$$A = \begin{bmatrix} a_{00} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 & 0 \\ a_{20} & a_{21} & a_{22} & 0 & 0 \\ a_{30} & a_{31} & a_{32} & a_{33} & 0 \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

is stored in a data vector as:

```
v = [ a00 a10 a11 a20 a21 a22 a30 a31 a32 a33 a40 a41 a42 a43 a44 ]
```

In general, the relationship between matrix and vector indices is:

```
A[i,j] = v[i(i+1)/2 + j]
```

17.2 Upper Triangular Matrices

An *upper triangular* matrix is a square matrix with all elements below the main diagonal equal to zero. That is, $a_{ij} = 0$ for $i > j$. For example, this is a 4 x 4 upper triangular matrix:

$$\begin{bmatrix} 2 & 0 & 1 & 1 \\ 0 & -1 & -2 & 3 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

Like lower triangular matrices, upper triangular matrices often arise at an intermediate stage in solving systems of equations and inverting matrices.

NMath provides upper triangular matrix classes for four datatypes: single- and double-precision floating point numbers, and single- and double-precision complex numbers. The classnames are **FloatUpperTriMatrix**, **DoubleUpperTriMatrix**, **FloatComplexUpperTriMatrix**, and **DoubleComplexUpperTriMatrix**.

For efficiency, zero elements below the main diagonal are not stored. Instead, matrix values are stored in a vector column by column. For example, the following 5 x 5 upper triangular matrix:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ 0 & a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & 0 & a_{44} \end{bmatrix}$$

is stored in a data vector as:

```
v = [ a00 a01 a11 a02 a12 a22 a03 a13 a23 a33 a04 a14 a24 a34 a44 ]
```

In general, the relationship between matrix and vector indices is:

```
A[i,j] = v[i + j(j+1)/2]
```

17.3 Symmetric 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. For example, this is a 4 x 4 symmetric matrix:

$$\begin{bmatrix} 2 & 0 & 1 & 1 \\ 0 & -1 & -2 & 3 \\ 1 & -2 & 0 & 4 \\ 1 & 3 & 4 & 2 \end{bmatrix}$$

Symmetric matrices are often used to represent quadratic forms.

NMath provides symmetric matrix classes for single- and double-precision floating point numbers. The classnames are **FloatSymmetricMatrix** and **DoubleSymmetricMatrix**. Hermitian matrices are a generalization of symmetric matrices for complex types (Section 17.4).

For efficiency, only the upper triangle is stored. The storage scheme is the same as for an upper triangular matrix (Section 17.2).

17.4 Hermitian Matrices

A *Hermitian* matrix is a square matrix which satisfies $A = \overline{A}^T$ where $\overline{A}^T$ denotes the conjugate transpose of A. That is, $a_{ij} = \overline{a_{ji}}$ for all i,j, where $\bar{z}$ denotes the complex conjugate. (The conjugate of a complex number $a + bi$ is defined as $a - bi$.) For example, this is a 4 x 4 Hermitian matrix:

$$\begin{bmatrix} -1 & 1-i & 1+2i & -i \\ 1+i & 3 & -2 & 3-2i \\ 1-2i & -2 & 0 & 4 \\ i & 3+2i & 4 & 2 \end{bmatrix}$$

According to the strict definition of a Hermitian matrix, the diagonal elements must be real numbers, since $a = \bar{a}$ only for real numbers, while other elements may be complex. **NMath** relaxes this requirement and permits complex elements on the diagonal. The provided `MakeDiagonalReal()` method sets the imaginary parts on the main diagonal to zero, thereby meeting the strict definition of a Hermitian matrix.

NMath provides Hermitian matrix classes for single- and double-precision complex numbers. The classnames are **FloatHermitianMatrix** and **DoubleHermitianMatrix**. A symmetric matrix is a special case of a Hermitian matrix where all the elements are real (Section 17.3).

For efficiency, only the upper triangle is stored. The storage scheme is the same as for an upper triangular matrix (Section 17.2).

17.5 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. That is, if ub is the upper bandwidth, and lb is the lower bandwidth, then $a_{ij} = 0$ whenever $j - i > ub$ or $i - j > lb$.

For example, this is a 7 x 7 banded matrix with upper bandwidth 1 and lower bandwidth 3:

$$A = \begin{bmatrix} 1 & -2 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 3 & 0 & 0 & 0 & 0 \\ 2 & 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & -4 & -2 & 1 & 4 & 0 & 0 \\ 0 & 2 & 2 & 1 & 3 & 1 & 0 \\ 0 & 0 & -1 & 2 & -3 & 0 & 2 \\ 0 & 0 & 0 & 3 & 3 & -1 & 1 \end{bmatrix}$$

NMath provides banded matrix classes for four datatypes: single- and double-precision floating point numbers, and single- and double-precision complex numbers. The classnames are **FloatBandMatrix**, **DoubleBandMatrix**, **FloatComplexBandMatrix**, and **DoubleComplexBandMatrix**.

For efficiency, zero elements outside the bandwidth are not stored. Instead, matrix values are stored in a vector column by column. Blank entries are inserted in the data vector so that the each column takes up the same number of elements, $ub + lb + 1$, in the vector. For example, the following 8 x 8 matrix with an upper bandwidth of 2 and a lower bandwidth of 1:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & 0 & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & a_{13} & 0 & 0 & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} & a_{35} & 0 & 0 \\ 0 & 0 & 0 & a_{43} & a_{44} & a_{45} & a_{46} & 0 \\ 0 & 0 & 0 & 0 & a_{54} & a_{55} & a_{56} & a_{57} \\ 0 & 0 & 0 & 0 & 0 & a_{65} & a_{66} & a_{67} \\ 0 & 0 & 0 & 0 & 0 & 0 & a_{76} & a_{77} \end{bmatrix}$$

is stored in a data vector as:

```
v = [x   x   a00 a10
     x   a01 a11 a21
     a02 a12 a22 a32
     a13 a23 a33 a43
     a24 a34 a44 a54
     a35 a45 a55 a65
     a46 a56 a66 a76
     a57 a67 a77 x  ]
```

where *x* denotes an unused location.

17.6 Tridiagonal Matrices

A tridiagonal matrix is a matrix which has all its non-zero entries on the main diagonal, the superdiagonal, and the subdiagonal. That is, $a_{ij} = 0$ whenever $j - i > 1$ or $i - j > 1$. For example, this is a 5 x 5 tridiagonal matrix:

$$A = \begin{bmatrix} 1 & -2 & 0 & 0 & 0 \\ 1 & -1 & 3 & 0 & 0 \\ 0 & 5 & 0 & -1 & 0 \\ 0 & 0 & -2 & 1 & 4 \\ 0 & 0 & 0 & 1 & 3 \end{bmatrix}$$

Tridiagonal matrices often occur in one-dimensional problems and at an intermediate stage in the process of finding eigenvalues.

NMath provides tridiagonal matrix classes for four datatypes: single- and double-precision floating point numbers, and single- and double-precision complex numbers. The classnames are **FloatTriDiagMatrix**, **DoubleTriDiagMatrix**, **FloatComplexTriDiagMatrix**, and **DoubleComplexTriDiagMatrix**.

For efficiency, zero elements outside the main diagonal, superdiagonal, and subdiagonal are not stored. A tridiagonal matrix is a special case of a banded matrix where the upper and lower bandwidths are one, and the storage scheme is the same as for a banded matrix (Section 17.5).

17.7 Symmetric Banded Matrices

A symmetric banded matrix is a symmetric matrix (Section 17.3) 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. That is, if hb is the half bandwidth, then $a_{ij} = 0$ whenever $j - i > hb$ or $i - j > hb$. For example, this is a 5 x 5 symmetric banded matrix with a half bandwidth of 1:

$$A = \begin{bmatrix} 1 & -2 & 0 & 0 & 0 \\ -2 & -1 & 3 & 0 & 0 \\ 0 & 3 & 0 & -1 & 0 \\ 0 & 0 & -1 & 1 & 4 \\ 0 & 0 & 0 & 4 & 3 \end{bmatrix}$$

Symmetric banded matrices often arise in one-dimensional finite element problems.

NMath provides symmetric banded matrix classes for single- and double-precision floating point numbers. The classnames are **FloatSymBandMatrix** and **DoubleSymBandMatrix**. Hermitian banded matrices are a generalization of symmetric banded matrices for complex types (Section 17.8).

For efficiency, the lower triangular part of the matrix and zero elements outside the bandwidth are not stored. Instead, matrix values are stored in a vector column by column. Blank entries are inserted in the data vector so that the each column takes up the same number of elements, $hb + 1$, in the vector. For example, the following 8 x 8 matrix with a half bandwidth of 2:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & 0 & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & a_{13} & 0 & 0 & 0 & 0 \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 & 0 \\ 0 & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & 0 & 0 \\ 0 & 0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & 0 \\ 0 & 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} \\ 0 & 0 & 0 & 0 & a_{64} & a_{65} & a_{66} & a_{67} \\ 0 & 0 & 0 & 0 & 0 & a_{75} & a_{76} & a_{77} \end{bmatrix}$$

is stored in a data vector as:

```
v = [x   x   a00
     x   a01 a11
     a02 a12 a22
     a13 a23 a33
     a24 a34 a44
     a35 a45 a55
     a46 a56 a66
     a57 a67 a77 ]
```

where *x* denotes an unused location.

17.8 Hermitian Banded Matrices

A Hermitian banded matrix is a Hermitian matrix (Section 17.4) 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. That is, if hb is the half bandwidth, then $a_{ij} = 0$ whenever $j - i > hb$ or $i - j > hb$. For example, this is a 5 x 5 Hermitian banded matrix with a half bandwidth of 1:

$$A = \begin{bmatrix} 1 & 2i & 0 & 0 & 0 \\ -2i & -1 & 3-i & 0 & 0 \\ 0 & 3+i & 0 & 1-5i & 0 \\ 0 & 0 & 1+5i & 1 & 4 \\ 0 & 0 & 0 & 4 & 3 \end{bmatrix}$$

According to the strict definition of a Hermitian matrix, the diagonal elements must be real numbers, since $a = \bar{a}$ only for real numbers), while other elements may be complex. **NMath** relaxes this requirement and permits complex elements on the diagonal. The provided `MakeDiagonalReal()` method sets the imaginary parts on the main diagonal to zero, thereby meeting the strict definition of a Hermitian matrix.

NMath provides Hermitian banded matrix classes for single- and double-precision complex numbers. The classnames are **FloatHermitianBandMatrix** and **DoubleHermitianBandMatrix**. A symmetric banded matrix is a special case of a Hermitian banded matrix where all the elements are real (Section 17.7).

For efficiency, the lower triangular part of the matrix and zero elements outside the bandwidth are not stored. The storage scheme is the same as for a symmetric banded matrix (Section 17.7).

CHAPTER 18. USING THE STRUCTURED SPARSE MATRIX CLASSES

NMath provides a variety of functions that take the structured sparse matrix types described in Chapter 17 as arguments. Methods are provided either as member functions on the matrix classes, or as static methods on class **MatrixFunctions**.

As a general rule, **NMath** only provides functions that preserve the shape of the structured sparse matrices. In some cases, this means that functions provided for the general matrix classes are not provided for the structured sparse matrix classes. For example, **NMath** does not generally provide trigonometric and transcendental functions for structured sparse matrix types. Such functions may change unstored zero values to non-zero values, thus changing a structured sparse matrix type into a general matrix.

If you want to apply an arbitrary function to all elements of a structured sparse matrix, including unstored zero values, you can always convert the matrix to a general matrix first. A `ToGeneralMatrix()` method is provided for this purpose. Alternatively, to apply an arbitrary function only to stored values, you can apply the function to the underlying data vector. Both techniques are described in more detail in Section 18.7.

This chapter describes how to create and manipulate the **NMath** structured sparse matrix types.

18.1 Creating Matrices

This section describes how to create instances of the structured sparse matrix classes.

Creating Default Matrices

You can construct default structured sparse matrices by supplying the necessary parameters to describe the matrix shape, as shown in Table 12. All stored values are initialized to zero.

Table 12 – Structured sparse matrix shape parameters

Matrix Type	Shape Parameters
Lower Triangular	Order
Upper Triangular	Order
Symmetric	Order
Hermitian	Order
Banded	Rows, Columns, Lower Bandwidth, Upper Bandwidth
TriDiagonal	Rows, Columns
Symmetric Banded	Order, Half Bandwidth
Hermitian Banded	Order, Half Bandwidth

Square matrix types are characterized by their *order*--that is, the number of rows and columns. For example, a matrix of order 3 is a 3 x 3 matrix. Thus, this code creates a default 5 x 5 Hermitian matrix of double-precision complex numbers:

Code Example – C# matrix

```
var A = new DoubleHermitianMatrix( 5 );
```

Code Example – VB matrix

```
Dim A As New DoubleHermitianMatrix(5)
```

Constructors for rectangular matrix types accept separate *row* and *column* shape parameters. For example:

Code Example – C# matrix

```
var A = new DoubleTriDiagMatrix ( 3, 5 );
```

Code Example – VB matrix

```
Dim A As New DoubleTriDiagMatrix(3, 5)
```

Constructors for banded matrix types also accept *bandwidth* parameters that describe the width of the banded region. Thus, the following code creates a 4 x 5 **FloatComplexBandMatrix** with a *lower bandwidth* of 1 and an *upper bandwidth* of 2:

Code Example – C# matrix

```
var A = new FloatComplexMatrix( 4, 5, 1, 2 );
```

Code Example – VB matrix

```
Dim A As New FloatComplexMatrix(4, 5, 1.0F, 2.0F)
```

This creates an 8 x 8 **FloatSymBandMatrix** with a *half bandwidth* of 2:

Code Example – C# matrix

```
var A = new FloatSymBandMatrix( 8, 2 );
```

Code Example – VB matrix

```
Dim A As New FloatSymBandMatrix(8, 2)
```

Once you've constructed a default matrix, you can set individual values using the provided indexers (Section 18.2). In some case, methods are also provided that return vector views of the underlying data, which can also be used to set matrix values (Section 18.5).

Creating Sparse Matrices from General Matrices

You can construct all **NMath** structured sparse matrix types from general matrix types. Such constructors extract the appropriate values from the general matrix. Data is copied.

For example, this code constructs a **FloatUpperTriMatrix** instance by extracting the upper triangular region of a square general matrix:

Code Example – C# matrix

```
var genMat = new FloatMatrix( 5, 5, 0, 1 );
var A = new FloatUpperTriMatrix( genMat );
```

Code Example – VB matrix

```
Dim GenMat As New FloatMatrix(5, 5, 0.0F, 1.0F)
Dim A As New FloatUpperTriMatrix(GenMat)
```

Constructors for square matrix types, such as upper triangular matrices, throw a **MatrixNotSquareException** if the given general matrix is not square. Alternatively, you can pass in a non-square general matrix and specify the order of the square submatrix to extract. Thus, this code creates a 3 x 3 **DoubleSymmetricMatrix** by extracting the upper triangular region of the 3 x 3 leading submatrix from the given 4 x 6 general matrix:

Code Example – C# matrix

```
var genMat = new DoubleMatrix( 4, 6, 0, 0.25 );
var A = new DoubleSymmetricMatrix( A, 3 );
```

Code Example – VB matrix

```
Dim GenMat As New DoubleMatrix(4, 6, 0.0, 0.25)
```

```
Dim A As New DoubleSymmetricMatrix(A, 3)
```

An **IndexOutOfRangeException** is raised if the given order specifies a submatrix that is out of bounds.

Banded matrix types can also be constructed from general matrices by specifying the desired bandwidth. For instance, the following code extracts the values required to construct a Hermitian banded matrix with a half bandwidth of 3 from the given general matrix:

Code Example – C# matrix

```
var incr = new DoubleComplex( 1, 0.25 );
var genMat = new DoubleComplexMatrix( 12, 12, 0, incr );
var A = new DoubleHermitianBandMatrix( A, 3 );
```

Code Example – VB matrix

```
Dim Incr As New DoubleComplex(1.0, 0.25)
Dim GenMat As New DoubleComplexMatrix(12, 12, 0.0, Incr)
Dim A As New DoubleHermitianBandMatrix(A, 3)
```

Creating Sparse Matrices from Other Sparse Matrices

Some structured sparse matrix types can be constructed from other structured sparse matrices. For example, a tridiagonal matrix is really a special case of a banded matrix with lower and upper bandwidth equal to 1. Therefore, banded matrices can be constructed from tridiagonal matrices, and *vice versa*. For example:

Code Example – C# matrix

```
int rows = 8, cols = 8, ub = 0, lb = 2;
var data = new FloatVector( (ub+lb+1)*cols, 1, 1 );
var A = new FloatBandMatrix( data, rows, cols, lb, ub );
var B = new FloatTriDiagMatrix( A );
```

Code Example – VB matrix

```
Dim Rows As Integer = 8
Dim Cols As Integer = 8
Dim UB As Integer = 0
Dim LB As Integer = 2
Dim Data As New FloatVector((UB + LB + 1) * Cols, 1.0F, 1.0F)
Dim A As New FloatBandMatrix(Data, Rows, Cols, LB, UB)
Dim B As New FloatTriDiagMatrix(A)
```

Similarly, you can construct banded matrices from symmetric or Hermitian banded matrices, or triangular matrices from symmetric or Hermitian matrices, and *vice versa*.

Creating Sparse Matrices from a Data Vector

You can construct all **NMath** structured sparse matrix types from an appropriate data vector and shape parameters. The vector storage scheme used by each structured sparse matrix type is described in Chapter 17. For example, you could create this 4 x 4 symmetric matrix:

$$\begin{bmatrix} 0 & 1 & 3 & 6 \\ 0 & 2 & 4 & 7 \\ 0 & 0 & 5 & 8 \\ 0 & 0 & 0 & 9 \end{bmatrix}$$

like this:

Code Example – C# matrix

```
var data = new DoubleVector( 10, 0, 1 );
var A = new DoubleSymmetricMatrix( data, 4 );
```

Code Example – VB matrix

```
Dim Data As New DoubleVector(10, 0.0, 1.0)
Dim A As New DoubleSymmetricMatrix(Data, 4)
```

Similarly, you could create this 5 x 7 banded matrix with an upper bandwidth of 1 and a lower bandwidth of 0:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

using this code:

Code Example – C# matrix

```
var data = new FloatVector( 14, 1 );
var A = new FloatBandMatrix( data, 5, 7, 0, 1 );
```

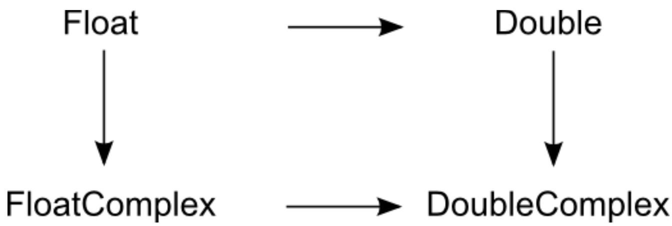
Code Example – VB matrix

```
Dim Data As New FloatVector(14, 1.0F)
Dim A As New FloatBandMatrix(Data, 5, 7, 0, 1)
```

Implicit Conversion

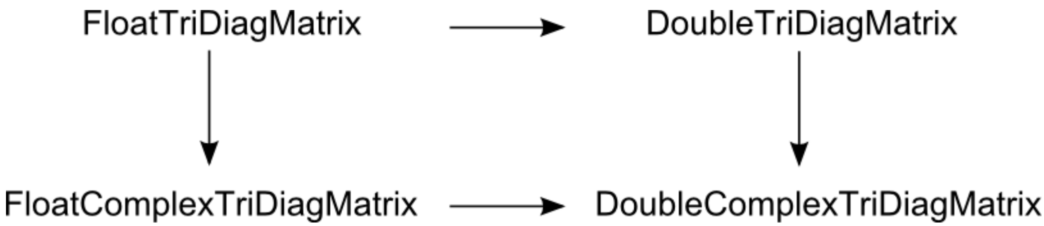
NMath provides implicit conversion operators for the structured sparse matrix classes. Single-precision types are implicitly promoted to double-precision types, and real types are implicitly promoted to complex types, as shown in Figure 4. An arrow indicates implicit promotion.

Figure 4 – Implicit conversion for matrix data types



For example, Figure 5 shows the pattern for implicit conversion among the tridiagonal types.

Figure 5 – Implicit conversion for tridiagonal matrices



Copying Matrices

The **NMath** structured sparse matrix classes provide three copy methods:

- `Clone()` returns a deep copy of a matrix. Data is copied; each matrix references different data.
- `ShallowCopy()` returns a shallow copy of a matrix. Data is not copied; both matrices reference the same data.
- `DeepenThisCopy()` copies the data viewed by a matrix to new data block. This guarantees that there is only one reference to the underlying data, and that this data is in contiguous storage.

For instance:

Code Example – C# matrix

```
var A = new FloatUpperTriMatrix( 5 );
```

```
FloatUpperTriMatrix B = A.ShallowCopy();

B[0,0] = 1;    // A[0,0] == B[0,0]
B.DeepenThisCopy();
B[0,0] = 2;    // A[0,0] != B[0,0]
```

Code Example – VB matrix

```
Dim A As New FloatUpperTriMatrix(5)
Dim B As FloatUpperTriMatrix = A.ShallowCopy()

B(0, 0) = 1    ' A[0,0] == B[0,0]
B.DeepenThisCopy()
B(0, 0) = 2    ' A[0,0] != B[0,0]
```

18.2 Value Operations on Matrices

The **NMath** structured sparse matrix classes have read-only properties for all shape parameters, and for the underlying data vector:

Table 13 – Structured sparse matrix shape parameters

Matrix Type	Read-Only Properties
Lower Triangular	Order, Rows, Cols, DataVector
Upper Triangular	Order, Rows, Cols, DataVector
Symmetric	Order, Rows, Cols, DataVector
Hermitian	Order, Rows, Cols, DataVector
Banded	Rows, Cols, LowerBandwidth, UpperBandwidth, Bandwidth, DataVector
TriDiagonal	Rows, Cols, DataVector
Symmetric Banded	Order, Rows, Cols, HalfBandwidth, Bandwidth, DataVector
Hermitian Banded	Order, Rows, Cols, HalfBandwidth, Bandwidth, DataVector

On square matrix types, the `Rows` and `Cols` properties simply return the order. On banded types, the `Bandwidth` property returns the total bandwidth. For general banded matrices, the total bandwidth is `LowerBandwidth + UpperBandwidth +`

1; for symmetric and Hermitian banded types, the total bandwidth is 2 * HalfBandwidth + 1.

For example, if A is a **FloatHermitianBandMatrix** instance:

Code Example – C# matrix

```
int order = A.Order;
int cols = A.Cols;              // cols == order
int rows = A.Rows;              // rows == order
int halfband = A.HalfBandwidth
int band = A.Bandwidth          // band = 2 * halfband + 1
FloatComplexVector data = A.DataVector;
```

Code Example – VB matrix

```
Dim Order As Integer = A.Order
Dim Cols As Integer = A.Cols            ' cols == order
Dim Rows As Integer = A.Rows            ' rows == order
Dim HalfBand As Integer = A.HalfBandwidth
Dim band As Integer = A.Bandwidth       ' band = 2 * halfband + 1
Dim Data As FloatComplexVector = A.DataVector
```

Accessing and Modifying Matrix Values

The matrix classes provide standard indexers for getting and setting element value at a specified row and column position in a matrix. Thus, A[i,j] always returns the element in the *i*th row and *j*th column of matrix A's view of the data.

NOTE—Indexing starts at 0.

Attempts to set zero elements outside the stored region to nonzero values raise a **NonModifiableElementException**. For instance:

Code Example – C# matrix

```
var A = new FloatComplexTriDiagMatrix( 8, 8 );
try
{
  A[7,0] = new FloatComplex( 1, -1 );
}
catch ( NonModifiableElementException )
{
  // Do something here
}
```

Code Example – VB matrix

```
Dim A As New FloatComplexTriDiagMatrix(8, 8)
Try
  A(7, 0) = New FloatComplex(1.0F, -1.0F)
Catch NonModifiableElementException
```

```
  ' Do something here
End Try
```

Symmetric matrices are in a different category than the other structured sparse matrix types, because unstored values are not constrained to be zero. Thus, even though only the upper triangular region is stored, you can “set” values in the lower triangular region. The corresponding value in the upper triangular region is changed. Thus:

Code Example – C# matrix

```
var A = new DoubleSymmetricMatrix( 12 );
Console.WriteLine( A[7,2] );  // "0"
Console.WriteLine( A[2,7] );  // "0"
A[7,2] = 1;
Console.WriteLine( A[7,2] );  // "1"
Console.WriteLine( A[2,7] );  // "1"
```

Code Example – VB matrix

```
Dim A As New DoubleSymmetricMatrix(12)
Console.WriteLine(A(7, 2))   ' "0"
Console.WriteLine(A(2, 7))   ' "0"
A(7, 2) = 1
Console.WriteLine(A(7, 2))   ' "1"
Console.WriteLine(A(2, 7))   ' "1"
```

Resizing a Matrix

The matrix classes provide `Resize()` methods for changing the size of a matrix after it has been created. Zeros are added or values are truncated as necessary. For instance:

Code Example – C# matrix

```
int order = 7;
var data =
  new DoubleComplexVector( ( order * ( order + 1 )) / 2,
  new RandGenMTwist() );
var A = new DoubleHermitianMatrix( data );

DoubleHermitianMatrix B2 = (DoubleHermitianMatrix)B.Clone();
B2.Resize( B.Order + 2 );
```

Code Example – VB matrix

```
Dim Order As Integer = 7
Dim Data As New DoubleComplexVector((Order * (Order + 1)) / 2,
  New RandGenMTwist())
Dim A As New DoubleHermitianMatrix(Data)
```

```
Dim B2 As DoubleHermitianMatrix = CType(B.Clone(),
  DoubleHermitianMatrix)
B.Resize(B.Order + 2)
```

18.3 Logical Operations on Matrices

Operator `==` tests for equality of two matrices, and returns `true` if both matrices have the same dimensions and all values are equal; otherwise, `false`. Following the convention of the .NET Framework, if both objects are `null`, they test equal.

The comparison of the values for double-precision floating point and complex numbers is done using operator `==` for doubles; comparison of the values for single precision numbers is done using operator `==` for floats. Therefore, the values of the matrices must be exactly equal for this method to return `true`. Operator `!=` returns the logical negation of `==`.

The `Equals()` member function also tests for equality.

18.4 Arithmetic Operations on Matrices

NMath provides overloaded arithmetic operators for structured sparse matrices with their conventional meanings for those .NET languages that support them, and equivalent named methods for those that do not. Table 14 lists the equivalent operators and methods

Table 14 – Arithmetic operators

Operator	Equivalent Named Method
`+`	`Add()`
`-`	`Subtract()`
`*`	`Multiply()`
`/`	`Divide()`
Unary `-`	`Negate()`

All binary operators and equivalent named methods work either with two matrices, or with a matrix and a scalar.

NOTE—Matrices must have the same dimensions to be combined using the element-wise operators. Otherwise, a MismatchedSizeException is raised.

For example, this C# code uses the overloaded operators:

Code Example – C# matrix

```
var genMat = new DoubleMatrix( 8, 8, 0, 1 );
var A = new DoubleBandMatrix( genMat, 4, 5, 1, 2 );
var B = new DoubleBandMatrix( genMat, 4, 5, 1, 1 );
double s = 2.25;

DoubleBandMatrix result = A + s*B;
```

Note that although the banded matrices must have the same dimensions, they do not need to have the same bandwidth.

This Visual Basic code uses the equivalent named methods:

Code Example – VB matrix

```
Dim genMat As new DoubleMatrix( 8, 8, 0, 1 )
Dim A As new DoubleBandMatrix( genMat, 4, 5, 1, 2 );
Dim B As new DoubleBandMatrix( genMat, 4, 5, 1, 1 );
Dim s As Double = 2.25;

Dim result As DoubleBandMatrix = _
   DoubleBandMatrix.Add(A, DoubleBandMatrix.Multiply(s, B));
```

18.5 Vector Views

Methods such as `Row()`, `Column()`, and `Diagonal()` return vector views of the data referenced by a general matrix. **NMath** does not generally provide such methods for structured sparse matrix types, because of the limitations on which elements in the matrix are modifiable.

The exception is the banded matrix types which provide a `Diagonal()` member function that returns a vector view of a diagonal of a matrix. If no diagonal is specified, a vector view of the main diagonal is returned. For example, this code increments every element along the main diagonal:

Code Example – C# matrix

```
var A = new FloatBandMatrix( 5, 5, 0, 0 );
A.Diagonal()++;
```

Code Example – VB matrix

```
Dim A As New FloatBandMatrix( 5, 5, 0, 0 )
A.Diagonal().Increment()
```

18.6 Functions of Matrices

NMath provides a variety of functions that take structured sparse matrices as arguments.

Matrix Transposition

The structured sparse matrix classes provide `Transpose()` member functions for calculating the transpose of a matrix: `B[i,j] = A[j,i]`. Class **MatrixFunctions** also provides a static `Transpose()` method that returns the transpose of a matrix. Data is copied. For instance:

Code Example – C# matrix

```
var A = new FloatTriDiagMatrix( 50, 50 );
A.Diagonal(1)++;    // increments the superdiagonal
A.Diagonal(-1)--;   // decrements the subdiagonal
FloatTriDiagMatrix B = A.Transpose();
FloatTriDiagMatrix C = MatrixFunctions.Transpose( A );   // B == C
```

Code Example – VB matrix

```
Dim A As New FloatTriDiagMatrix(50, 50)
A.Diagonal(1).Increment()     ' increments the superdiagonal
A.Diagonal(-1).Decrement()    ' decrements the subdiagonal
Dim B As FloatTriDiagMatrix = A.Transpose()
Dim C As FloatTriDiagMatrix = MatrixFunctions.Transpose(A) ' B == C
```

NOTE—By definition, a symmetric matrix is equal to its own transpose, so the Transpose() method has no effect for these types.

Matrix Inner Products

Class **MatrixFunctions** provides the static `Product()` method for calculating the inner product of a matrix and a vector:

Code Example – C# matrix

```
var data = new DoubleVector( 10, 1, 1 );
var A = new DoubleSymmetricMatrix( data, 4 );
var x = new DoubleVector( 4, 1, 1 );
DoubleVector y = MatrixFunctions.Product( A, x );
```

Code Example – VB matrix

```
Dim Data As New DoubleVector(10, 1.0, 1.0)
Dim A As New DoubleSymmetricMatrix(Data, 4)
Dim X As New DoubleVector(4, 1.0, 1.0)
Dim Y As DoubleVector = MatrixFunctions.Product(A, X)
```

For banded matrices, additional overloads of the `Product()` method calculate the product of two matrices. For example:

Code Example – C# matrix

```
int rows = 8, cols = 6, lb = 2, ub = 1;
DoubleComplexVector data =
  new DoubleComplexVector( (ub+lb+1)*cols, 0, 1 );
DoubleComplexBandMatrix A =
  new DoubleComplexBandMatrix( data, rows, cols, lb, ub );
DoubleComplexBandMatrix B =
  new DoubleComplexBandMatrix( ++data, cols, cols, lb, ub );
DoubleComplexBandMatrix C =
  MatrixFunctions.Product( A, B );
```

Code Example – VB matrix

```
Dim Rows As Integer = 8
Dim Cols As Integer = 6
Dim LB As Integer = 2
Dim UB As Integer = 1
Dim Data As New DoubleComplexVector((UB + LB + 1) * Cols, 0, 1)
Dim A As New DoubleComplexBandMatrix(Data, Rows, Cols, LB, UB)
Dim B As New DoubleComplexBandMatrix(Data.Increment(), Cols, Cols,
  LB, UB)
Dim C As DoubleComplexBandMatrix = MatrixFunctions.Product(A, B)
```

Also for banded matrices, the static `TransposeProduct()` method on **NMathFunctions** computes the matrix inner product of the transpose of a given matrix and a second matrix. Thus, assuming `A` and `B` are **DoubleBandMatrix** instances, this code calculates the inner product of the transpose of `A` with `B`:

Code Example – C# matrix

```
DoubleBandMatrix C = MatrixFunctions.TransposeProduct( A, B );
```

Code Example – VB matrix

```
Dim C As DoubleBandMatrix = MatrixFunctions.TransposeProduct(A, B)
```

Matrix Norms

MatrixFunctions provides static functions `OneNorm()` to compute the 1-norm (or largest column sum) of a matrix, and `InfinityNorm()` to compute the infinity-norm (or largest row sum) of a matrix. For instance:

Code Example – C# matrix

```
var A = new DoubleMatrix( "3x3 [1 2 3  4 5 6  7 8 9]" );
double d1 = A.OneNorm();
double d2 = A.InfinityNorm();
```

Code Example – VB matrix

```
Dim A As New DoubleMatrix("3x3 [1 2 3  4 5 6  7 8 9]")
Dim D1 As Double = A.OneNorm()
Dim D2 As Double = A.InfinityNorm()
```

The `OneNorm()` method has overloads for all structured sparse matrix types; `InfinityNorm()` only for banded and tridiagonal types.

Trigonometric and Transcendental Functions

In general, **NMath** does not provide trigonometric and transcendental functions for sparse matrix types. Such functions may change unstored zero values to non-zero values, thus changing a sparse matrix type into a general matrix. If you want to apply a trigonometric or transcendental function to all elements of a sparse matrix, including unstored zero values, convert the matrix to a general matrix first, using the `ToGeneralMatrix()` method. Alternatively, to apply a trigonometric or transcendental function only to stored values, apply the function to the underlying data vector. These techniques are described in more detail in Section 18.7.

Symmetric matrices are in a different category than the other sparse matrix types, because unstored values are not constrained to be zero. Therefore, **NMath** extends standard trigonometric functions `Acos()`, `Asin()`, `Atan()`, `Cos()`, `Cosh()`, `Sin()`, `Sinh()`, `Tan()`, and `Tanh()` to take symmetric matrix arguments. Class **MatrixFunctions** provides these functions as static methods. For instance, this code construct a symmetric matrix whose contents are the cosines of another symmetric matrix:

Code Example – C# matrix

```
var genMat = new FloatMatrix( 10, 10, 0, Math.Pi/4 );
var A = new FloatSymmetricMatrix( genMat );
FloatSymmetricMatrix cosA = MatrixFunctions.Cos( A );
```

Code Example – VB matrix

```
Dim GenMat As New FloatMatrix(10, 10, 0.0F, Math.PI / 4.0F)
Dim A As New FloatSymmetricMatrix(GenMat)
Dim CosA As FloatSymmetricMatrix = MatrixFunctions.Cos(A)
```

The static `Atan2()` method takes two symmetric matrices and applies the two-argument arc tangent function to corresponding pairs of elements.

MatrixFunctions also provides standard transcendental functions `Log()` and `Log10()` that take symmetric matrix arguments.

Absolute Value

The static `Abs()` function on class **MatrixFunctions** applies the absolute value function to each element of a given matrix:

Code Example – C# matrix

```
int order = 5, hb = 2;
var data = new DoubleComplexVector( "(0,0) (0,0)
  (1,-2) (0,0) (2,-4) (3,-6) (4,-8) (5,-10) (6,-12) (7,-14) (8,-16)
  (9,-18) (10,-20) (11,-22) (12,-24)" );
var A = new DoubleHermitianBandMatrix( data, order, hb );
DoubleSymBandMatrix B = MatrixFunctions.Abs( A );
```

Code Example – VB matrix

```
Dim Order As Integer = 5
Dim HB As Integer = 2
Dim Data As New DoubleComplexVector("(0,0) (0,0) " & _
  "(1,-2) (0,0) (2,-4) (3,-6) (4,-8) (5,-10) (6,-12) (7,-14) " &
  "(8,-16) (9,-18) (10,-20) (11,-22) (12,-24)")
Dim A As New DoubleHermitianBandMatrix(Data, Order, HB)
Dim B As DoubleSymBandMatrix = MatrixFunctions.Abs(A)
```

Complex Matrix Functions

Static methods `Real()` and `Imag()` on class **MatrixFunctions** return the real and imaginary part of the elements of a matrix. If the elements of the given matrix are real, `Real()` simply returns the given matrix and `Imag()` returns a matrix of the same dimensions containing all zeros.

Static methods `Arg()` and `Conj()` on class **MatrixFunctions** return the arguments (or phases) and complex conjugates of the elements of a matrix. If the elements of the given matrix are real, both methods simply return the given matrix.

For instance:

Code Example – C# matrix

```
var initValue = new FloatComplex( 1, -1.5F );
var increment = new FloatComplex( 1, 0.25F );
var getMat = new FloatComplexMatrix( 7, 6, initValue, increment );
var A = new FloatComplexTriDiagMatrix( getMat );

FloatTriDiagMatrix AImag = MatrixFunctions.Imag( A );
```

Code Example – VB matrix

```
Dim InitValue As New FloatComplex(1, -1.5F)
Dim Increment As New FloatComplex(1, 0.25F)
Dim GetMat As New FloatComplexMatrix(7, 6, InitValue, Increment)
Dim A As New FloatComplexTriDiagMatrix(GetMat)

Dim AImag As FloatTriDiagMatrix = MatrixFunctions.Imag(A)
```

18.7 Generic Functions

NMath provides convenience methods for applying unary and binary functions to elements of a general matrix. The `Apply()` method returns a new matrix whose contents are the result of applying a given function delegate to each element of the matrix. The `Transform()` method modifies a matrix object by applying the given function to each of its elements.

NMath, however, does not generally support applying arbitrary functions to structured sparse matrix types. As described in Section 18.6, such functions may change unstored zero values to non-zero values, thus changing a structured sparse matrix type into a general matrix. Again, the exception is the symmetric matrices which are in a different category than the other sparse matrix types, because unstored values are not constrained to be zero. Therefore, **NMath** provides `Apply()` and `Transform()` methods on these types. For example:

Code Example – C# matrix

```
int order = 9;
DoubleVector data =
  new DoubleVector(( order * ( order + 1 )) / 2,
  new RandGenMTwist() );
data -= 0.5;
var A = new DoubleSymmetricMatrix( order, data );
DoubleSymmetricMatrix B = A.Apply( NMathFunctions.SinFunc );
A.Transform( NMathFunctions.CosFunc );
```

Code Example – VB matrix

```
Dim Order As Integer = 9
Dim Data As New DoubleVector((Order * (Order + 1)) / 2,
  New RandGenMTwist())
Data -= 0.5
Dim A As New DoubleSymmetricMatrix(Order, Data)
Dim B As DoubleSymmetricMatrix = A.Apply(NMathFunctions.SinFunc)
A.Transform(NMathFunctions.CosFunc)
```

The code above creates a 9 x 9 symmetric matrix filled with random values between 0 and 1, then applies the sine function to create a new symmetric matrix containing the sines of the original values. Finally, the original matrix is transformed using the cosine function.

For structured sparse matrix types other than symmetric, there are two ways to apply an arbitrary function: convert the matrix to a general matrix and apply the function, or retrieve the underlying data vector and apply the function. The difference is whether the function is applied to all elements of the matrix, including unstored zero values, or just to the stored values.

For instance, this code converts an upper triangular matrix to a general matrix, then applies the cosine function to all elements of general matrix, including the zero values in the lower triangular region:

Code Example – C# matrix

```
var data = new DoubleVector( 10, 0, Math.PI/4 );
var A = new DoubleUpperTriMatrix( data, 4 );

DoubleMatrix genMat = A.ToGeneralMatrix();
genMat.Transform( NMathFunctions.CosFunc );
```

Code Example – VB matrix

```
Dim Data As New DoubleVector(10, 0.0, Math.PI / 4.0)
Dim A As New DoubleUpperTriMatrix(Data, 4)

Dim GenMat As DoubleMatrix = A.ToGeneralMatrix()
GenMat.Transform(NMathFunctions.CosFunc)
```

NOTE—Data is copied when converting a structured sparse matrix to a general matrix.

In contrast, this code retrieves the underlying data vector from the upper triangular matrix, and transforms it using the cosine function, then creates a new upper triangular matrix using the new data:

Code Example – C# matrix

```
var data = new DoubleVector( 10, 0, Math.PI/4 );
var A = new DoubleUpperTriMatrix( data, 4 );

A.DataVector.Transform( NMathFunctions.CosFunc );
```

Code Example – VB matrix

```
Dim Data As New DoubleVector(10, 0.0, Math.PI / 4.0)
Dim A As New DoubleUpperTriMatrix(Data, 4)

A.DataVector.Transform(NMathFunctions.CosFunc)
```

In this case, the zeros in the lower triangular region have not been transformed, and the matrix remains an upper triangular matrix. No data was copied.

CHAPTER 19. 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 in Chapter 17, where the zero elements are distributed according to some pattern, general sparse matrices make no assumptions about the sparsity structure of the matrix.

NMath also provides classes for computing and storing factorizations of general sparse matrices. Once a factorization is constructed, it can be reused to solve for different right-hand sides.

This chapter describes the **NMath** general sparse vector and matrix classes.

19.1 Sparse Vectors

NMath provides four classes for storing sparse vectors:

- Class **FloatSparseVector** stores a sparse vector of `float` values.
- Class **DoubleSparseVector** stores a sparse vector of `double` values.
- Class **FloatComplexSparseVector** stores a sparse vector of **FloatComplex** values.
- Class **DoubleComplexSparseVector** stores a sparse vector of **DoubleComplex** values.

Only the non-zero elements are stored.

Storage Format

Class **SparseVectorData** stores sparse vector data, and is parameterized on the type, *T,* of values stored in the vector. The nonzero elements of the vector are stored in a resizable array of type *T*, and their corresponding indexes are stored in a separate, parallel array of integers.

For example, the vector

```
v = ( 0 0 1.15 0 3.14 0 0 -2.23 0 0 )
```

is stored as

```
values = ( 1.15, 3.14, -2.23 );
indices = ( 2, 4, 7 );
```

The sparse vector classes extend **SparseVectorData**.

Creating Sparse Vectors

Instances of sparse vectors are created in two ways: by providing the necessary storage arrays to constructors, or by *gathering* data from a dense vector. For example, this code uses a **FloatSparseVector** constructor:

Code Example – C# sparse vector

```
var indices = new IndexArray( 1, 12, 2, 15 );
var values = new float[] { 2, 3.14, -4, -.6 };
var sv = new FloatSparseVector( values, indices );
```

Code Example – VB sparse vector

```
Dim Indices As New IndexArray(1, 12, 2, 15)
Dim Values = New Single() {2.0, 3.14, -4.0, -0.6}
Dim SV As New FloatSparseVector(Values, Indices)
```

This code uses the `MatrixFunctions.Gather()` method to create a **DoubleSparseVector** from the non-zero elements in a **DoubleVector** `v`:

Code Example – C# sparse vector

```
DoubleSparseVector sv = MatrixFunctions.Gather( v );
```

Code Example – VB sparse vector

```
Dim SV As DoubleSparseVector = MatrixFunctions.Gather(V)
```

You can also create a sparse vector by selecting specific elements from a dense vector. For instance, this code creates a sparse vector containing the specified values from `v`:

Code Example – C# sparse vector

```
DoubleSparseVector sv = MatrixFunctions.Gather( v, indices );
```

Code Example – VB sparse vector

```
Dim SV As DoubleSparseVector = MatrixFunctions.Gather(Y, Indices)
```

Accessing and Modifying Sparse Vector Values

The sparse vector classes inherit the following properties from **SparseVectorData**:

- `Entries` gets and set the array of non-zero entries.
- `Indices` gets and sets the array of indices of the non-zero elements.
- `NumberNonZero` gets and sets the number of non-zero elements.

The sparse vector classes also provide standard indexers for getting and setting individual element values.

Code Example – C# sparse vector

```
int nnz = 3;
var sv = new DoubleSparseVector( nnz );
sv[4] = 10;
sv[100] = 3;
```

Code Example – VB sparse vector

```
Dim NNZ As Integer = 3
Dim SV As New DoubleSparseVector(NNZ)
SV(4) = 10
SV(100) = 3
```

Operations on Sparse Vectors

Operator `==` tests for equality of two sparse vectors, and returns `true` if both vecrtors have the same nonzero elements; otherwise, `false`. Following the convention of the .NET Framework, if both objects are `null`, they test equal. Operator `!=` returns the logical negation of `==`. The `Equals()` member function also tests for equality.

NMath provides overloaded arithmetic operators for general sparse vectors with their conventional meanings for those .NET languages that support them, and equivalent named methods for those that do not. All binary operators and equivalent named methods work either with two vectors, or with a vector and a scalar.

Code Example – C# sparse vector

```
double a = 3.18;
var sv1 = new DoubleSparseVector( data, indices );
DoubleSparseVector sv2 = a * sv1;
```

Code Example – VB sparse vector

```
Dim A As Double = 3.18
Dim SV1 As New DoubleSparseVector(Data, Indices)
```

```
Dim SV2 As DoubleSparseVector = A * SV1
```

Sparse Vector Functions

The sparse vector classes provide the following member functions that operate on the elements of the vector:

- `TwoNorm()` computes the Euclidean norm of the elements of a sparse vector.
- `Scale()` scales each element in a sparse vector by the specified value.

MatrixFunctions also provides a variety of functions that take general sparse vectors as arguments:

- `AbsSum` calculates the sum of the absolute value of a given vector's elements.
- `MaxAbsIndex` calculates the index of the maximum absolute value a given the vector's elements.
- `MinAbsIndex` calculates the index of the minimum absolute value of a sparse vector's elements.
- `Dot` calculates the dot product of a sparse vector and a dense vector.

For example:

Code Example – C# sparse vector

```
double sumOfAbsValues = MatrixFunctions.AbsSum( sv );
double maxAbsValueIndex = MatrixFunctions.MaxAbsIndex( sv );

var w = new DoubleVector( 66, 1.2 );
double dot = MatrixFunctions.Dot( w, sv );
```

Code Example – VB sparse vector

```
Dim SumOfAbsValues As Double = MatrixFunctions.AbsSum(SV)
Dim MaxAbsValueIndex As Integer = MatrixFunctions.MaxAbsIndex(SV)

Dim W As New DoubleVector(66, 1.2)
Dim Dot As Double = MatrixFunctions.Dot(W, SV)
```

Creating Dense Vectors from Sparse Vectors

The `MatrixFunctions.Scatter()` method scatters the elements of a compressed sparse vector to a full storage vector. For example, this code constructs a dense vector from a sparse vector by specifying the length of the dense vector and scattering the nonzero values from the sparse vector into the dense vector:

Code Example – C# sparse vector

```
DoubleVector v = MatrixFunctions.Scatter( sv, 20 );
```

Code Example – VB sparse vector

```
Dim Y As DoubleVector = MatrixFunctions.Scatter(SV, 20)
```

19.2 Sparse Matrices

NMath provides the classes shown in Table 15 for storing general sparse matrices.

Table 15 – General sparse matrix classes

Class	Description
FloatCsrSparseMatrix **DoubleCsrSparseMatrix**	Store a general sparse matrix of `float` or `double` values.
FloatSymCsrSparseMatrix **DoubleSymCsrSparseMatrix**	Extend basic CSR sparse matrix class for matrices symmetric about the diagonal.
FloatComplexCsrSparseMatrix **DoubleComplexCsrSparseMatrix**	Store a general sparse matrix of **FloatComplex** or **DoubleComplex** values.
FloatHermCsrSparseMatrix **DoubleHermCsrSparseMatrix**	Extend basic complex CSR spare matrix class for matrices which satisfy $A = \overline{A}^T$, where $\overline{A}^T$ denotes the conjugate transpose of A.

Only the non-zero elements are stored.

Storage Format

Class **SparseMatrixData** stores general sparse matrix data, and is parameterized on both the storage format used, and the type, *T,* of values stored in the vector. Storage formats implement the **ISparseMatrixStorage** interface. **NMath** currently provides only one implementation—class **CompressedSparseRow**, which stores sparse matrix data in *compressed row format*.

The compressed row storage format (CSR) makes no assumptions about the sparsity structure of the matrix. CSR puts data into three arrays:

- an array of type *T* values containing the non-zero values of the matrix. The values are mapped into this array in row-major format.
- an integer column index array, where element *i* is the number of the column that contains the *i*th element of the values array.
- an integer row index array, where element *j* gives the index of the element in the values array that is the first non-zero element in the row *j*. As the row index array gives the location of the first non-zero element within a row, and the non-zero elements are stored consecutively, the number of non-zero elements in the *i*th row is equal to the difference of `rowIndex[i]` and `rowIndex[i+1]`. To have this relationship hold for the last row of the matrix, an additional entry is added to the end of `rowIndex`. Its value is equal to the number of non-zero elements. This makes the `rowIndex` array one larger that the number of rows in the matrix.

NOTE—Indexing starts at 0. Each row in compressed sparse row format must contain at least one nonzero entry.

For example, the matrix

```
     |  1 -1  0 -3  0 |
     | -2  5  0  0  0 |
 A = |  0  0  4  6  4 |
     | -4  0  2  7  0 |
     |  0  8  0  0 -5 |
```

is stored as

```
values   = (1, -1, -3, -2, 5, 4, 6, 4, -4, 2, 7, 8, -5)
columns  = ( 0, 1, 3, 0, 1, 2, 3, 4, 0, 2, 3, 1, 4 )
rowIndex = ( 0, 3, 5, 8, 11, 13 )
```

Creating Sparse Matrices

Instances of sparse matrix classes are created in a variety ways. They can be constructed from **CompressedSparseRow** objects containing the sparse data, like so:

Code Example – C# sparse matrix

```
var values = new double[4] { 1, 2, 3, 4 };
int[] columns = new int[4] { 0, 2, 1, 0 };
int[] rowIndex = new int[4] { 0, 2, 3, 4 };
int cols = 3;
```

```
var sparseData = new CompressedSparseRow<double>( values, columns,
  rowIndex, cols );

var sA = new DoubleCsrSparseMatrix( sparseData );
```

Code Example – VB sparse matrix

```
Dim Values = New Double() {1.0, 2.0, 3.0, 4.0}
Dim Columns = New Integer() {0, 2, 1, 0}
Dim RowIndex = New Integer() {0, 2, 3, 4}
Dim Cols As Integer = 3

Dim SparseData As New CompressedSparseRow(Of Double)(Values,
  Columns, RowIndex, Cols)
Dim SA As New DoubleCsrSparseMatrix(SparseData)
```

Or they can be constructed from values stored as an **IDictionary**, where row-column pair are the keys and the non-zero entries are the values:

Code Example – C# sparse matrix

```
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 );
```

Code Example – VB sparse matrix

```
Dim Values As New Dictionary(Of 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)

Dim Cols As Integer = 3
Dim SA As New DoubleCsrSparseMatrix(Values, Cols)
```

As a convenience, class **SparseMatrixBuilder** implements the interface `System.Collections.Generic.IDictionary{IntPair,T}`, providing matrix-like row and column indexing for setting and retrieving values.

Code Example – C# sparse matrix

```
var smb = new SparseMatrixBuilder<double>();
smb[0,0] = 1;
smb[0,2] = 2;
smb[1,2] = 3;
smb[2,1] = 4;
```

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

Code Example – VB sparse matrix

```
Dim SMB As New SparseMatrixBuilder(Of Double)()
SMB(0, 0) = 1
SMB(0, 2) = 2
SMB(1, 2) = 3
SMB(2, 1) = 4

Dim Cols = 3
Dim SA As New DoubleCsrSparseMatrix(SMB, Cols)
```

Lastly, sparse matrix can be generated from values in a dense matrix. This code uses the `MatrixFunctions.ToSparseMatrix()` method to create a **DoubleCsrSparseMatrix** from the non-zero elements in a **DoubleMatrix** `A`:

Code Example – C# sparse matrix

```
int maxNonzero = 500;
DoubleCsrSparseMatrix sA =
  MatrixFunctions.ToSparseMatrix( A, maxNonzero );
```

Code Example – VB sparse matrix

```
Dim MaxNonZero As Integer = 500
Dim SA As DoubleCsrSparseMatrix =
  MatrixFunctions.ToSparseMatrix(A, MaxNonZero)
```

Accessing and Modifying Sparse Matrix Values

Sparse matrix classes inherit the following properties from **SparseMatrixData**:

- `Rows` gets the number of rows in the matrix.
- `Cols` gets the number of columns in the matrix.
- `Data` gets and sets the formatted data for the matrix as an **ISparseMatrixStorage** object.

The sparse matrix classes also provide standard indexers for getting and setting individual element values:

Code Example – C# sparse matrix

```
double x = sA[4, 100];
```

Code Example – VB sparse matrix

```
Dim X As Double = SA(4, 100)
```

NOTE—Attempts to set zero elements raise a NonModifiableElementException.

Operations on Sparse Matrices

Operator `==` tests for equality of two sparse matrices, and returns `true` if both vecrtors have the same nonzero elements; otherwise, `false`. Following the convention of the .NET Framework, if both objects are `null`, they test equal. Operator `!=` returns the logical negation of `==`. The `Equals()` member function also tests for equality.

NMath provides overloaded arithmetic operators for general sparse matrices with their conventional meanings for those .NET languages that support them, and equivalent named methods for those that do not. All binary operators and equivalent named methods work either with two matrices, or with a matrix and a scalar.

Code Example – C# sparse matrix

```
double a = 3.18;
var sA1 = new DoubleCsrSparseMatrix( data );
DoubleCsrSparseMatrix sA2 = a * sA1;
```

Code Example – VB sparse matrix

```
Dim A As Double = 3.18
Dim SA1 As New DoubleCsrSparseMatrix(Data)
Dim SA2 As DoubleCsrSparseMatrix = A * SA1
```

Sparse Matrix Functions

The sparse matrix classes provide the `Scale()` function to scale each element in a sparse matrix by the specified value. **MatrixFunctions** also provides a variety of functions that take general sparse matrices as arguments:

- `Product()` computes the inner product of two sparse matrices, and returns the result as a sparse matrix.
- `DenseProduct()` computes the inner product of two sparse matrices, and returns the result as a dense matrix.
- `TransposeProduct()` computes the transpose inner product of two sparse matrices, and returns the result as a sparse matrix.
- `DenseTransposeProduct()` computes the transpose inner product of two sparse matrices, and returns the result as a dense matrix.

For instance, if `sA` and `sB` are **DoubleCsrSparseMatrix** objects:

Code Example – C# sparse matrix

```
DoubleCsrSparseMatrix sC = MatrixFunctions.Product( sA, sB )
```

Code Example – VB sparse matrix

```
Dim SC As DoubleCsrSparseMatrix = MatrixFunctions.Product(SA, SB)
```

Creating Dense Matrices from Sparse Matrices

The `MatrixFunctions.ToDenseMatrix()` method copies the elements of a compressed sparse matrix to a full storage matrix. For example, this code creates a new dense matrix from **DoubleCsrSparseMatrix** `sA`:

Code Example – C# sparse matrix

```
DoubleMatrix A = MatrixFunctions.ToDenseMatrix( sA );
```

Code Example – VB sparse matrix

```
Dim A As DoubleMatrix = MatrixFunctions.ToDenseMatrix(SA)
```

19.3 Sparse Matrix Factorizations

NMath 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.

Factorization Classes

The factorization classes associated with each general sparse matrix type are shown in Table 16.

Table 16 – NMath general sparse matrix factorization classes

Matrix Classes	Factorization Classes
FloatCsrSparseMatrix	**FloatSparseFact**
FloatSymCsrSparseMatrix	**FloatSparseSymFact** **FloatSparseSymPDFact**
FloatComplexCsrSparseMatrix	**FloatComplexSparseFact**
FloatHermCsrSparseMatrix	**FloatSparseHermFact** **FloatSparseHermPDFact**

Table 16 – NMath general sparse matrix factorization classes

Matrix Classes	Factorization Classes
DoubleCsrSparseMatrix	**DoubleSparseFact**
DoubleSymCsrSparseMatrix	**DoubleSparseSymFact** **DoubleSparseSymPDFact**
DoubleComplexCsrSparseMatrix	**DoubleComplexSparseFact**
DoubleHermCsrSparseMatrix	**DoubleSparseHermFact** **DoubleSparseHermPDFact**

Note that there are two factorization classes for symmetric and Hermitian types: one for indefinite matrices, and one for *positive definite* (PD) matrices.

SparseMatrixFact is the base class for sparse matrix factorizations, and is parameterized on the type, *T*, of values stored in the vector.

Creating Factorizations

You can create an instance of a factorization class by supplying the constructor with a matrix to factor. The following code creates a symmetric sparse matrix from the given data, then factors the matrix:

Code Example – C# sparse matrix factorization

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

Code Example – VB sparse matrix factorization

```
Dim SA As New DoubleSymCsrSparseMatrix(SparseData)
Dim Fact As New DoubleSparseSymFact(SA)
```

You can also use an existing instance to factor other matrices with the provided `Factor()` method. Thus, if `sB` is another **DoubleSymCsrSparseMatrix**:

Code Example – C# sparse matrix factorization

```
fact.Factor( sB );
```

Code Example – VB sparse matrix factorization

```
Fact.Factor(SB)
```

The read-only `ErrorStatus` property gets an `Error` enumerated value. For example:

Code Example – C# sparse matrix factorization

```
if ( fact.ErrorStatus == DoubleSparseSymFact.Error.NoError )
{
   // Do something here...
}
```

Code Example – VB sparse matrix factorization

```
If Fact.ErrorStatus = DoubleSparseSymFact.Error.NoError Then
  ' Do something here...
End If
```

Using Factorizations

Once a factorization is constructed from a matrix, it can be used to solve for different right hand sides. For instance, this code solves for one right-hand side:

Code Example – C# sparse matrix factorization

```
var b = new DoubleVector( 8, 1 );
DoubleVector x = fact.Solve( b );
```

Code Example – VB sparse matrix factorization

```
Dim B As New DoubleVector(8, 1)
Dim X As DoubleVector = Fact.Solve(B)
```

Similarly, you can use the `Solve()` method to solve for multiple right-hand sides. This code solves for 3 right-hand sides:

Code Example – C# sparse matrix factorization

```
int nrhs = 3;
var B = new DoubleMatrix( 8, nrhs, new RandGenBeta() );
DoubleMatrix X = fact.Solve( B );
```

Code Example – sparse matrix factorization

```
Dim NRHS As Integer = 3
Dim B As New DoubleMatrix(8, NRHS, New RandGenBeta())
Dim X As DoubleMatrix = Fact.Solve(B)
```

The right-hand sides are the columns of matrix `B`, and the corresponding solutions are the columns of matrix `X`.

NOTE—Be sure to check the ErrorStatus property on the factorization before calling Solve() to confirm that the factorization is valid.

CHAPTER 20.
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, and *Cholesky factorization* for symmetric and Hermitian positive definite 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**.

This chapter describes the **NMath** structured sparse matrix factorization classes, and how to construct and use them.

20.1 Factorization Classes

The factorization classes associated with each matrix type are shown in Table 17.

Table 17 – NMath factorization classes

Matrix Classes	Factorization Classes
<Type>SymmetricMatrix	**<Type>SymFact** **<Type>SymPDFact**
<Type>HermitianMatrix	**<Type>HermitianFact** **<Type>HermitianPDFact**
<Type>BandMatrix	**<Type>BandFact**
<Type>TriDiagMatrix	**<Type>TriDiagFact** **<Type>SymPDTriDiagFact** **<Type>HermPDTriDiagFact**
<Type>SymBandMatrix	**<Type>SymPDBandFact**
<Type>HermitianBandMatrix	**<Type>HermitianPDBandFact**

Note that lower and upper triangular types do not have factorization classes; these types are typically the result of factoring other matrices (for example, into the product of a lower and upper triangular matrix). Static methods for solving for different right-hand sides, and computing inverses, determinants, and condition numbers, are provided on class **MatrixFunctions** for triangular types.

Note also that **NMath** provides two factorization classes for symmetric and Hermitian types: one for indefinite matrices, and one for *positive definite* (PD) matrices. A symmetric matrix A is positive definite if there exists a nonsingular matrix B such that:

$$A = B^T B$$

where B^T is the transpose of B. A Hermitian matrix is positive definitive if there exists a nonsingular matrix B such that:

$$A = \overline{B}^T B$$

where $\overline{B}^T$ is the conjugate transpose of B. Positive definite matrices arise frequently in statistical applications.

If you don't know whether a particular symmetric or Hermitian matrix is positive definite, the easiest way to find out in **NMath** is to attempt to factor the matrix using the associated PD factorization class (Section 20.2). The read-only property `IsPositiveDefinite` returns `true` if the given matrix is positive definite and the factorization can be used to solve equations, compute determinants, inverses, and so on.

20.2 Creating Factorizations

You can create an instance of a factorization class by supplying the constructor with a matrix to factor. This code creates a 12 x 12 **FloatBandMatrix**, with upper bandwidth of 1 and lower bandwidth of 2 and values generated randomly from the interval -1 to 1, then factors the matrix using the **FloatBandFact** class constructor:

Code Example – C# matrix factorization

```
int rows = 12, cols = 12, ub = 1, lb = 2;
FloatVector data =
   new FloatVector( cols*(ub+lb+1), new RandGenUniform(-1, 1) );
FloatBandMatrix A =
   new FloatBandMatrix( data, rows, cols, lb, ub );

var F = new FloatBandFact( A );
```

Code Example – VB matrix factorization

```
Dim Rows As Integer = 12
Dim Cols As Integer = 12
Dim UB As Integer = 1
Dim LB As Integer = 2
Dim Data As New FloatVector(Cols * (UB + LB + 1),
  New RandGenUniform(-1.0, 1.0))
Dim A As New FloatBandMatrix(Data, Rows, Cols, LB, UB)

Dim F As New FloatBandFact(A)
```

You can also use an existing instance to factor other matrices with the provided `Factor()` method. Thus, if `B` is another **FloatBandMatrix**:

Code Example – C# matrix factorization

```
F.Factor( B );
```

Code Example – VB matrix factorization

```
F.Factor(B)
```

The read-only `IsGood` property gets a boolean value that is `true` if the matrix factorization succeeded and the factorization may be used to solve equations, compute determinants, inverses, and so on. Otherwise, it returns `false`. For example:

Code Example – C# matrix factorization

```
if ( F.IsGood )
{
   // Do something here...
}
```

Code Example – VB matrix factorization

```
If F.IsGood Then
  ' Do something here...
End If
```

Other read-only properties provide information about the matrix used to construct an factorization:

- `Cols` gets the number of columns of the factored matrix.
- `Rows` gets the number of rows of the factored matrix.
- On indefinite factorization classes, `IsSingular` returns `true` if the matrix was singular; otherwise, `false`.
- On positive definite factorization classes, `IsPositiveDefinite` returns `true` if the matrix was positive definite; otherwise, `false`.

20.3 Using Factorizations

Once a factorization is constructed from a matrix (see Section 20.2), it can be reused to solve for different right hand sides, and to compute inverses, determinants, and condition numbers. Static methods are also provided on **MatrixFunctions** to perform these functions without having to explicitly construct a factorization object.

Solving for Right-Hand Sides

You can use a factorization to solve for right-hand sides using the `Solve()` method. For instance, this code solves for one right-hand side:

Code Example – C# matrix factorization

```
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 );
```

Code Example – VB matrix factorization

```
Dim GenMat As 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 ]")
Dim A As New DoubleSymmetricMatrix(GenMat)
Dim F As New DoubleSymPDFact(A)
Dim V As New DoubleVector(A.Order, New RandGenUniform(-1.0, 1.0))
Dim X As DoubleVector = F.Solve(V)
```

The returned vector `x` is the solution to the linear system `Ax = v`. Note that the length of vector `v` must be equal to the number of columns in the factored matrix `A` or a **MismatchedSizeException** is thrown.

To do the same thing without explicitly constructing a factorization object, you could do this:

Code Example – C# matrix factorization

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

Code Example – VB matrix factorization

```
Dim X As DoubleVector = 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:

Code Example – C# matrix factorization

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

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

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

Code Example – VB matrix factorization

```
Dim Rows As Integer = 8
Dim Cols As Integer = 8
Dim Data As New DoubleComplexVector(Cols * 3,
  New RandGenUniform(-1.0, 1.0))
Dim A As New DoubleComplexTriDiagMatrix(Data, Rows, Cols)
Dim F As New DoubleComplexTriDiagFact(A)

Dim B As New DoubleComplexMatrix(A.Cols, 10,
  New RandGenUniform(-1.0, 1.0))

Dim X As DoubleComplexMatrix = 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 columns as the factored matrix `A`.

To do the same thing without explicitly constructing a factorization object, you could do this:

Code Example – C# matrix factorization

```
DoubleComplexMatrix X = MatrixFunctions.Solve( A, B );
```

Code Example – VB matrix factorization

```
Dim X As DoubleComplexMatrix = MatrixFunctions.Solve(A, B)
```

Computing Inverses, Determinants, and Condition Numbers

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

Code Example – C# matrix factorization

```
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();
```

Code Example – VB matrix factorization

```
Dim Rows As Integer = 8
Dim Cols As Integer = 8
Dim Lehmer As New FloatComplexMatrix(Rows, Cols)
For I As Integer = 0 To Rows - 1
  For J As Integer = 0 To Cols - 1
    If J >= I Then
      Lehmer(I, J) = CType(I + 1, Single) / CType(J + 1, Single)
    End If
  Next
Next

Dim A As New FloatHermitianMatrix(Lehmer)

Dim F As New FloatHermitianPDFact(A)
Dim AInv As FloatHermitianMatrix = F.Inverse()
Dim Det As FloatComplex = F.Determinant()
```

The `ConditionNumber()` method computes an estimate of the condition number in the one-norm. The condition number of a matrix `A` is:

```
kappa = ||A|| ||AInv||
```

where `AInv` is the inverse of the matrix `A`. For instance:

Code Example – C# matrix factorization

```
DoubleMatrix genMat =
  new DoubleMatrix( 25, 25, new RandGenUniform( 0, 100 ) );
var A = new DoubleSymmetricMatrix( genMat );

var F = new DoubleSymFact( A );
double cond = F.ConditionNumber();
```

Code Example – VB matrix factorization

```
Dim GenMat As New DoubleMatrix(25, 25,
  New RandGenUniform(0.0, 100.0))
Dim A As New DoubleSymmetricMatrix(GenMat)

Dim F As New DoubleSymFact(A)
Dim Cond As Double = F.ConditionNumber()
```

NOTE—The ConditionNumber() method returns the reciprocal of the condition number, rho, where rho = 1/kappa.

Banded and tridiagonal factorization classes also provide an overload of the `ConditionNumber()` method that accepts a value from the `NormType` enumeration for specifying the matrix norm.

Thus, this code estimates the condition number in the infinity-norm:

Code Example – C# matrix factorization

```
int rows = 4, cols = 4;
FloatVector data =
   new FloatVector( cols*3, new RandGenUniform(-1, 1) );
var A = new FloatTriDiagMatrix( data, rows, cols );

var F = new FloatTriDiagFact( A );
float cond = F.ConditionNumber( NormType.InfinityNorm );
```

Code Example – VB matrix factorization

```
Dim Rows As Integer = 4
Dim Cols As Integer = 4
Dim Data As New FloatVector(Cols * 3,
  New RandGenUniform(-1.0, 1.0))
Dim A As New FloatTriDiagMatrix(Data, Rows, Cols)

Dim F As New FloatTriDiagFact(A)
Dim Cond As Single = F.ConditionNumber(NormType.InfinityNorm)
```

Inverses, determinants, and condition numbers can also be computed without explicitly constructing a factorization object by using static methods on **MatrixFunctions**. For instance:

Code Example – C# matrix factorization

```
DoubleMatrix genMat =
  new DoubleMatrix( 12, 12, new RandGenUniform( -1, 1 ) );
var A = new DoubleSymmetricMatrix( genMat );

DoubleSymmetricMatrix AInv = MatrixFunctions.Inverse( A );
double det = MatrixFunctions.Determinant( A );
double cond = MatrixFunctions.ConditionNumber( A );
```

Code Example – VB matrix factorization

```
Dim GenMat As New DoubleMatrix(12, 12,
  New RandGenUniform(-1.0, 1.0))
Dim A As New DoubleSymmetricMatrix(GenMat)

Dim AInv As DoubleSymmetricMatrix = MatrixFunctions.Inverse(A)
Dim Det As Double = MatrixFunctions.Determinant(A)
Dim Cond As Double = MatrixFunctions.ConditionNumber(A)
```

CHAPTER 21. 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 $y = x_0 + x_1 a_1 + \ldots + x_n a_n$. The 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||`.

NMath provides classes for:

- ordinary least squares (OLS)
- weighted least squares (WLS)
- iteratively reweighted least squares (IRLS)

This chapter describes the **NMath** least square classes, and how to construct and use them.

21.1 Ordinary Least Squares Methods

NMath includes least squares classes 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 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.

21.2 Creating Ordinary Least Squares Objects

NMath provides ordinary least squares classes for four datatypes: single- and double-precision floating point numbers, and single- and double-precision complex numbers. The classnames are shown in Table 18.

Table 18 – Ordinary least squares classes

Least Squares Method	Classes
Cholesky	**FloatCholeskyLeastSq** **DoubleCholeskyLeastSq** **FloatComplexCholeskyLeastSq** **DoubleComplexCholeskyLeastSq**

Table 18 – Ordinary least squares classes

Least Squares Method	Classes
QR Decomposition	**FloatQRLeastSq** **DoubleQRLeastSq** **FloatComplexQRLeastSq** **DoubleComplexQRLeastSq**
SVD	**FloatSVDLeastSq** **DoubleSVDLeastSq** **FloatComplexSVDLeastSq** **DoubleComplexSVDLeastSq**

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**:

Code Example – C# least squares

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

Code Example – VB least squares

```
Dim A As New FloatMatrix("4x2[ 1 0 0 1 0 0 0 0]")
Dim LSQ As New FloatCholeskyLeastSq(A)
```

QR and SVD least squares classes also provide constructor overloads that accept a *tolerance* value. The specified tolerance is used in computing the numerical rank of the matrix. For example, if `A = QR` is the QR factorization of a matrix `A`, then elements on the main diagonal of `R` are considered to be zero if their absolute value is less than or equal to the tolerance. Similarly, in singular value decomposition, all singular values of the matrix `A` less than the tolerance are set to zero. Thus, this code sets all singular values less than 10^{-13} to zero:

Code Example – C# least squares

```
DoubleMatrix A = new DoubleMatrix( "4x2[ 1 0  0 1  0 0  0 0 ]" );
DoubleSVDLeastSq lsq = new DoubleSVDLeastSq( A, 1e-13 );
```

Code Example – VB least squares

```
Dim A As New DoubleMatrix( "4x2[1 0 0 1 0 0 0 0 ]" )
Dim LSQ As New DoubleSVDLeastSq( A, "1e-13" )
```

21.3 Using Ordinary Least Squares Objects

Once a least squares object has been constructed from a matrix (Section 21.2), it may be used to solve least squares problems, if the factorization or decomposition was successful.

Testing for Goodness

Read-only properties are provided for determining whether the decomposition method was successful. The SVD least squares classes provide a `Fail` property that returns `true` if the SVD algorithm failed to converge.

Other methods are guaranteed to complete, but the resultant object may still be unusable for solving least squares problems, if for example the original matrix `A` was not of full rank. All least squares classes therefore provide an `IsGood` property that returns `true` if the method succeeded and the decomposition can be used to solve least squares problems.

Solving 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:

Code Example – C# least squares

```
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 );
}
```

Code Example – VB least squares

```
Dim Rows As Integer = 6
Dim Cols As Integer = 3
Dim RNG As New RandGenUniform(-2, 2)

Dim A As DoubleMatrix = GenerateData(Rows, Cols, RNG)
Dim LSQ As New DoubleCholeskyLeastSq(A)

Dim Y As New DoubleVector(Rows, RNG)
```

```
If LSQ.IsGood Then
  Dim X As DoubleVector = LSQ.Solve(Y)
End If
```

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.

Retrieving Information About the Original Matrix

Read-only properties are also provided for retrieving information about the original matrix `A`:

- `Rows` gets the number of rows.
- `Cols` gets the number of columns.
- `Rank` (QR and SVD only) gets the numerical rank.

For example:

Code Example – C# least squares

```
var A = new DoubleComplexMatrix(
  "4x2[ (1,0) (0,0)  (0,0) (1,0)  (0,0) (0,0)  (0,0) (0,0) ]" );
var lsq = new DoubleComplexQRLeastSq( A );
int rank = lsq.Rank;
```

Code Example – VB least squares

```
Dim A As New DoubleComplexMatrix(
  "4x2[ (1,0) (0,0)  (0,0) (1,0)  (0,0) (0,0)  (0,0) (0,0) ]")
Dim LSQ As New DoubleComplexQRLeastSq(A)
Dim Rank As Integer = LSQ.Rank
```

21.4 Weighted Least Squares

NMath provides class **DoubleCOWeightedLeastSq** for solving weighted least squares (WLS) problems. WLS can modulate the importance of each observation in the final solution to correct for violations of the homoscedasticity assumption in ordinary least squares, to give less weight to outliers, or to give less weight to observations thought to be less reliable.

DoubleCOWeightedLeastSq uses a complete orthogonal decomposition technique.[2] The computed solution minimizes the 2-norm of the weighted residual vector

$$\left\| \frac{1}{\sqrt{D}}(Ax - y) \right\|$$

where *D* is a diagonal weight matrix whose diagonal consists of the weights.

Prerequisites on the matrix *A* are that it has more rows than columns, and is of full rank. Note that the algorithm satisfies an accuracy bound that is not affected by ill conditioning in the weight matrix *D*.

Instances of **DoubleCOWeightedLeastSq** are constructed from a matrix of observations and a vector of weights. For example:

Code Example – C# weighted least squares

```
var A = new DoubleMatrix( "5x2[1 2  1 3  1 6  1 10  1 7]" );
var weights = new DoubleVector( A.Rows, .2, .2 );
var wls = new DoubleCOWeightedLeastSq( A, weights );
```

Code Example – VB weighted least squares

```
Dim A As New DoubleMatrix("5x2[1 2  1 3  1 6  1 10  1 7]")
Dim Weights As New DoubleVector(A.Rows, 0.2, 0.2)
Dim WLS As New DoubleCOWeightedLeastSq(A, Weights)
```

In this case, the weights are arbitrary—observations are simply given increasingly higher weights.

DoubleCOWeightedLeastSq provides a `Solve()` method that accepts a vector `y`, and computes the solution:

Code Example – C# weighted least squares

```
var y = new DoubleVector( "[3 6 8 10 11]" );
var solution = wls.Solve( y );
```

Code Example – VB weighted least squares

```
Dim Y As New DoubleVector("[3 6 8 10 11]")
Dim Solution As DoubleVector = WLS.Solve(Y)
```

Other properties and methods on **DoubleCOWeightedLeastSq** include:

- Property `A` gets the original matrix of observations.

[2]Patricia D. Hough and Stephen A. Vavasis, "Complete Orthogonal Decomposition For Weighted Least Squares", *SIAM J. Matrix Anal. Appl.* 18, no. 2 (April 1997): 369-392

- `ResidualVector()` returns the residual vector $Ax - y$.
- `ResidualNormSqr()` computes the 2-norm squared of the residual vector.
- `Factor()` factors other matrices.
- `Reweight()` updates the weights.

NMath provides a selection of weighting functions for use in iteratively reweighted least squares (IRLS; Section 21.5). These functions can also be used to create weights for WLS. Typical weighting functions used in IRLS are a function of the adjusted residuals from the previous iteration. For example, this code computes an ordinary least squares solution, then uses the resulting residuals to solve the same problem using WLS, downweighting the outliers:

Code Example – C# weighted least squares

```
// compute ordinary least squares solution
var ols = new DoubleQRLeastSq( A );
DoubleVector olsSolution = ols.Solve( y );
DoubleVector olsResiduals = ols.ResidualVector( y );

// compute weights from residuals using bisquare function
var weights = new DoubleVector( residuals.Length );
IDoubleLeastSqWeightingFunction weightingFunction =
  new DoubleBisquareWeightingFunction( A );
weightingFunction.GetWeights( olsResiduals, ref weights );

// compute weighted least squares solution
var wls = new DoubleCOWeightedLeastSq( A, weights );
DoubleVector wlsSolution = wls.Solve( y );
```

Code Example – VB weighted least squares

```
' compute ordinary least squares solution
Dim OLS As New DoubleQRLeastSq(A)
Dim OLSSolution As DoubleVector = OLS.Solve(Y)
Dim OLSResiduals As DoubleVector = OLS.ResidualVector(Y)

' compute weights from residuals using bisquare function
Dim Weights As New DoubleVector(Residuals.Length)
Dim WeightingFunction As IDoubleLeastSqWeightingFunction =
  New DoubleBisquareWeightingFunction(A)
WeightingFunction.GetWeights(OLSResiduals, Weights)

' compute weighted least squares solution
Dim WLS As New DoubleCOWeightedLeastSq(A, Weights)
Dim WLSSolution As DoubleVector = WLS.Solve(Y)
```

21.5 Iteratively Reweighted Least Squares

Iteratively reweighted least squares (IRLS) is an iterative minimization technique in which each step involves solving a standard weighted least squares (Section 21.4). New weights are computed at each iteration by applying a *weighting function* to the current residuals. The process terminates when either a specified *convergence function* returns `true`—typically when either the residuals or the solution are unchanged on successive iterations—or when a specified maximum number of iterations is reached.

NMath provides class **DoubleIterativelyReweightedLeastSq** for solving IRLS problems. The default weighting function is a bisquare weighting function, and the default convergence function returns `true` when the solutions do not change, within tolerance, on successive iterations. For instance:

Code Example – C# iteratively reweighted least squares

```
var irls = new DoubleIterativelyReweightedLeastSq();
```

Code Example – VB iteratively reweighted least squares

```
Dim IRLS As New DoubleIterativelyReweightedLeastSq()
```

Additional constructors enable you to specify the tolerance, the maximum number of iterations, and a weighting function (see below). Properties `Tolerance`, `MaxIterations`, and `WeightsFunction` are also provided for modifying these values post-construction.

The `Solve()` method solves the least squares problem `Ax = y` for x using the IRLS method:

Code Example – C# iteratively reweighted least squares

```
DoubleVector x = irls.Solve( A, y );
```

Code Example – VB iteratively reweighted least squares

```
Dim X As DoubleVector = IRLS.Solve(A, Y)
```

Property `Residuals` gets the residual vector from the most recent computation. `Iterations` gets the number of iterations performed. For instance:

Code Example – C# iteratively reweighted least squares

```
if ( irls.MaxIterationsMet ) {
  Console.WriteLine(
    "The algorithm failed to converge in {0} iterations.",
    irls.MaxIterations
  );
}
else {
```

```
  Console.WriteLine(
    "Algorithm converged in {0} iterations.", irls.Iterations );
}
```

Code Example – VB iteratively reweighted least squares

```
If IRLS.MaxIterationsMet Then
  Console.WriteLine(
    "The algorithm failed to converge in {0} iterations.",
    IRLS.MaxIterations)
Else
  Console.WriteLine(
    "Algorithm converged in {0} iterations.", IRLS.Iterations)
End If
```

Convergence Functions

The convergence function is called at the end of each iteration. If the function returns `true`, the algorithm is terminated; otherwise, iteration continues.

Convergence functions are specified as instances of delegate `DoubleIterativelyReweightedLeastSq.ToleranceMetFunction`:

Code Example – C# iteratively reweighted least squares

```
public delegate bool ToleranceMetFunction(
  double tolerance,
  DoubleVector lastSolution,
  DoubleVector currentSolution,
  DoubleVector lastResiduals,
  DoubleVector currentResiduals);
```

Code Example – VB iteratively reweighted least squares

```
Delegate Function ToleranceMetFunction(
  Tolerance As Double,
  LastSolution As DoubleVector,
  CurrentSolution As DoubleVector,
  LastResiduals As DoubleVector,
  CurrentResiduals As DoubleVector) As Boolean
```

For example, this code encapsulates the user-defined function `MyFunction` as a `DoubleIterativelyReweightedLeastSq.ToleranceMetFunction` delegate:

Code Example – C# iteratively reweighted least squares

```
public static bool MyFunction(
  double tolerance,
  DoubleVector lastSolution,
  DoubleVector currentSolution,
  DoubleVector lastResiduals,
  DoubleVector currentResiduals )
{
  double a =
    NMathFunctions.MaxAbsValue( currentResiduals - lastResiduals );
  double b = NMathFunctions.MaxAbsValue( currentResiduals );
  return ( a/b ) < tolerance;
}

public static
 DoubleIterativelyReweightedLeastSq.ToleranceMetFunction
   residualsUnchanged =
    new DoubleIterativelyReweightedLeastSq.ToleranceMetFunction(
      MyFunction );
```

Code Example – VB iteratively reweighted least squares

```
public Shared Function MyFunction( Tolerance As Double, ,
  LastSolution As DoubleVector, CurrentSolution As DoubleVector,
  LastResiduals As DoubleVector, CurrentResiduals As DoubleVector)
As Boolean
  Dim A As Double =
    NMathFunctions.MaxAbsValue(CurrentResiduals - LastResiduals)
  Dim B As Double = NMathFunctions.MaxAbsValue(CurrentResiduals)
  Return (A / B) < Tolerance
End Function
```

Property `ConvergenceFunction` can be used to get and set the convergence function on a **DoubleIterativelyReweightedLeastSq** instance:

Code Example – C# iteratively reweighted least squares

```
irls.ConvergenceFunction = residualsUnchanged;
```

Code Example – VB iteratively reweighted least squares

```
IRLS.ConvergenceFunction = ResidualsUnchanged
```

Weighting Functions

NMath provides a selection of least squares weighting functions. Typical weighting functions used in IRLS are a function of the adjusted residuals from the previous iteration. Abstract base class **DoubleLeastSqWeightingFunction** provides method `AdjustedResiduals()` for calculating the adjusted residuals according to the following formula:

$$r' = \frac{r}{(ts\sqrt{1-h})}$$

where

- r' is the adjusted residuals.
- r is the actual residuals.
- t is a *tuning constant* by which the residuals are divided before computing weights. Decreasing the tuning constant increases the downweight assigned to large residuals, and increasing the tuning constant decreases the downweight assigned to large residuals.
- h is the vector of leverage values that adjust the residuals by downweighting high-leverage data points, which have a large effect on the least squares fit. The leverage values are the main diagonal of the *hat* matrix

  $$H = A(A'A)^{-1}A'$$

- s is an estimate of the standard deviation of the error term given by

  $$s = \frac{MAD}{0.6745}$$

  where `MAD` is the median absolute deviation of the residuals from their median. The constant `0.6745` makes the estimate unbiased for the normal distribution.

DoubleLeastSqWeightingFunction also implements the **IDoubleLeastSqWeightingFunction** interface, which provides methods `Initialize()` for performing any needed initialization given the matrix A, and `GetWeights()` for computing weights from a given vector of residuals.

Two concrete implementations of **DoubleLeastSqWeightingFunction** are provided:

- **DoubleBisquareWeightingFunction** applies the *bisquare* weighting formula to a set of adjusted residuals:

$$w_i = \begin{cases} (1-(|r_i|)^2)^2 & |r_i| < 1 \\ 0 & |r_i| \geq 1 \end{cases}$$

where r is the adjusted residuals computed by the `AdjustedResidual()` function on the base class **DoubleLeastSqWeightingFunction**.

- **DoubleFairWeightingFunction** applies the *fair weighting* formula to a set of adjusted residuals:

$$w_i = \frac{1}{(1 + |r_i|)}$$

where r is the adjusted residuals computed by the `AdjustedResidual()` function on the base class **DoubleLeastSqWeightingFunction**.

The default weighting function used is the bisquare weighting function. This code constructs a **DoubleIterativelyReweightedLeastSq** instance using the fair weighting function:

Code Example – C# iteratively reweighted least squares

```
var weightingFunction = DoubleFairWeightingFunction();
var irls = new DoubleIterativelyReweightedLeastSq(
weightingFunction );
```

Code Example – VB iteratively reweighted least squares

```
Dim WeightingFunction As IDoubleLeastSqWeightingFunction =
  New DoubleFairWeightingFunction()
Dim IRLS As New
  DoubleIterativelyReweightedLeastSq(WeightingFunction)
```

The weighting function can also be changed on an existing **DoubleIterativelyReweightedLeastSq** object using the `WeightsFunction` property:

Code Example – C# iteratively reweighted least squares

```
irls.WeightsFunction = new DoubleFairWeightingFunction();
```

Code Example – C# iteratively reweighted least squares

```
IRLS.WeightsFunction = New DoubleFairWeightingFunction()
```

CHAPTER 22.
DECOMPOSITIONS

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

This chapter describes the **NMath** decomposition and decomposition server classes, and how to construct and use them.

22.1 QR Decompositions

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).

Creating QR Decompositions

NMath provides QR decomposition classes for four datatypes: single- and double-precision floating point numbers, and single- and double-precision complex numbers. The classnames are **FloatQRDecomp**, **DoubleQRDecomp**, **FloatComplexQRDecomp**, and **DoubleComplexQRDecomp**.

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

Code Example – C# QR decomposition

```
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 );
```

Code Example – VB QR decomposition

```
Dim A As New FloatMatrix(
  "5x3 [ 1 2 3  4 5 6  7 8 9  0 1 2  3 4 5 ]")
Dim QR As New FloatQRDecomp(A)
```

By default, pivoting is done so that the entries along the diagonal of `R` are non-increasing. For greater control, **NMath** provides QR decomposition server classes that create QR decomposition objects with non-default decomposition parameters. The classnames are **FloatQRDecompServer**, **DoubleQRDecompServer**, **FloatComplexQRDecompServer**, and **DoubleComplexQRDecompServer**.

The QR decomposition server classes all have the same interface:

- The `Pivoting` property sets whether or not pivoting is performed. By default, pivoting is `true`.
- The `SetInitialColumn()` method moves a given column to the beginning of `AP` before the computation, and fixes it in place during the computation.
- The `SetFreeColumn()` method allows a given column to be interchanged during the computation with any other free column. By default, all columns are free.
- The `GetDecomp()` method takes a matrix and returns a decomposition object using the current pivoting parameters.

For example, this code uses a **DoubleComplexQRDecompServer** to turn off pivoting:

Code Example – C# QR decomposition

```
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 );
```

Code Example – VB QR decomposition

```
Dim QRS As New DoubleComplexQRDecompServer()
QRS.Pivoting = False

Dim Rows As Integer = 10
Dim Cols As Integer = 3
Dim A As New DoubleComplexMatrix(Rows, Cols,
  New RandGenUniform(-1.0, 1.0))
Dim QR As DoubleComplexQRDecomp = QRS.GetDecomp(A)
```

This code moves column 7 to the beginning of AP before the computation, and fixes it in place during the computation:

Code Example – C# QR decomposition

```
var qrs = new DoubleQRDecompServer();
qrs.SetIntialColumn( 7 );

int rows = 20, cols = 12;
var A = new DoubleMatrix( rows, cols,
  new RandGenUniform(-1,1) );
DoubleQRDecomp qr = qrs.GetDecomp( A );
```

Code Example – VB QR decomposition

```
Dim QRS As New DoubleQRDecompServer()
QRS.SetIntialColumn(7)

Dim Rows As Integer = 20
Dim Cols As Integer = 12
Dim A As New DoubleMatrix(Rows, Cols,
  New RandGenUniform(-1.0, 1.0))
Dim QR As DoubleQRDecomp = QRS.GetDecomp(A)
```

Using QR Decompositions

Once a QR decomposition object has been constructed from a matrix, various read-only properties are provided for retrieving the elements of the decomposition, and for retrieving information about the original matrix:

- P gets the permutation matrix.
- Q gets the orthogonal matrix.
- R gets the upper trapezoidal matrix.
- Rows gets the number of rows in the original matrix A.
- Cols gets the number of columns in the original matrix A.

For example:

Code Example – C# QR decomposition

```
int rows = 10, cols = 3;
DoubleMatrix A =
  new DoubleMatrix( rows, cols, new RanGenUniform( 1, -1 ) );

var qr = new DoubleQRDecomp( A );
DoubleMatrix Q = qr.Q;
DoubleMatrix R = qr.R;
DoubleMatrix P = qr.P;
```

Code Example – VB QR decomposition

```
Dim Rows As Integer = 10
Dim Cols As Integer = 3
Dim A As New DoubleMatrix(Rows, Cols,
  New RandGenUniform(-1.0, 1.0))

Dim QR As New DoubleQRDecomp(A)
Dim Q As DoubleMatrix = QR.Q
Dim R As DoubleMatrix = QR.R
Dim P As DoubleMatrix = QR.P
```

Methods are also provided for manipulating the component matrices `P`, `Q`, or `R`:

- `Px()`, `Qx()`, and `Rx()` compute the inner product of a component matrix and a given vector.
- `PTx()`, `QTx()`, and `RTx()` compute the inner product of the transpose of a component matrix and a given vector, or conjugate transpose for complex types.
- `QM()` computes the inner product of the orthogonal matrix `Q` and a given matrix. `QTM()` uses the transpose of `Q`, or conjugate transpose for complex types.
- `RInvx()` computes the inner product of the inverse of the the upper trapezoidal matrix `R` and a given vector. `RTInvx()` uses the transpose of `R`, or conjugate transpose for complex types.
- `RDiagonal()` returns the main diagonal of the upper trapezoidal matrix `R`.

These methods are more efficient than retrieving a component matrix using the `P`, `Q`, and `R` properties and manipulating it yourself.

For example:

Code Example – C# QR decomposition

```
int rows = 12, cols = 20;
var A = new FloatComplexMatrix( rows, cols, rng );
var qr = new FloatComplexQRDecomp( A );

var x = new FloatComplexVector( qr.P.Cols, 1, 1 );
FloatComplexVector y = qr.Px( x );
```

Code Example – VB QR decomposition

```
Dim Rows As Integer = 12
Dim Cols As Integer = 20
Dim A As New FloatComplexMatrix(Rows, Cols, rng)
Dim QR As New FloatComplexQRDecomp(A)

Dim X As New FloatComplexVector(QR.P.Cols, 1, 1)
```

```
Dim Y As FloatComplexVector = QR.Px(X)
```

Reusing QR Decompositions

An existing decomposition object can be reused with another matrix using the `Factor()` method:

Code Example – C# QR decomposition

```
int rows = 10, cols = 3;
var rng = new RandGenUniform( -1, 1 );
var A = new FloatMatrix( rows, cols, rng );

var qr = new FloatQRDecomp( A );
FloatMatrix Q1 = qr.Q;
FloatMatrix R1 = qr.R;
FloatMatrix P1 = qr.P;

rows = 7;
cols = 7;
var B = new FloatMatrix( rows, cols, rng );

qr.Factor( B );
FloatMatrix Q2 = qr.Q;
FloatMatrix R2 = qr.R;
FloatMatrix P2 = qr.P;
```

Code Example – VB QR decomposition

```
Dim Rows As Integer = 10
Dim Cols As Integer = 3
Dim RNG As New RandGenUniform(-1.0, 1.0)
Dim A As New FloatMatrix(Rows, Cols, RNG)

Dim QR As New FloatQRDecomp(A)
Dim Q1 As FloatMatrix = QR.Q
Dim R1 As FloatMatrix = QR.R
Dim P1 As FloatMatrix = QR.P

Rows = 7
Cols = 7
Dim B As New FloatMatrix(Rows, Cols, RNG)

QR.Factor(B)
Dim Q2 As FloatMatrix = QR.Q
Dim R2 As FloatMatrix = QR.R
Dim P2 As FloatMatrix = QR.P
```

22.2 Singular Value Decompositions

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*.

Creating Singular Value Decompositions

NMath provides singular value decomposition classes for four datatypes: single- and double-precision floating point numbers, and single- and double-precision complex numbers. The classnames are **FloatSVDecomp**, **DoubleSVDecomp**, **FloatComplexSVDecomp**, and **DoubleComplexSVDecomp**.

Instances of the singular value decomposition classes are constructed from general matrices of the appropriate datatype. For example, this code creates a **DoubleSVDecomp** from a **DoubleMatrix**:

Code Example – C# SVD

```
DoubleMatrix A =
  new DoubleMatrix( "4 x 3 [ 1 2 3  12 -2 6  -8 9 11  5 7 -1]" );
var svd = new DoubleSVDecomp( A );
```

Code Example – VB SVD

```
Dim A As New DoubleMatrix(
  "4 x 3 [ 1 2 3  12 -2 6  -8 9 11  5 7 -1]")
Dim SVD As New DoubleSVDecomp(A)
```

By default, the reduced singular value decomposition and all singular vectors are computed. For greater control, **NMath** provides singular value decomposition server classes that create singular value decomposition objects with non-default decomposition parameters. The classnames are **FloatSVDecompServer**, **DoubleSVDecompServer**, **FloatComplexSVDecompServer**, and **DoubleComplexSVDecompServer**.

The singular value decomposition server classes all have the same interface:

- The `ComputeFull` property gets and sets whether the full or reduced singular value decomposition is computed. (If matrix `A` is square, the full and reduced singular value decompositions are the same.)

- The `ComputeLeftVectors` property gets and sets whether or not the left singular vectors are computed.
- The `ComputeRightVectors` property gets and sets whether or not the right singular vectors are computed.
- The `Tolerance` property gets and sets the tolerance for truncating all singular values. Values less than the tolerance are set to zero.
- The `GetDecomp()` method takes a matrix and returns a singular value decomposition object using the current decomposition parameters.

For example, this code uses a **FloatComplexSVDecompServer** to turn off the computation of the singular vectors:

Code Example – C# SVD

```
var svds = new FloatComplexSVDecompServer();
svds.ComputeLeftVectors = false;
svds.ComputeRightVectors = false;

int rows = 10, cols = 10;
var A = new FloatComplexMatrix( rows, cols,
  new RandGenUniform( -1, 1 ) );
FloatComplexSVDecomp svd = svds.GetDecomp( A );
```

Code Example – VB SVD

```
Dim SVDS As New FloatComplexSVDecompServer()
SVDS.ComputeLeftVectors = False
SVDS.ComputeRightVectors = False

Dim Rows As Integer = 10
Dim Cols As Integer = 10
Dim A As New FloatComplexMatrix(Rows, Cols,
  New RandGenUniform(-1.0, 1.0))
Dim SVD As FloatComplexSVDecomp = SVDS.GetDecomp(A)
```

Using Singular Value Decompositions

Once a singular value decomposition object has been constructed from a matrix, various read-only properties are provided for retrieving the elements of the decomposition, and for retrieving information about the original matrix:

- `LeftVectors` gets the matrix whose columns are the left singular vectors.
- `RightVectors` gets the matrix whose columns are the right singular vectors.
- `NumberLeftVectors` gets the number of left singular vectors.

- NumberRigthVectors gets the number of right singular vectors.
- SingularValues gets the singular values of this decomposition. The values are non-negative and arranged in decreasing order.
- Rank gets the rank of the original matrix A.
- Rows gets the number of rows in the original matrix A.
- Cols gets the number of columns in the original matrix A.
- Fail gets the status of the decomposition. The property returns true if the decomposition algorithm failed to converge; otherwise, false.

For instance:

Code Example – C# SVD

```
int rows = 5, cols = 5;
var A =
  new FloatMatrix( rows, cols, new RandGenUniform( 1, -1 ) );

var svd = new FloatSVDecomp( A );
FloatMatrix U = svd.LeftVectors;
FloatMatrix V = svd.RightVectors;
FloatVector s = svd.SingularValues;
```

Code Example – VB SVD

```
Dim Rows As Integer = 5
Dim Cols As Integer = 5
Dim A As New FloatMatrix(rows, cols,
  New RandGenUniform(-1.0, 1.0))

Dim SVD As New FloatSVDecomp(A)
Dim U As FloatMatrix = svd.LeftVectors
Dim Y As FloatMatrix = svd.RightVectors
Dim S As FloatVector = svd.SingularValues
```

Methods are also provided for retrieving individual singular vectors and singular values:

- LeftVector() returns a specified left singular vector.
- RightVector() returns a specified right singular vector.
- SingularValue() returns a specified singular value.

For example, this code returns the first singular value, which is equal to the Euclidean (L_2) norm of the matrix A:

Code Example – C# SVD

```
int rows = 12, cols = 6;
```

```
var A = new DoubleComplexMatrix( rows, cols,
  new RandGenUniform( -1, 1) );

var svd = new DoubleComplexSVDecomp( A );
double l2 = svd.SingularValue( 0 );
```

Code Example – VB SVD

```
Dim Rows As Integer = 12
Dim Cols As Integer = 6
Dim A As New DoubleComplexMatrix(Rows, Cols,
  New RandGenUniform(-1.0, 1.0))

Dim SVD As New DoubleComplexSVDecomp(A)
Dim L2 As Double = SVD.SingularValue(0)
```

Lastly, a `Truncate()` method is provided that sets all singular values less than a given tolerance to zero. Corresponding singular vectors are also removed.

NOTE—This method can change the numerical rank of the matrix A, which is equal to the number of non-zero singular values.

Code Example – C# SVD

```
var A = new DoubleMatrix(
  "5x5[1 2 3 4 5  6 7 8 9 0  1 2 3 4 5  6 7 8 9 0  1 2 3 4 5]" );

var svd = new DoubleSVDecomp( A );
int fullRank = svd.Rank;       // == 5

svd.Truncate( 1e-14 );
int deficientRank = svd.Rank;  // == 2
```

Code Example – VB SVD

```
Dim A As New DoubleMatrix(
  "5x5[1 2 3 4 5  6 7 8 9 0  1 2 3 4 5  6 7 8 9 0  1 2 3 4 5]")

Dim SVD As New DoubleSVDecomp(A)
Dim FullRank As Integer = SVD.Rank       ' == 5

SVD.Truncate("1e-14")
Dim DeficientRanks As Integer = svd.Rank  ' == 2
```

Reusing Singular Value Decompositions

An existing decomposition object can be reused with another matrix using the `Factor()` method:

Code Example – C# SVD

```
int rows = 12, cols = 6;
```

```
FloatMatrix A =
  new FloatMatrix( rows, cols, new RandGenUniform( -1, 1 ) );

var svd = new DoubleSVDecomp( A );
FloatVector svA = svd.SingularValues;

var B = new DoubleMatrix(
  "5x5[1 2 3 4 5  6 7 8 9 0  1 2 3 4 5  6 7 8 9 0  1 2 3 4 5]" );

svd.Factor( B );
FloatVector svB = svd.SingularValues;
```

Code Example – VB SVD

```
Dim Rows As Integer = 12
Dim Cols As Integer = 6
Dim A As New FloatMatrix(Rows, Cols, New RandGenUniform(-1.0, 1.0))

Dim SVD As New DoubleSVDecomp(A)
Dim SVA As FloatVector = SVD.SingularValues

Dim B As New DoubleMatrix(
  "5x5[1 2 3 4 5  6 7 8 9 0  1 2 3 4 5  6 7 8 9 0  1 2 3 4 5]")

SVD.Factor(B)
Dim SVB As FloatVector = svd.SingularValues
```

CHAPTER 23. EIGENVALUE PROBLEMS

NMath includes classes for solving symmetric, Hermitian, and nonsymmetric eigenvalue problems. The classical eigenvalue problem is defined as the solution to:

$$AV = V\Omega$$

for a matrix A, eigenvectors V, and the diagonal matrix of eigenvalues Ω. **NMath** also provides eigenvalue *server* classes that construct instances of the eigenvalue classes, allowing you greater control over how the eigenvalue decomposition is performed.

For example, class **DoubleSymEigDecomp** computes the eigenvalues and eigenvectors of a **DoubleSymmetricMatrix**. By default, this class computes both eigenvalues and eigenvectors. For more control, the associated decomposition server class, **DoubleSymEigDecompServer**, 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. A tolerance for the convergence of the algorithm may also be specified.

This chapter describes the **NMath** eigenvalue and eigenvalue server classes, and how to construct and use them.

23.1 Eigenvalue Classnames

NMath provides eigenvalue and eigenvalue server classes for the usual four datatypes (single- and double-precision floating point numbers, and single- and double-precision complex numbers), in both nonsymmetric and symmetric forms. The classnames are shown in Table 19.

Table 19 – Eigenvalue classes

Nonsymmetric	Symmetric/Hermitian
FloatEigDecomp **FloatEigDecompServer**	**FloatSymEigDecomp** **FloatSymEigDecompServer**
DoubleEigDecomp **DoubleEigDecompServer**	**DoubleSymEigDecomp** **DoubleSymEigDecompServer**
FloatComplexEigDecomp **FloatComplexEigDecompServer**	**FloatHermitianEigDecomp** **FloatHermitianEigDecompServer**
DoubleComplexEigDecomp **DoubleComplexEigDecompServer**	**DoubleHermitianEigDecomp** **DoubleHermitianEigDecompServer**

23.2 Using the Eigenvalue Classes

The **NMath** eigenvalue classes solve symmetric, Hermitian, and nonsymmetric eigenvalue problems.

Constructing Eigenvalue Objects

Instances of the eigenvalue classes are constructed from matrices of the appropriate type. For example, this code creates a **FloatSymEigDecomp** from a **FloatSymmetricMatrix**:

Code Example – C# eigenvalue decomposition

```
var A = new FloatMatrix( "4x4 [ 0 1.73205080756888 0 0
                                1.73205080756888 0 2 0
                                0 2 0 1.73205080756888
                                0 0 1.73205080756888 0 ]");
var Asym = new FloatSymmetricMatrix( A );
var eig = new FloatSymEigDecomp( Asym );
```

Code Example – VB eigenvalue decomposition

```
Dim A As New FloatMatrix("4x4 [ 0 1.73205080756888 0 0" & _
                               "1.73205080756888 0 2 0" & _
                               "0 2 0 1.73205080756888" & _
                               "0 0 1.73205080756888 0 ]")
Dim Asym As New FloatSymmetricMatrix(A)
Dim Eig As New FloatSymEigDecomp(Asym)
```

Similarly, if `A` is a **DoubleHermitianMatrix**, this code creates a **DoubleHermitianEigDecomp** object from `A`:

Code Example – C# eigenvalue decomposition

```
var eig = new DoubleHermitianEigDecomp( A );
```

Code Example – VB eigenvalue decomposition

```
Dim Eig As New DoubleHermitianEigDecomp(A)
```

Testing for Goodness

All eigenvalue classes provide an `IsGood` property that returns `true` if all the eigenvalues and eigenvectors were successfully computed:

Code Example – C# eigenvalue decomposition

```
var eig = new DoubleComplexEigDecomp( A );
if ( eig.IsGood )
{
  // Do something here...
}
```

Code Example – VB eigenvalue decomposition

```
Dim Eig As New DoubleComplexEigDecomp(A)
If Eig.IsGood Then
  ' Do something here...
End If
```

Retrieving Eigenvalues and Eigenvectors

All eigenvalue classes provide read-only properties and member functions for retrieving eigenvalues and eigenvectors.

- `NumberOfEigenValues` gets the number of eigenvalues computed.
- `EigenValues` gets the vector of computed eigenvalues.
- `EigenValue()` returns the specified eigenvalue.
- `NumberOfLeftEigenVectors` gets the number of left eigenvectors.
- `LeftEigenVectors` gets the matrix of left eigenvectors.
- `LeftEigenVector()` returns the specified left eigenvector.

- `NumberOfRightEigenVectors` gets the number of right eigenvectors.
- `RightEigenVectors` gets the matrix of right eigenvectors.
- `RightEigenVector()` returns the specified right eigenvector.

For example:

Code Example – C# eigenvalue decomposition

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

Code Example – VB eigenvalue decomposition

```
Dim Decomp As New FloatEigDecomp(A)
Console.WriteLine("Eigenvalues = {0}", Decomp.EigenValues)
Console.WriteLine("Left eigenvectors = {0}",
  Decomp.LeftEigenVectors)
Console.WriteLine("Right eigenvectors = {0}",
  Decomp.RightEigenVectors)
```

Retrieving Information About the Original Matrix

Read-only properties are also provided for retrieving information about the original matrix `A`:

- `Rows` gets the number of rows.
- `Cols` gets the number of columns.

Reusing Eigenvalue Decompositions

An existing eigenvalue object can be reused with another matrix using the `Factor()` method:

Code Example – C# eigenvalue decomposition

```
var eig = new FloatSymEigDecomp( A );
if ( eig.IsGood )
{
  // Do something here...
}
```

```
eig.Factor( B );
if ( eig.IsGood )
{
  // Do something here...
}
```

Code Example – VB eigenvalue decomposition

```
Dim Eig As New FloatSymEigDecomp(A)
If Eig.IsGood Then
  ' Do something here...
End If

Eig.Factor(B)
If Eig.IsGood Then
  ' Do something here...
End If
```

23.3 Using the Eigenvalue Server Classes

The **NMath** eigenvalue server classes construct instances of the eigenvalue classes (Section 23.2), allowing you greater control over how the eigenvalue decomposition is performed. Servers can be configured to compute eigenvalues only, or both eigenvalues and eigenvectors. In addition, servers can be configured to compute only the eigenvalues in a given range. A tolerance for the convergence of the algorithm may also be specified.

Constructing Eigenvalue Servers

Instances of the eigenvalue server classes are constructed using a default constructor, then configured as desired. For example, this code creates a default **DoubleSymEigDecompServer**:

Code Example – C# eigenvalue decomposition

```
var server = new DoubleSymEigDecompServer();
```

Code Example – VB eigenvalue decomposition

```
Dim Server As New DoubleSymEigDecompServer()
```

Configuring Eigenvalue Servers

All eigenvalue server classes provide properties and member functions for configuring the server after construction:

- `ComputeRightVectors` gets and sets a boolean value indicating whether or not right eigenvectors should be computed (`true` by default).
- `ComputeLeftVectors` gets and sets a boolean value indicating whether or not left eigenvectors should be computed (`true` by default).
- `ComputeAllEigenValues()` configures a server to compute all eigenvalues.
- `ComputeEigenValueRange()` configures a server to compute only the eigenvalues in a specified range. Only eigenvalues that are greater than the given lower bound and less than or equal to the given upper bound are computed.
- `Balance` gets and sets the balance option, using a value from the **BalanceOption** enumeration: `None`, `Permute`, `Scale`, `Both`. Balancing a matrix means permuting the rows and columns to make the matrix more nearly upper triangular, and applying a diagonal similarity transformation to make the rows and columns closer in norm and the condition numbers of the eigenvalues and eigenvectors smaller.
- `AbsTolerance` gets and sets the absolute tolerance for each eigenvalue. An approximate eigenvalue is accepted as converged when it lies in an interval `[a,b]` of width less than or equal to `AbsTolerance + epsilon * max(abs(a),abs(b))`, where `epsilon` is machine precision. If `AbsTolerance` is set less than or equal to zero then `epsilon * ||T||` is used, where `T` is the tridiagonal matrix obtained by reducing the decomposed matrix to tridiagonal form, and `||T||` is the one-norm of `T`.

NOTE—Eigenvalue ranges and tolerance are only provided for symmetric and Hermitian eigenvalue server classes. For general matrices, eigenvalues may be complex, and hence non-orderable.

For example, this code creates a default **DoubleSymEigDecompServer**, then configures the object not to compute eigenvectors, and only to compute eigenvalues within a specified range:

Code Example – C# eigenvalue decomposition

```
var server = new FloatSymEigDecompServer();
server.ComputeLeftVectors = false;
server.ComputeRightVectors = false;
server.ComputeEigenValueRange( 0, 3 );
```

Code Example – VB eigenvalue decomposition

```
Dim Server As New FloatSymEigDecompServer()
Server.ComputeLeftVectors = False
Server.ComputeRightVectors = False
Server.ComputeEigenValueRange(0, 3)
```

Creating Eigenvalue Objects from a Server

Eigenvalue server objects are used to create instances of the associated eigenvalue class, using the `Factor()` method. For instance, this code creates a **FloatEigDecomp** object from a configured **FloatEigDecompServer**:

Code Example – C# eigenvalue decomposition

```
var eigServer = new FloatEigDecompServer();
eigServer.ComputeLeftVectors = false;
eigServer.ComputeRightVectors = false;
eigServer.Balance = BalanceOption.Permute;
FloatEigDecomp decomp = eigServer.Factor( A );
```

Code Example – VB eigenvalue decomposition

```
Dim EigServer As New FloatEigDecompServer()
eigServer.ComputeLeftVectors = False
eigServer.ComputeRightVectors = False
eigServer.Balance = BalanceOption.Permute
Dim Decomp = EigServer.Factor(A)
```

PART IV - ANALYSIS

CHAPTER 24. THE ANALYSIS NAMESPACE

The `CenterSpace.NMath.Core` namespace provides the following analytical classes:

- 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), non-linear 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, fzero 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.
- Solutions to first order initial value differential equations by the Runge-Kutta method

To avoid using fully qualified names, preface your code with an appropriate namespace statement:

Code Example – C#

```
using CenterSpace.NMath.Core;
```

Code Example – VB

```
Imports CenterSpace.NMath.Core
```

CHAPTER 25. ENCAPSULATING MULTIVARIATE FUNCTIONS

The `CenterSpace.NMath.Core` namespace includes classes for encapsulating univariate functions, including base class **OneVariableFunction**, and derived types **Polynomial** and **TabulatedFunction** (Chapter 13). In addition, the **MultiVariableFunction** class encapsulates an arbitrary function of one or more variables, and works with other **NMath** classes to approximate integrals and minima.

This chapter describes how to create and manipulate **MultiVariableFunction** function objects.

25.1 Creating Multivariate Functions

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:

Code Example – C# multivariate functions

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

Code Example – VB multivariate functions

```
Function MyFunction(V As DoubleVector) As Double
  Return (NMathFunctions.Sum(V * V))
End Function
```

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

Code Example – C# multivariate functions

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

Code Example – VB multivariate functions

```
Dim D As New Func(Of DoubleVector, Double)(AddressOf MyFunction)
```

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

Code Example – C# multivariate functions

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

Code Example – VB multivariate functions

```
Dim F As New MultiVariableFunction(D)
```

A `Func<DoubleVector, double>` is also implicitly converted to a **MultiVariableFunction**. Thus:

Code Example – C# multivariate functions

```
MultiVariableFunction f = d;
```

Code Example – VB multivariate functions

```
Dim F = D
```

Class **MultiVariableFunction** provides a `Function` property that gets the encapsulated function delegate after construction.

25.2 Evaluating Multivariate Functions

The `Evaluate()` method on `MultiVariableFunction` evaluates a function at a given point. For instance, if `f` is a `MultiVariableFunction` of four variables:

Code Example – C# multivariate functions

```
var point = new DoubleVector( 0.0, 1.0, 0.0, -1.0 );
double z = f.Evaluate( point );
```

Code Example – VB multivariate functions

```
Dim Point As New DoubleVector(0.0, 1.0, 0.0, -1.0)
Dim Z As Double = F.Evaluate(Point)
```

25.3 Algebraic Manipulation of Multivariate Functions

NMath provides overloaded arithmetic operators for multivariate functions with their conventional meanings for those .NET languages that support them, and equivalent named methods for those that do not. Table 20 lists the equivalent operators and methods.

Table 20 – Arithmetic operators

Operator	Equivalent Named Method
`+`	`Add()`
`-`	`Subtract()`
`*`	`Multiply()`
`/`	`Divide()`
Unary `-`	`Negate()`

All binary operators and equivalent named methods work either with two functions, or with a function and a scalar. For example, this C# code uses the overloaded operators:

Code Example – C# multivariate functions

```
MultiVariableFunction g = f/2;
MultiVariableFunction sum = f + g;
MultiVariableFunction neg = -f;
```

This Visual Basic code uses the equivalent named methods:

Code Example – VB multivariate functions

```
Dim G = MultiVariableFunction.Divide(F, 2)
Dim Sum = MultiVariableFunction.Add(F, G)
Dim Neg = MultiVariableFunction.Negate(F)
```

CHAPTER 26. MINIMIZING UNIVARIATE FUNCTIONS

NMath provides classes for minimizing univariate functions using golden section search and Brent's method. Minimization is the process of finding the value of the variable *x* within some interval where *f(x)* takes on a minimum value. (To maximize a function *f*, simply minimize *-f*.)

All **NMath** minimization classes derive from the abstract base class **MinimizerBase**, which provides `Tolerance` and `MaxIterations` properties. In general, minimization stops when either the decrease in function value is less than the tolerance, or the maximum number of iterations is reached. Setting the error tolerance to less than zero ensures that the maximum number of iterations is always reached. After minimization, the following properties on **MinimizerBase** can be useful for gathering more information about the minimum just computed:

- `Error` gets the error associated with the mimimum just computed.
- `ToleranceMet` returns a boolean value indicating whether the minimum just computed stopped because the error tolerance was reached.
- `MaxIterationsMet` returns a boolean value indicating whether the minimum just computed stopped because the maximum number of iterations was reached.

The univariate minimization classes also implement one of the following interfaces:

- Classes that implement the **IOneVariableMinimizer** interface require only function evaluations to minimize a function.
- Classes that implement the **IOneVariableDMinimizer** interface also require evaluations of the derivative of a function.

This chapter describes how to use the univariate minimizer classes.

26.1 Bracketing a Minimum

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. Beginning with a pair of points, **Bracket** searches in the downhill direction for a new pair of points that bracket a minimum of a function. For example, if `function` is a **OneVariableFunction**:

Code Example – C# minimization

```
var bracket = new Bracket( function, 0, 1 );
```

Code Example – VB minimization

```
Dim Bracket As New Bracket( MyFunction, 0, 1 )
```

Once constructed, a **Bracket** object provides the following properties:

- `Function` gets the function whose minimum is bracketed.
- `Lower` gets a lower bound on a minimum of the function.
- `Upper` gets an upper bound on a minimum of the function.
- `Interior` gets a point between the lower and upper bound such that $x_{lower} < x_{interior} < x_{upper}$, $f(x_{interior}) < f(x_{lower})$, and $f(x_{interior}) < f(x_{upper})$
- `FLower` gets the function evaluated at the lower bound.
- `FUpper` gets the function evaluated at the upper bound.
- `FInterior` gets the function evaluated at the interior point.

26.2 Minimizing Functions Without Calculating the Derivative

NMath provides two classes that implement the **IOneVariableMinimizer** interface, and minimize a **OneVariableFunction** using only function evaluations:

- Class **GoldenMinimizer** performs a *golden section search* for a minimum of a function, by successively narrowing an interval know to contain a local minimum. The golden section search method is linearly convergent.
- Class **BrentMinimizer** uses *Brent's Method* to minimize a function. Brent's Method combines golden section search with parabolic interpolation. Parabolic interpolation fits a parabola through the current set of points, then uses the parabola to estimate the function's minimum. The faster

parabolic interpolation is used wherever possible, but in steps where the projected minimum falls outside the interval, or when successive steps are becoming larger, Brent's Method resorts back to the slower golden section search. Brent's Method is quadratically convergent.

Instances of **GoldenMinimizer** and **BrentMinimizer** are constructed by specifying an error tolerance and a maximum number of iterations, or by accepting the defaults for these values. For example, this code constructs a **GoldenMinimizer** using the default tolerance and a maximum of `50` iterations:

Code Example – C# minimization

```
int maxIter = 50;
var minimizer = new GoldenMinimizer( maxIter );
```

Code Example – VB minimization

```
Dim MaxIter As Integer = 50
Dim Minimizer As New GoldenMinimizer(MaxIter)
```

Instances of **GoldenMinimizer** and **BrentMinimizer** provide `Minimize()` methods for minimizing a given function within a given interval. Overloads of `Minimize()` accept a bounding **Interval**, a **Bracket**, or a triplet of points satisfying the bracketing conditions (Section 26.1). For example, the function

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

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

Code Example – C# minimization

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

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

Code Example – VB minimization

```
Public Shared Function MyFunction(X As Double) As Double
  Return Math.Pow(X - 1, 4)
End Function

Dim F As New OneVariableFunction(
  New Func(Of Double, Double)(AddressOf MyFunction))
```

This code finds a minimum of *f* in the interval `(0,2)` using golden section search:

Code Example – C# minimization

```
var minimizer = new GoldenMinimizer();
```

```
int lower = 0;
int upper = 2;
double min = minimizer.Minimize( f, lower, upper );
```

Code Example – VB minimization

```
Dim Minimizer As New GoldenMinimizer()
Dim Lower As Integer = 0
Dim Upper As Integer = 2
Dim Min As Double = Minimizer.Minimize(F, Lower, Upper)
```

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

Code Example – C# minimization

```
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 );
```

Code Example – VB minimization

```
Dim Tol As Double = "1e-9"
Dim MaxIter As Integer = 25
Dim Minimizer As New BrentMinimizer(Tol, MaxIter)
Dim Bracket As New Bracket(F, 0, 10)
Dim Min As Double = Minimizer.Minimize(Bracket)
```

26.3 Minimizing Derivable Functions

Class **DBrentMinimizer** implements the **IOneVariableDMinimizer** interface and minimizes a univariate function using *Brent's Method* in combination with evaluations of the first derivative. As described in Section 26.2, *Brent's Method* uses parabolic interpolation to fit a parabola through the current bracketing triplet, then uses the parabola to estimate the function's minimum. Class **DBrentMinimizer** uses the sign of the derivative at the central point of the bracketing triplet to decide which region should be used for the next test point.

Like **GoldenMinimizer** and **BrentMinimizer** (Section 26.2), instances of **DBrentMinimizer** are constructed by specifying an error tolerance and a maximum number of iterations, or by accepting the defaults for these values. This code constructs a **DBrentMinimizer** using the default error tolerance and maximum number of iterations:

Code Example – C# minimization

```
var minimizer = new DBrentMinimizer();
```

Code Example – VB minimization

```
Dim Minimizer As New DBrentMinimizer()
```

This code uses an error tolerance of `10-4` and a maximum of `50` iterations:

Code Example – C# minimization

```
double tol = 1e-4
int maxIter = 50;
var minimizer = new DBrentMinimizer( tol, maxIter );
```

Code Example – VB minimization

```
Dim Tol As Double = 0.0001
Dim MaxIter As Integer = 50
Dim Minimizer As New DBrentMinimizer(Tol, MaxIter)
```

Once you have constructed a **DBrentMinimizer** instance, you can use the `Minimize()` method to minimize a given function within a given interval. Overloads of `Minimize()` accept a bounding **Interval**, a **Bracket**, or a triplet of points satisfying the bracketing conditions (Section 26.1). Because **DBrentMinimizer** uses evaluations of the first derivative of the function, you must also supply a **OneVariableFunction** encapsulating the derivative. For example, the function:

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

has a minimum at `5.0`. To compute the minimum, first encapsulate the function and its derivative:

Code Example – C# minimization

```
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 Func<double, double>( MyFunction ) );
var df = new OneVariableFunction(
  new Func<double, double>( MyFunctionPrime ) );
```

Code Example – VB minimization

```
Public Shared Function MyFunction(X As Double) As Double
  Return ((x - 5) * (x - 5))
End Function
```

```
Public Shared Function MyFunctionPrime(X As Double) As Double
  Return (2 * X) - 10
End Function

Dim F As New OneVariableFunction(
  New Func(Of Double, Double)(AddressOf MyFunction))
Dim DF As New OneVariableFunction(
  New Func(Of Double, Double)(AddressOf MyFunctionPrime))
```

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

Code Example – C# minimization

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

Code Example – VB minimization

```
Dim Minimizer As New DBrentMinimizer()
Dim Bracket As New Bracket(F, 1, 2)
Dim Min As Double = Minimizer.Minimize(Bracket, DF)
```

CHAPTER 27.
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.

Like the univariate minimization classes described in Chapter 26, the multivariate minimization classes derive from the abstract base class **MinimizerBase**, which provides `Tolerance` and `MaxIterations` properties. In general, minimization stops when either the decrease in function value is less than the tolerance, or the maximum number of iterations is reached.

The multivariate minimization classes also implement one of the following interfaces:

- Classes that implement the **IMultiVariableMinimizer** interface require only function evaluations to minimize a function.
- Classes that implement the **IMultiVariableDMinimizer** interface also require evaluations of the derivative of a function.

This chapter describes how to use the multivariate minimizer classes.

27.1 Minimizing Functions Without Calculating the Derivative

NMath provides two classes that implement the **IMultiVariableMinimizer** interface, and minimize a **MultiVariableFunction** using only function evaluations (derivative calculations are not required):

- Class **DownhillSimplexMinimizer** minimizes a multivariate function using the downhill simplex method of Nelder and Mead.[3] A simplex in n-dimensions consists of $n+1$ distinct vertices. The method involves moving the simplex downhill, or if that is not possible, shrinking its size. The method is not highly efficient, and is appropriate only for small numbers of variables (usually fewer than 6), but is very robust. *Powell's Method* is faster in most applications (see below).

[3] J. A. Nelder and R. Mead (1965), "A Simplex Method for Function Minimization," Computer Journal, Vol. 7, p. 308-313.

- Class **PowellMinimizer** minimizes a multivariate function using *Powell's Method*. *Powell's Method* is a member of the family of *direction set* optimization methods, each of which is based on a series of one-dimensional line minimizations. The methods differ in how they choose the next dimension at each stage from among a current set of candidates. *Powell's Method* begins with a set of *N* linearly independent, mutually conjugate directions, and at each stage discards the direction in which the function made its largest decrease, to avoid a buildup of linear dependence. *Brent's Method* (Section 26.2) is used for the successive line minimizations.

Instances of **DownhillSimplexMinimizer** and **PowellMinimizer** are constructed by specifying an error tolerance and a maximum number of iterations, or by accepting the defaults for these values. For example, this code constructs a **PowellMinimizer** using the default tolerance and a maximum of `20` iterations:

Code Example – C# minimization

```
int maxIter = 20;
var minimizer = new PowellMinimizer( maxIter );
```

Code Example – VB minimization

```
Dim MaxIter As Integer = 20
Dim Minimizer As New PowellMinimizer(MaxIter)
```

Class **DownhillSimplexMinimizer** and **PowellMinimizer** implement the **IMultiVariableMinimizer** interface, which provides a single `Minimize()` method that takes a **MultiVariableFunction** to minimize, and a starting point. For instance, if `f` is an encapsulated multivariate function (Chapter 25) of three variables, this code minimizes the function using the downhill simplex method, starting at the origin:

Code Example – C# minimization

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

Code Example – VB minimization

```
Dim Minimizer As New DownhillSimplexMinimizer()
Dim Start As New DoubleVector(0.0, 0.0, 0.0)
Dim Min As DoubleVector = Minimizer.Minimize(F, Start)
```

Both **DownhillSimplexMinimizer** and **PowellMinimizer** provide additional overloads of `Minimize()` that allow you more control over the initial conditions. The downhill simplex method, for example, begins with an initial simplex consisting of $n+1$ distinct vertices. If you provide only a starting point, as illustrated above, a starting simplex is constructed by adding `1.0` in each dimension. For example, in two dimensions the simplex is a triangle. If the starting point is (x_0, x_1), the remaining vertices of the starting simplex will be (x_0+1, x_1) and

(x_0, x_1+1). Overloads of the `Minimize()` method allow you to specify the amount added in each dimension from the starting point when constructing the initial simplex, or simply to specify the initial simplex itself.

Similarly, *Powell's Method* begins with an initial direction set, a set of *N* linearly independent, mutually conjugate directions. An overload of `Minimize()` enables you to specify the initial direction set. If you provide only a starting point to the `Minimize()` method, as illustrated above, the starting direction set is simply the unit vectors.

27.2 Minimizing Derivable Functions

NMath provides two classes that implement the **IMultiVariableDMinimizer** interface, and minimize a **MultiVariableFunction** using function evaluations and derivative calculations:

- Class **ConjugateGradientMinimizer** minimizes a multivariate function using the Polak-Ribiere variant of the Fletcher-Reeves *conjugate gradient* method. Gradients are calculated using the partial derivatives, then chosen based on a direction that is conjugate to the old gradient and, insofar as possible, to all previous directions traversed.
- Class **VariableMetricMinimizer** minimizes a multivariate function using the Broyden-Fletcher-Goldfarb-Shanno *variable metric* (or *quasi-Newton*) method. Variable metric methods are very similar to conjugate gradient methods—both calculate gradients using the partial derivatives. Storage is less efficient (order N^2 storage, versus order a few times N), but since variable metric methods predate conjugate gradient methods, they are still widely used.

Like all **NMath** minimizers, instances of **ConjugateGradientMinimizer** and **VariableMetricMinimizer** are constructed by specifying an error tolerance and a maximum number of iterations, or by accepting the defaults for these values. For example, this code constructs a **VariableMetricMinimizer** using a tolerance or 10^{-5} and a maximum of `20` iterations:

Code Example – C# minimization

```
double tol = 1e-5;
int maxIter = 20;
VariableMetricMinimizer minimizer =
  new VariableMetricMinimizer( tol, maxIter );
```

Code Example – VB minimization

```
Dim Tol As Double = "1e-5"
Dim MaxIter As Integer = 20
```

```
Dim Minimizer As New VariableMetricMinimizer(Tol, MaxIter)
```

Class **ConjugateGradientMinimizer** and **VariableMetricMinimizer** implement the **IMultiVariableDMinimizer** interface, which provides a single `Minimize()` method with the following signature:

Code Example – C# minimization

```
DoubleVector Minimize( MultiVariableFunction f,
                       MultiVariableFunction[] df,
                       DoubleVector x );
```

Code Example – VB minimization

```
Minimize(F As MultiVariableFunction,
         DF As MultiVariableFunction(),
         X as DoubleVector) As DoubleVector
```

where `f` is the function to minimize, `df` is an array of partial derivatives, and `x` is the start point.

For instance, given the following function and partial derivatives:

Code Example – C# minimization

```
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;
}
```

Code Example – VB minimization

```
Protected Shared Function MyFunction(V As DoubleVector) As Double
  Return ((V(0) - 5.0) * (V(0) - 5.0)) +
         ((V(1) + 3.0) * (V(1) + 3.0))
End Function

Protected Shared Function Dx(V As DoubleVector) As Double
  Return (2 * V(0)) - 10
End Function

Protected Shared Function Dy(V As DoubleVector) As Double
```

```
  Return (2 * V(1)) + 6
End Function
```

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

Code Example – C# minimization

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

var partialx = new MultiVariableFunction(
  new Func<DoubleVector, double>( MyFunctionDx ) );
var partialy = new MultiVariableFunction(
  new Func<DoubleVector, double>( MyFunctionDy ) );
var df = new MultiVariableFunction[] { partialx, partialy };

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

Code Example – VB minimization

```
Dim MultiFunction As New MultiVariableFunction(
  New Func(Of DoubleVector, Double)(AddressOf MyFunction))
Dim PartialX As New MultiVariableFunction(
  New Func(Of DoubleVector, Double)(AddressOf Dx))
Dim PartialY As New MultiVariableFunction(
  New Func(Of DoubleVector, Double)(AddressOf Dy))

Dim Minimizer As New ConjugateGradientMinimizer()
Dim Start As New DoubleVector(2, 0)
Dim Min As DoubleVector = Minimizer.Minimize(F, DF, Start)
```

CHAPTER 28. 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.

This chapter describes how to use class **AnnealingMinimizer**.

28.1 Temperature

Temperature values are simply scalars used at each iteration of the minimization to introduce noise into the process. Each search movement is jittered `+/- T ln(r)`, where `r` is a random deviate between `0` and `1`.

Temperatures that are too low, or that drop too quickly, increase the likelihood of getting caught in a local minimum. Temperatures that are too high simply cause minimization to jump randomly around the search space without settling into a solution. Annealing schedules must therefore be chosen carefully. Unfortunately, this is something of a trial-and-error process. What is an appropriate regime will be entirely dependent on the characteristics of the function being minimized, which may not be well understood in advance.

28.2 Annealing Schedules

In simulated annealing, the annealing schedule governs the choice of initial temperature, how many iterations are performed at each temperature, and how much the temperature is decremented at each step as cooling proceeds.

For example, the annealing schedule shown in Table 21 has four steps.

Table 21 – 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.

NOTE—Annealing schedules must end with a temperature of zero. Otherwise, they never converge on a minimum.

In **NMath**, **AnnealingScheduleBase** is the abstract base class for classes that define annealing schedules. Two concrete implementations are provided.

Linear Annealing Schedules

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 21:

Code Example – C# simulated annealing

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

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

Code Example – VB simulated annealing

```
Dim Steps = 4
Dim IterationsPerStep = 20
Dim StartTime As Double = 100.0

Dim Schedule As New LinearAnnealingSchedule(Steps,
  IterationsPerStep, StartTemp)
```

You may optionally also provide a non-default error tolerance. At each annealing step, iteration stops if the estimated error is less than the tolerance, but typically this only occurs during the final step, when the temperature is zero.

Once constructed, a **LinearAnnealingSchedule** instance provides the following properties:

- `Steps` gets the number of steps in the schedule.
- `Iterations` gets and sets the number of iterations per step.
- `TotalIterations` gets and sets the total number of iterations in this schedule. When set, the number of iterations per step is scaled appropriately.
- `StartingTemperature` gets and sets the starting temperature.
- `Tolerance` gets and sets the error tolerance used in computing minima estimates.

Custom Annealing Schedules

For more control over the temperature decay, you can use class **CustomAnnealingSchedule**. Instances of **CustomAnnealingSchedule** are constructed from an array containing the number of iterations for each step, and the temperature for each step.

For example:

Code Example – C# simulated annealing

```
var iterations = new int[] { 50, 30, 20, 20 };
var temps = new double[] { 75.3, 20.0, 10.5, 0.0 };

var schedule =
  new CustomAnnealingSchedule( iterations, temps );
```

Code Example – VB simulated annealing

```
Dim Iterations() As Integer = {50, 30, 20, 20}
Dim Temps() As Double = {75.3, 20.0, 10.5, 0.0}

Dim Schedule As New CustomAnnealingSchedule(Iterations, Temps)
```

NOTE—An InvalidArgumentException is raised if the final temperature in a custom annealing schedule is not zero. Without a final temperature of zero, the system never settles into a minimum.

You may optionally also provide a non-default error tolerance. At each annealing step, iteration stops if the estimated error is less than the tolerance, but typically this only occurs during the final step, when the temperature is zero.

Once constructed, a **CustomAnnealingSchedule** instance provides the following properties:

- `Steps` gets the number of steps in the schedule.
- `Iterations` gets and sets the arrray of iterations for each step.
- `TotalIterations` gets and sets the total number of iterations in this schedule. When set, the number of iterations per step is scaled appropriately.
- `Temperatures` gets and sets the vector of temperatures for each step.
- `Tolerance` gets and sets the error tolerance used in computing minima estimates.

28.3 Minimizing Functions by Simulated Annealing

Instances of **AnnealingMinimizer** are constructed from an annealing schedule (Section 28.2). For instance:

Code Example – C# simulated annealing

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

Code Example – VB simulated annealing

```
Dim Schedule As New LinearAnnealingSchedule(5, 25, 100.0)
Dim Minimizer As New AnnealingMinimizer(Schedule)
```

After construction, you can use the `Schedule` property to get and set the annealing schedule associated with an **AnnealingMinimizer**.

The `RandomNumberGenerator` property gets and sets the random number generator associated with this minimizer. The random number generator is used for making temperature-dependent, random steps in the search space as part of the annealing process. The random number generator is initially set at construction time to the value of static property `DefaultRandomNumberGenerator`, which defaults to an instance of **RandGenUniform**.

Class **AnnealingMinimizer** implements the **IMultiVariableMinimizer** interface, which provides a single `Minimize()` method that takes a **MultiVariableFunction** to minimize, and a starting point. For instance, if `f` is an encapsulated multivariable function (Chapter 25) of five variables, this code minimizes the function using the downhill simplex method, starting at `(0.2,0.2,-.2,0.0,0.0)`:

Code Example – C# simulated annealing

```
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 );
```

Code Example – VB simulated annealing

```
Dim Minimizer As New AnnealingMinimizer(Schedule)
Dim Start As New DoubleVector(0.2, 0.2, -0.2, 0.0, 0.0)
Dim Min As DoubleVector = Minimizer.Minimize(F, Start)
```

After minimization, the following properties on **AnnealingMinimizer** can be useful for gathering more information about the minimum just computed:

- `Error` gets the error associated with the mimimum just computed.
- `ToleranceMet` returns a boolean value indicating whether the minimum just computed stopped because the error tolerance was reached. (At each annealing step, iteration stops if the estimated error is less than the tolerance, but typically this only occurs during the final step, when the temperature is zero.)

For more information on the annealing process just completed, access the annealing history (Section 28.4).

28.4 Annealing History

For annealing to successfully locate the global minimum of a function, an appropriate annealing schedule must be chosen, but unfortunately this is something of a trial-and-error process. An appropriate regime is entirely dependent on the characteristics of the function being minimized, which may not be well understood in advance.

To help you in this process, class **AnnealingMinimizer** can be configured to keep a history of the annealing process. There is a cost in memory and execution to record this information, so it is not enabled by default. To record the annealing history, set the `KeepHistory` property to `true`. Thus:

Code Example – C# simulated annealing

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

Code Example – VB simulated annealing

```
Dim Minimizer As New AnnealingMinimizer(Schedule)
Minimizer.KeepHistory = True
```

AnnealingMinimizer performs a minimization at each step in an annealing schedule. 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:

Code Example – C# simulated annealing

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

Code Example – VB simulated annealing

```
Dim Min As DoubleVector = Minimizer.Minimize(F, StartingPoint)
Dim History As AnnealingHistory = Minimizer.AnnealingHistory
Console.WriteLine(History)
```

AnnealingHistory also provides a variety of properties for accessing specific information:

- `Function` gets the function that was minimized.
- `MaximumIterations` gets the number of maximum iterations at each step in the annealing history.
- `Iterations` gets the number of iterations actually performed at each step in the annealing history.
- `Temperatures` gets the temperatures at each step in the annealing history.
- `Simplexes` gets the starting simplexes at each step in the annealing history.
- `MinimumPoints` gets the minima computed at each step in the annealing history.
- `MinimumValues` gets the function evaluated at the minima computed at each step in the annealing history.
- `Errors` gets the errors at each step in the annealing history.

The inner class **AnnealingHistory.Step** encapsulates all of the data associated with a particular step in an **AnnealingHistory**. The `AnnealingHistory.Steps` property returns a **IList** of the steps in the annealing history:

Code Example – C# simulated annealing

```
AnnealingHistory history = minimizer.AnnealingHistory;
foreach( AnnealingHistory.Step step in history )
{
  Console.WriteLine( step );
}
```

Code Example – VB simulated annealing

```
Dim History As AnnealingHistory = Minimizer.AnnealingHistory

For Each AnnealingStep As AnnealingHistory.Step In History
  Console.WriteLine(AnnealingStep)
Next
```

The provided indexer can also be used to retrieve information about a particular step. For example, this code prints out a summary of the third step:

Code Example – C# simulated annealing

```
Console.WriteLine( history[3] );
```

Code Example – VB simulated annealing

```
Console.WriteLine(History(3))
```

CHAPTER 29. LINEAR PROGRAMMING

A linear programming (LP) problem optimizes a linear objective function subject to a set of linear constraints, and optionally subject to a set of variable bounds. For example:

```
Maximize
Z = X1 + 4 X2 + 9 X3

Subject To
X1 + X2 <= 5
X1 + X3 >= 10
-X2 + X3 = 7

Bounds
0 <= X1 <= 4
0 <= X2 <= 1
```

In **NMath**, class **LinearProgrammingProblem** encapsulates an LP problem. **MixedIntegerLinearProgrammingProblem** encapsulates an LP problem which may contain integral or binary constraints.

Class **PrimalSimplexSolverORTools** solves linear programming problems using the primal simplex method. The class **DualSimplexSolverORTools** uses the dual simplex method. The simplex method solves LP problems by constructing an initial solution at a vertex of a simplex, then walking along edges of the simplex to vertices with successively higher values of the objective function until the optimum is reached. These two classes use the Google OR Tools computational engine for solving both LP and MIP problems.

This chapter describes how to solve LP problems using **NMath**.

29.1 Encapsulating LP Problems

Class **LinearProgrammingProblem** encapsulates an LP problem. Instances are constructed from a vector of coefficients representing the objective function.

Code Example – C# linear programming

```
// z = x1 + 4*x2 + 9*x3
var coeff = new DoubleVector( "[1 4 9]" );
var problem = new LinearProgrammingProblem( coeff );
```

MixedIntegerLinearProgrammingProblem encapsulates an LP problem which may contain integer or binary constraints.

Adding Bounds and Constraints

LinearProgrammingProblem instances maintain a list of **LinearContraint** objects, accessible via the `Constraints` property. A *linear constraint* on a set of variables is a constraint upon a linear combination of those variables. **LinearConstraint** supports to two such constraints: equality constraints and lower bound constraints. That is, given variables *x0*, *x1*,..., *xn* and constants *b*, *a0*, *a1*,..., *an*, two types of constraints may be formed

```
a0*x0 + a1*x1 + . . . + an*xn = b
```

and

```
a0*x0 + a1*x1 + . . . + an*xn >= b
```

NOTE—Upper bound constraints are represented as negations of lower bound constraints.

Constraints may be added to a **LinearProgrammingProblem** by working directly with the `Constraints` list, or by using the `AddConstraint()` method.

LinearContraint instances are constructed from a vector of coefficients, a right-hand side, and a constraint type from the **ConstraintType** enumeration.

Code Example – C# linear programming

```
// 0 <= x0 + 2*x1 + 2*x2
var coeff = new DoubleVector( 1.0, 2.0, 2.0 );
var constraint = new LinearConstraint( coeff, 0,
  ConstraintType.GreaterThanOrEqualTo);
problem.AddConstraint( constraint );
```

A variety of convenience methods are also provided on **LinearProgrammingProblem** for adding constraints and variable bounds to an existing LP problem. These methods create the required **LinearConstraint** objects for you and add them to the `Constraints` list.

Code Example – C# linear programming

```
// 0 <= x0 + 2*x1 + 2*x2 <= 72
var coeff = new DoubleVector( 1.0, 2.0, 2.0 );
problem.AddConstraint( coeff, 0, 72 );
```

MixedIntegerLinearProgrammingProblem encapsulates an LP problem which may contain integer or binary constraints. For example, in this code the first variable is constrained to be integer valued.

Code Example – C# integer programming

```
problem.AddIntegralConstraint( 0 );
```

Here, the *i*th variable in the solution must be `binary`.

Code Example – C# binary programming

```
problem.AddBinaryConstraint( i );
```

A binary constraint restricts the variable to a value of zero or one.

Method `GetIntegrality()` gets the integral constraint state of the variable at the given index. `IntegralVariableIndices` returns the indices of variables with integral constraints.

29.2 Solving LP Problems

Class **PrimalSimplexSolverORTools** solves linear programming problems using the primal simplex method. **DualSimplexSolverORTools** uses the dual simplex method. The simplex method solves LP problems by constructing an initial solution at a vertex of a simplex, then walking along edges of the simplex to vertices with successively higher values of the objective function until the optimum is reached.

The `Solve()` method takes a **LinearProgrammingProblem** or **MixedIntegerLinearProgrammingProblem** and, optionally, a boolean variable to indicate if the objective is to be minimized (`true`) or maximized.

Code Example – C# linear programming

```
var solver = new PrimalSimplexSolverORTools();
solver.Solve( problem, true );
```

This code demonstrates using a solver parameter object.

Code Example – C# linear programming

```
var solver = new DualSimplexSolverORTools();
solver.Solve( problem, true );
```

It is important to check whether a finite solution was found, since your problem may be unbounded or infeasible. If a finite solution was found, you can access the solution using the `OptimalX` property. The `OptimalObjectiveFunctionValue` property gets the value of the objective function evaluated at the solution.

Code Example – C# linear programming

```
if ( solver.Result ==
```

```
PrimalSimplexSolverORTools.SolveResult.Optimal )
{
  Console.WriteLine( solver.OptimalX );
  Console.WriteLine( solver.OptimalObjectiveFunctionValue );
}
```

If the solver result is `SolverResult.Optimal`, then the `solver.OptimalX` will contain the optimal solution with all constraints satisfied. Otherwise the `SolverResult` object may indicate one of the following results: `Feasible, Infeasible, Unbounded, Abnormal, or NotSolved.` The specified optimal X vector is not valid if the solver indicates either an unbounded, abnormal or not solved flag.

CHAPTER 30. NONLINEAR AND QUADRATIC PROGRAMMING

NMath provides classes for solving both Nonlinear Programming (NLP) and Quadratic Programming (QP) problems.

This chapter describes how to use QP and NLP classes.

30.1 Objective and Constraint Function Classes

Nonlinear and quadratic programming problems seek to minimize an objective function, subject to a set of constraint functions. **NMath** provides classes for encapsulating these functions, used by both QP and NLP solvers.

Objective Function Classes

Two classes support objective functions:

- Class **DoubleFunctional** is an abstract class which derives from **DoubleMultiVariableFunction**. It is a particular type of multivariable function, where the dimension of the range space is one. Deriving classes must implement the `Evaluate()` method, and may optionally provide a `Gradient()` method.
- Since it is sometimes convenient to specify the objective function and its corresponding gradient using delegates (including anonymous delegates and lambda expressions), class **DoubleFunctionalDelegate** derives from **DoubleFunctional** and provides an easy way to wrap delegates in a **DoubleFunctional** interface. Thus, all functions which take a **DoubleFunctional** argument are overloaded to take a delegate argument.

For example, this code sub-classes **DoubleFunctional** to encapsulate an objective function:

Code Example – C#

```
// f(x) = exp(x0)*(4*x0^2 + 2*x1^2 + 4*x0*x1 + 2*x1 + 1)

class MyObjectiveFunction : DoubleFunctional
{
```

```
  // Constructor. Must initilialize the base class with the
  // dimension of the domain--2 in this case.
  public ObjectiveFunction()
    : base( 2 )
  {}

  public override double Evaluate( DoubleVector x )
  {
    double x0 = x[0];
    double x1 = x[1];
    return Math.Exp( x0 ) * ( 4 * x0 * x0 + 2 * x1 * x1 + 4 * x0 *
      x1 + 2 * x1 + 1 );
  }

  public override void Gradient( DoubleVector x,
    DoubleVector grad )
  {
    double x0 = x[0];
    double x1 = x[1];
    double ex0 = Math.Exp( x0 );
    grad[0] = ex0 * ( 4 * x0 * x0 + 2 * x1 * x1 + 4 * x0 * x1 + 2 *
      x1 + 1 ) + ex0 * ( 8 * x0 + 4 * x1 );
    grad[1] = ex0 * ( 4 * x0 + 4 * x1 + 2 );
  }
}
```

Code Example – VB

```
' f(x) = exp(x0)*(4*x0^2 + 2*x1^2 + 4*x0*x1 + 2*x1 + 1)
Public Class MyObjectiveFunction
  Inherits DoubleFunctional

  ' Constructor. Must initilialize the base class with the
  ' dimension of the domain--2 in this case.
  Public Sub New()
    MyBase.New(2)
  End Sub

  Public Overrides Function Evaluate(X As DoubleVector) As Double
    Dim X0 As Double = X(0)
    Dim X1 As Double = X(1)
    Return Math.Exp(X0) * (4 * X0 * X0 + 2 * X1 * X1 + 4 * X0 *
      X1 + 2 * X1 + 1)
  End Function

  Public Overrides Sub Gradient(X As DoubleVector, Grad As
DoubleVector)
    Dim X0 = X(0)
    Dim X1 = X(1)
    Dim EX0 = Math.Exp(X0)
```

```
    Grad(0) = EX0 * (4 * X0 * X0 + 2 * X1 * X1 + 4 * X0 * X1 + 2 *
      X1 + 1) + EX0 * (8 * X0 + 4 * X1)
    Grad(1) = EX0 * (4 * X0 + 4 * X1 + 2)
  End Sub

End Class
```

This code uses a **DoubleFunctionalDelegate**:

Code Example – C#

```
public double MyFunction( DoubleVector x )
{
  // f(x) = -x0 * x1 *x2
  return -x[0] * x[1] * x[2];
}

int xDim = 3;
Func<DoubleVector, double> functional = MyFunction;

var objective = new DoubleFunctionalDelegate( xDim, functional )
```

Code Example – VB

```
Public Function MyFunction(X As DoubleVector) As Double
  ' f(x) = -x0 * x1 *x2
  Return -X(0) * X(1) * X(2)
End Function

Dim XDim As Integer = 3
Dim Functional As New Func(Of DoubleVector, Double
  (AddressOf MyFunction)

Dim Objective As New DoubleFunctionalDelegate(XDim, Functional)
```

Constraint Function Classes

NMath provides two concrete constraint classes: **LinearConstraint** and **NonlinearConstraint**, which both derive from the abstract base class **Constraint**. Constraint objects contain a constraint function *c(x)* and a constraint type, either equality or inequality, specified using the **ConstraintType** enumeration.

NOTE—It is assumed that equality type constraints have their constraint function c(x) equal to zero, and inequality type constraints have their constraint function c(x) greater than or equal to zero.

A *linear constraint* on a set of variables is a constraint upon a linear combination of those variables. **LinearConstraint** supports to two such constraints: equality constraints and lower bound constraints. That is, given variables *x0, x1,..., xn* and constants *b, a0, a1,..., an*, two types of constraints may be formed

```
a0*x0 + a1*x1 + . . . + an*xn = b
```

and

```
a0*x0 + a1*x1 + . . . + an*xn >= b
```

Upper bound constraints are represented as negations of lower bound constraints.

Nonlinear constraints are of the form $c(x) \geq 0$ (inequality constraint), or $c(x) = 0$ (equality constraint), where *c(x)* is a real-valued, smooth function of the vector variable. Constraints can also be constructed with a tolerance. Equality constraints are satisfied when $|c(x)| \leq \text{tolerance}$; inequality constraints are satisfied when $c(x) \geq -\text{tol}$.

In most cases, you will not need to create constraint objects directly. QP and NLP problem classes provide methods for adding constraints which construct the necessary constraint objects for you.

30.2 Nonlinear Programming

A general formulation of a nonlinear programming (NLP) problem is:

$$\begin{array}{l} \min f(x) \\ x \in R^n \end{array}$$

subject to

$$c_i(x) = 0, i \in E$$

$$c_i(x) \geq 0, i \in I$$

where the functions *f* and c_i are all smooth (continuous derivative), real-valued functions on a subset of R^n, and *E* and *I* are finite sets of indices. Function *f* is called the *objective function*, and functions c_i are called the *constraint functions*.

Encapsulating the Problem

In **NMath**, class **NonlinearProgrammingProblem** encapsulates an NLP problem. **MixedIntegerNonlinearProgrammingProblem** encapsulates an NLP which may contain integral or binary constraints.

Instances are constructed from an objective function to minimize, and optionally an **IEnumerable** of **Constraint** objects. Alternatively, constraints can be added post-construction using convenience methods.

For example, if **MyObjectiveFunction** extends **DoubleFunctional** (see Section 30.1):

Code Example – C# nonlinear programming

```
DoubleFunctional objective = new MyObjectiveFunction();
var problem = new NonlinearProgrammingProblem( objective );
```

Code Example – VB nonlinear programming

```
Dim Objective As DoubleFunctional = New MyObjectiveFunction()
Dim Problem As New NonlinearProgrammingProblem(Objective)
```

Rather than sub-classing, you can also use a delegate to express the objective function, in which case you must also specify the dimension of the domain of the objective function. For instance:

Code Example – C# nonlinear programming

```
public double MyFunction( DoubleVector x )
{
  // min f(x) = -x0 * x1 * x2
  return -x[0] * x[1] * x[2];
}

int xDim = 3;
Func<DoubleVector, double> objective = MyFunction;
var problem = new NonlinearProgrammingProblem( xDim, objective );
```

Code Example – VB nonlinear programming

```
Public Function MyFunction(X As DoubleVector)
  ' min f(x) = -x0 * x1 * x2
  Return -X(0) * X(1) * X(2)
End Function

Dim XDim As Integer = 3
Dim Objective As New Func(Of DoubleVector, Double)(MyFunction)
Dim Problem As New NonlinearProgrammingProblem(XDim, Objective)
```

This code specifies two constraints in the constructor:

Code Example – C# nonlinear programming

```
var constraints = new List<Constraint>();
var c1 =
    new DoubleFunctionalDelegate( 2, new Func<DoubleVector,
        double>(delegate(DoubleVector v) { return v[0]; }) );
var constraint1 = new NonlinearConstraint(
    c1, ConstraintType.GreaterThanOrEqualTo )
constraints.Add( constraint1 );

var c2 =
```

```
    new DoubleFunctionalDelegate(2, new Func<DoubleVector,
        double>(delegate(DoubleVector v) { return v[1]; }));
var constraint2 = new NonlinearConstraint(
    c2, ConstraintType.GreaterThanOrEqualTo )
constraints.Add( constraint2 );

var problem =
    new NonlinearProgrammingProblem( objective, constraints );
```

Code Example – VB nonlinear programming

```
Dim Constraints As New List(Of Constraint)
Dim C1 As New DoubleFunctionalDelegate(2,
  New Func(Of DoubleVector, Double)(MyConstraintFunction1))
Dim Constraint1 As New NonlinearConstraint(C1,
  ConstraintType.GreaterThanOrEqualTo)
Constraints.Add(Constraint1)

Dim C2 As New DoubleFunctionalDelegate(2,
  New Func(Of DoubleVector, Double)(MyConstraintFunction2))
Dim Constraint2 As New NonlinearConstraint(C2,
  ConstraintType.GreaterThanOrEqualTo)
Constraints.Add(Constraint2)

Dim Problem As New NonlinearProgrammingProblem(Objective,
  Constraints)
```

Adding Bounds and Constraints

Class **NonlinearProgrammingProblem** provides several convenience methods for adding constraints and variable bound to an existing problem object.

For example, this code adds lower and upper variable bounds:

Code Example – C# nonlinear programming

```
// 0 <= x0, x1, x2 <= 42
for ( int i = 0; i < xDim; i++ ) {
    problem.AddBounds( i, 0.0, 42.0 );
}
```

Code Example – VB nonlinear programming

```
' 0 &lt;= x0, x1, x2 &lt;= 42
For I As Integer = 0 To XDim - 1
  Problem.AddBounds(I, 0.0, 42.0)
Next
```

This code adds a linear constraint:

Code Example – C# nonlinear programming

```
// 0 <= x0 + 2*x1 + 2*x2 <= 72,
problem.AddLinearConstraint( new DoubleVector( 1.0, 2.0, 2.0 ),
    0.0, 72 );
```

Code Example – VB nonlinear programming

```
' 0 &lt;= x0 + 2*x1 + 2*x2 &lt;= 72,
Problem.AddLinearConstraint(New DoubleVector(1.0, 2.0, 2.0),0.0,
  72)
```

This code adds constraint functions:

Code Example – C# nonlinear programming

```
int xDim = 2;

// x0*x1 >= -10
problem.AddLowerBoundConstraint( xDim,
    ( DoubleVector x ) => x[0] * x[1], -10.0 );

// x0*x1 - x0 -x1 <= -1.5
problem.AddUpperBoundConstraint( xDim,
    ( DoubleVector x ) => x[0] * x[1] - x[0] - x[1], -1.5 );
```

Code Example – VB nonlinear programming

```
Dim XDim As Integer = 2

Public Function LowerConstraint(X As DoubleVector) As Double
  Return X(0) * X(1)
End Function

Public Function UpperConstraint(X As DoubleVector) As Double
  Return X(0) * X(1) - X(0) - X(1)
End Function

' x0*x1 &gt;= -10
Problem.AddLowerBoundConstraint(xDim, New Func(Of DoubleVector,
  Double)(AddressOf LowerConstraint), -10.0)

' x0*x1 - x0 -x1 &lt;= -1.5
Problem.AddUpperBoundConstraint(xDim, New Func(Of DoubleVector,
  Double)(AddressOf UpperConstraint), -1.5)
```

MixedIntegerNonlinearProgrammingProblem encapsulates an NLP which may contain integral or binary constraints. For example, in this code variable index `2` is constrained to be integer valued.

Code Example – C# integer programming

```
problem.AddIntegralConstraint( 2 );
```

Here, variable indices `0` and `1` must be binary.

Code Example – C# binary programming

```
problem.AddBinaryConstraint( 0, 1 );
```

A binary constraint restricts the variable to a value of zero or one.

Method `GetIntegrality()` gets the integral constraint state of the variable at the given index. `IntegralVariableIndices` returns the indices of variables with integral constraints.

Solving the Problem

NMath provides two types of NLP solvers: Stochastic Hill Climbing and Sequential Quadratic Programming (SQP).

Stochastic Hill Climbing

The strategy of the Stochastic Hill Climbing algorithm is to iteratively make small random changes to the decision values. A candidate solution is accepted if it results in an improvement, and rejected if it makes it worse. The strategy addresses the limitations of deterministic hill climbing techniques, which are prone to getting stuck in local optima due to their greedy acceptance of neighboring moves.

StochasticHillClimbingSolver solves NLP problems using the Stochastic Hill Climbing algorithm.

Code Example – C# nonlinear programming

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

Code Example – VB nonlinear programming

```
Dim Solver As New StochasticHillClimbingSolver()
```

The algorithm is stochastic. Setting a random seed ensures consistent results between runs.

Code Example – C# nonlinear programming

```
solver.RandomSeed = 0x248;
```

Code Example – VB nonlinear programming

```
Solver.RandomSeed = &H248
```

Additional parameters are specified using an instance of **StochasticHillClimbingParameters**.

Code Example – C# nonlinear programming

```
var solverParams = new StochasticHillClimbingParameters
{
  TimeLimitMilliSeconds = 10000,
  Presolve = true
};
```

Code Example – VB nonlinear programming

```
Dim SolverParams As New StochasticHillClimbingParameters()
SolverParams.TimeLimitMilliSeconds = 10000
SolverParams.Presolve = True
```

Note that this example sets a time limit of `10` seconds for the solver. By default, the solver runs until a solution is found. Since this may take forever, it is a good idea to set a reasonable time limit on the solve. If an optimal solution is not found within the specified time limit, the solver exits and the solver's `Result` property will be equal to `SolverResult.SolverInterrupted`.

This example also sets `Presolve = true`. By default there is no pre-solve step. For some problems pre-solve can reduce the size and complexity and result in fewer steps to reach a solution.

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

Code Example – C# nonlinear programming

```
solver.Solve( problem, solverParams );
Console.WriteLine( "Solver Result = " + solver.Result );
Console.WriteLine( "Number of steps = " + solver.Steps );
Console.WriteLine( "Optimal x = " + solver.OptimalX );
Console.WriteLine( "Optimal function value = " +
  solver.OptimalObjectiveFunctionValue );
```

Code Example – VB nonlinear programming

```
Solver.Solve(Problem, SolverParams)
Console.WriteLine("Solver Result = {0}", Solver.Result)
Console.WriteLine("Number of steps = {0}", Solver.Steps)
Console.WriteLine("Optimal x = {0}", Solver.OptimalX)
Console.WriteLine("Optimal function value = {0}",
  Solver.OptimalObjectiveFunctionValue)
```

Sequential Quadratic Programming (SQP)

SQP algorithms solve NLP problems iteratively. At each step, a quadratic sub-problem is formed from the Hessian of the Lagrangian, H_k, the constraints, and the current iterate value x_k. The solution of this sub-problem yields a step direction p_k. Next a step size a_k is determined, and the new iterate value is obtained as $x_{k+1}=x_k+a_k\, p_k$.

SequentialQuadraticProgrammingSolver is the abstract base class for SQP solvers. **NMath** currently provides one concrete implementation: **ActiveSetLineSearchSQP** solves NLP problems using an active set algorithm.

Code Example – C# nonlinear programming

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

Code Example – VB nonlinear programming

```
Dim Solver As New ActiveSetLineSearchSQP()
```

A convergence tolerance and maximum number of iterations can also be specified in the constructor, as well as other advanced options (see below):

Code Example – C# nonlinear programming

```
double tolerance = 1e-4;
var solver = new ActiveSetLineSearchSQP( tolerance );
```

Code Example – VB nonlinear programming

```
Dim Tolerance As Double = "1e-4"
Dim Solver As New ActiveSetLineSearchSQP(Tolerance)
```

The `Solve()` method solves the problem given an initial starting position, and returns `true` if the algorithm terminated successfully:

Code Example – C# nonlinear programming

```
var x0 = new DoubleVector( 3, 1.0 );
bool success = solver.Solve( problem, x0 );
Console.WriteLine( "Termination status = " +
    solver.SolverTerminationStatus );
```

Code Example – VB nonlinear programming

```
Dim X0 As New DoubleVector(3, 1.0)
Dim Success = Solver.Solve(Problem, X0)
Console.WriteLine("Termination status = {0}",
  Solver.SolverTerminationStatus)
```

Properties on the solver get the minimum *x*-value found, and the objective function evaluated at that point:

Code Example – C# nonlinear programming

```
Console.WriteLine( "X = " + solver.OptimalX );
Console.WriteLine( "f(x) = " +
  solver.OptimalObjectiveFunctionValue );
```

Code Example – VB nonlinear programming

```
Console.WriteLine("X = {0}", Solver.OptimalX)
Console.WriteLine("f(x) = {0}",
   Solver.OptimalObjectiveFunctionValue)
```

ActiveSetLineSearchSQP.Options provides advanced options for controlling the **ActiveSetLineSearchSQP** algorithm, such as the step size and finer grain convergence tolerances:

Code Example – C# nonlinear programming

```
var solverOptions = new ActiveSetLineSearchSQP.Options();
solverOptions.StepSizeCalculator = new ConstantSQPStepSize( 1 );
solverOptions.StepDirectionTolerance = 1e-8;
solverOptions.FunctionChangeTolerance = 1e-6;

var solver = new ActiveSetLineSearchSQP( solverOptions );
```

Code Example – VB nonlinear programming

```
Dim SolverOptions As New ActiveSetLineSearchSQP.Options()

SolverOptions.StepSizeCalculator = New ConstantSQPStepSize(1)
SolverOptions.StepDirectionTolerance = "1e-8"
SolverOptions.FunctionChangeTolerance = "1e-6"

Dim Solver As New ActiveSetLineSearchSQP(SolverOptions)
```

This code sets the step size calculator to use an instance of **ConstantSQPStepSize**, which simply returns a constant step size regardless of iteration values. By default, the **ActiveSetLineSearchSQP** algorithm uses an instance of **L1MeritStepSize**, which computes the step size based on sufficient decrease in the L1 merit function.

30.3 Quadratic Programming

A quadratic programming (QP) problem is a NLP problem with a specific form for the objective and constraint functions. A QP problem has the following form:

$$\min_{x \in R^n} q(x) = \frac{1}{2}x^T H x + x^T c$$

subject to

$$a_i^T x = b_i, i \in E$$

$$a_i^T x \geq b_i, i \in I$$

where H is a symmetric nxn matrix, E and I are finite sets of indices, and c, x, and a_i are vectors in R^n. The matrix H is the Hessian of the objective function $q(x)$.

NOTE—Only convex QP problems are supported. A QP problem is convex if the matrix H in the objective function is positive definite.

Encapsulating the Problem

In **NMath**, class **QuadraticProgrammingProblem** class encapsulates a QP problem. The objective function is specified by providing the matrix *H* and the vector *c*. The matrix *H*, usually referred to as the *Hessian*, is the quadratic coefficient matrix. The vector *c*, sometimes referred to as the *gradient*, contains the coefficients for the linear terms.

For example, to minimize

```
q(x) = (x0 - 1)^2 + (x1 - 2.5)^2
```

Translate the objective function into the form `0.5*x'Hx + x'c`. In this case:

```
H = | 2 0 |
    | 0 2 |

c = [-2 -5]
```

This code sets up the QP problem:

Code Example – C# quadratic programming problem

```
var H = new DoubleMatrix( "2x2[2 0  0 2]" );
var c = new DoubleVector( -2.0, -5.0 );
var problem = new QuadraticProgrammingProblem( H, c );
```

Code Example – VB quadratic programming problem

```
Dim H As New DoubleMatrix("2x2[2 0  0 2]")
Dim C As New DoubleVector(-2.0, -5.0)
Dim Problem As New QuadraticProgrammingProblem(H, C)
```

Adding Bounds and Constraints

The constraints in a QP problem must be linear. There are several convenience methods provided for adding constraints and variable bounds.

For instance, given constraints:

```
-x0 + 2*x1 <= 2

 x0 - 2*x1 >= -6
-x0 + 2*x1 >= -2

 x0 >= 0
 x1 >= 0
```

The following code adds these constraints to an existing **QuadraticProgrammingProblem** object:

Code Example – C# quadratic programming problem

```
problem.AddUpperBoundConstraint(
  new DoubleVector( -1.0, 2.0 ), 2.0 );
problem.AddLowerBoundConstraint(
  new DoubleVector( 1.0, -2.0 ), -6.0 );
problem.AddLowerBoundConstraint(
  new DoubleVector( -1.0, 2.0 ), -2.0 );
problem.AddLowerBound( 0, 0 );
problem.AddLowerBound( 1, 0 );
```

Code Example – VB quadratic programming problem

```
Problem.AddUpperBoundConstraint( New DoubleVector(-1.0, 2.0), 2.0)
Problem.AddLowerBoundConstraint( New DoubleVector(1.0, -2.0), -6.0)
Problem.AddLowerBoundConstraint( New DoubleVector(-1.0, 2.0), -2.0)
Problem.AddLowerBound(0, 0)
Problem.AddLowerBound(1, 0)
```

Solving the Problem

NMath provides two classes for solving quadratic programming problems:

- Class **ActiveSetQPSolver** solves QP problems using an active set algorithm.
- Class **InteriorPointQPSolver** solves QP problems using an interior point algorithm.

Active Set

Class **ActiveSetQPSolver** solves QP problems using an active set algorithm. The active set contains a subset of inequalities to watch while searching for a solution, which reduces the complexity of the search.

Code Example – C# active set quadratic programming

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

Code Example – VB active set quadratic programming

```
Dim Solver As New ActiveSetQPSolver()
```

The `Solve()` method solves the problem, and returns `true` if the algorithm terminated successfully:

Code Example – C# active set quadratic programming

```
if ( !solver.Solve( problem ) ) {
  Console.WriteLine( "Solver failed: {0}", solver.Status );
}
else {
  Console.WriteLine("Solver found solution (x0, x1) = ({0}, {1})",
        solver.OptimalX[0], solver.OptimalX[1] );
  Console.WriteLine("After {0} iterations", solver.Iterations );
  Console.WriteLine( "Optimal objective function value = {0}",
    solver.OptimalObjectiveFunctionValue );
}
```

Code Example – VB active set quadratic programming

```
If Not Solver.Solve(Problem) Then
  Console.WriteLine("Solver failed: {0}", Solver.Status)
Else
  Console.WriteLine("Solver found solution (x0, x1) = ({0}, {1})",
    Solver.OptimalX(0), Solver.OptimalX(1))
  Console.WriteLine("After {0} iterations", Solver.Iterations)
  Console.WriteLine("Optimal objective function value = {0}",
    Solver.OptimalObjectiveFunctionValue)
End If
```

The `Solve()` method also optionally accepts a starting point for the solution search. The starting point need not be a feasible point.

Interior Point

Class **InteriorPointQPSolver** solves QP problems using an interior point algorithm.

Code Example – C# interior point quadratic programming

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

Code Example – VB interior point quadratic programming

```
Dim Solver As New InteriorPointQPSolver()
```

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

Code Example – C# interior point quadratic programming

```
var solverParams = new InteriorPointQPSolverParams
{
  KktForm = InteriorPointQPSolverParams.KktFormOption.Blended,
  Tolerance = 1e-6,
  MaxDenseColumnRatio = 0.9,
  PresolveLevel =
    InteriorPointQPSolverParams.PresolveLevelOption.Full,
  SymbolicOrdering = InteriorPointQPSolverParams.
    SymbolicOrderingOption.ApproximateMinDegree
};
```

Code Example – VB interior point quadratic programming

```
Dim SolverParams As New InteriorPointQPSolverParams
SolverParams.KktForm =
  InteriorPointQPSolverParams.KktFormOption.Blended
SolverParams.Tolerance = "1e-6"
SolverParams.MaxDenseColumnRatio = 0.9
SolverParams.PresolveLevel =
  InteriorPointQPSolverParams.PresolveLevelOption.Full
SolverParams.SymbolicOrdering = InteriorPointQPSolverParams.
  SymbolicOrderingOption.ApproximateMinDegree
```

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

Code Example – C# interior point quadratic programming

```
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 );
```

Code Example – VB interior point quadratic programming

```
Console.WriteLine("Solver Parameters:")
Console.WriteLine(SolverParams.ToString())
Console.WriteLine()
Console.WriteLine("Result = {0}", Solver.Result)
Console.WriteLine("Optimal x = {0}", Solver.OptimalX)
Console.WriteLine("Optimal Function value = {0}",
  Solver.OptimalObjectiveFunctionValue)
Console.WriteLine("iterations = {0}", Solver.IterationCount)
```

30.4 Constrained Least Squares

When least squares problems are unconstrained, they can be solved by geometric means, such as orthogonal projection. When constraints are introduced, however, nonlinear optimization techniques are required. In **NMath**, class **ConstrainedLeastSquaresProblem** encapsulates a constrained least squares problem, which can be solved using class **ConstrainedLeastSquares**. The problem is solved by reformulating as a quadratic programming problem (Section 30.3).

Encapsulating the Problem

A least squares problem solves `Cx = d` by minimizing $\|Cx - d\|^2$. Class **ConstrainedLeastSquaresProblem** encapsulates a constrained least squares problem. First construct the problem object from the matrix *C* and the vector *d*.

Code Example – C# constrained least squares

```
var C = new DoubleMatrix(
  "5x4 [0.9501    0.7620    0.6153    0.4057 " +
       "0.2311    0.4564    0.7919    0.9354 " +
       "0.6068    0.0185    0.9218    0.9169 " +
       "0.4859    0.8214    0.7382    0.4102 " +
       "0.8912    0.4447    0.1762    0.8936]" );

var d = new DoubleVector( 0.0578, 0.3528, 0.8131, 0.0098, 0.1388 );

var problem = new ConstrainedLeastSquaresProblem( C, d );
```

Code Example – VB constrained least squares

```
Dim C As New DoubleMatrix(
  "5x4 [0.9501    0.7620    0.6153    0.4057 " &
       "0.2311    0.4564    0.7919    0.9354 " &
       "0.6068    0.0185    0.9218    0.9169 " &
       "0.4859    0.8214    0.7382    0.4102 " &
       "0.8912    0.4447    0.1762    0.8936]")

Dim D As New DoubleVector(0.0578, 0.3528, 0.8131, 0.0098, 0.1388)

Dim Problem As New ConstrainedLeastSquaresProblem(C, D)
```

Adding Bounds and Constraints

Next, add the bounds and constraints. Constraints are specified using a constraint matrix, a vector of right-hand sides, and a tolerance. For example, this code adds the inequality constraints `Ax <= b` using a constraint tolerance of `0.00001`. This allows for small violations of the constraints. Specifically the constraints will be considered satisfied for a vector *x* if `Ax <= b + 0.00001`.

Code Example – C# constrained least squares

```
var A = new DoubleMatrix(
  "3x4[0.2027    0.2721    0.7467    0.4659 " +
      "0.1987    0.1988    0.4450    0.4186 " +
      "0.6037    0.0152    0.9318    0.8462]" );
var b = new DoubleVector( 0.5251, 0.2026, 0.6721 );
double constraintTolerance = 0.00001;

for ( int i = 0; i < A.Rows; i++ )
{
  problem.AddUpperBoundConstraint( A.Row( i ), b[i],
    constraintTolerance );
}
```

Code Example – VB constrained least squares

```
Dim A As New DoubleMatrix(
  "3x4[0.2027    0.2721    0.7467    0.4659 " &
      "0.1987    0.1988    0.4450    0.4186 " &
      "0.6037    0.0152    0.9318    0.8462]")
Dim B As New DoubleVector(0.5251, 0.2026, 0.6721)
Dim ConstraintTolerance As Double = 0.00001

Dim I As Integer
For I = 0 To A.Rows - 1
  Problem.AddUpperBoundConstraint(A.Row(I), B(I),
ConstraintTolerance)
Next
```

This code add variable bounds `-0.1 <= x[i] <= 2.0`.

Code Example – C# constrained least squares

```
for ( int i = 0; i < problem.NumVariables; i++ )
{
  problem.AddBounds( i, -0.10, 2.0, .00001 );
}
```

Code Example – VB constrained least squares

```
For I = 0 To Problem.NumVariables - 1
  Problem.AddBounds(I, -0.1, 2.0, 0.00001)
Next
```

Solving the Problem

ConstrainedLeastSquares uses a Quadratic Programming (QP) solver to solve the constrained least squares problem.

Code Example – C# constrained least squares

```
var solver = new ConstrainedLeastSquares();
bool success = solver.Solve( problem );
Console.WriteLine( "Success = {0}", success );
Console.WriteLine( "Solution x = {0}", solver.X );
Console.WriteLine( "Residual norm = {0}", solver.ResidualNorm );
Console.WriteLine( "Performed {0} iterations", solver.Iterations );
```

Code Example – VB constrained least squares

```
Dim Solver As New ConstrainedLeastSquares()
Dim Success As Boolean = Solver.Solve(Problem)
Console.WriteLine("Success = {0}", Success)
Console.WriteLine("Solution x = {0}", Solver.X)
Console.WriteLine("Residual norm = {0}", Solver.ResidualNorm)
Console.WriteLine("Performed {0} iterations", Solver.Iterations)
```

By default, the QP solver used is the active set solver with default options (Section 30.3). You can also pass in an instance of a QP solver for the constrained least squares class to use. This allows you to set options on the QP solver and inspect results.

Code Example – C# constrained least squares

```
var interiorPointQp = new InteriorPointQPSolver();

var solverParams = new InteriorPointQPSolverParams
{
  MaxIterations = 10000,
  PresolveLevel =
    InteriorPointQPSolverParams.PresolveLevelOption.None
};

solver.Solve( problem, interiorPointQp,
  solverParams );
Console.WriteLine( "Interior point QP result = {0}",
  interiorPointQp.Result );
```

Code Example – VB constrained least squares

```
Dim InteriorPointQp As New InteriorPointQPSolver()

Dim SolverParams = New InteriorPointQPSolverParams()
SolverParams.MaxIterations = 10000
SolverParams.PresolveLevel =
  InteriorPointQPSolverParams.PresolveLevelOption.None

Solver.Solve(Problem, InteriorPointQp, SolverParams)
Console.WriteLine("Interior point QP result = {0}",
  InteriorPointQp.Result)
```

If you use the active set QP solver you can determine which constraints are active in the solution by accessing the Lagrange multiplier property. A constraint is active if its corresponding Lagrange multiplier is nonzero.

Code Example – C# constrained least squares

```
var activeSetQP = new ActiveSetQPSolver();
solver.Solve( problem, activeSetQP );

// Print out the active constraints.
for ( int i = 0; i < activeSetQP.LagrangeMultiplier.Length; i++ )
{
  if ( activeSetQP.LagrangeMultiplier[i] != 0.0 )
  {
    Console.WriteLine( "Constraint {0} = {1} is active", i,
      problem.Constraints[i].ToString() );
  }
}
```

Code Example – VB constrained least squares

```
Dim ActiveSetQP As New ActiveSetQPSolver()
Solver.Solve(Problem, ActiveSetQP)

'' Print out the active constraints.
For I = 0 To ActiveSetQP.LagrangeMultiplier.Length - 1
  If (ActiveSetQP.LagrangeMultiplier(I) <> 0.0) Then
    Console.WriteLine("Constraint {0} = {1} is active", I,
      Problem.Constraints(I).ToString())
  End If
Next
```

CHAPTER 31.
FITTING POLYNOMIALS

As described in Chapter 8, the `CenterSpace.NMath.Core` namespace includes classes for calculating least squares fits of linear functions to a set of points. In addition, the class **PolynomialLeastSquares**, performs a least squares fit of a **Polynomial** to a set of points.

This chapter describes how to use class **PolynomialLeastSquares**.

NOTE—For testing the goodness of fit of PolynomialLeastSquares fits, see class GoodnessOfFit. Available statistics include the residual standard error, the coefficient of determination (R2 and "adjusted" R2), the F-statistic for the overall model with its numerator and denominator degrees of freedom, and standard errors, t-statistics, and finally corresponding (two-sided) p-values for the model parameters.

31.1 Creating PolynomialLeastSquares

A **PolynomialLeastSquares** is constructed from paired vectors of known *x*- and *y*-values, and the desired degree of the fitted polynomial. For example, this code fits a cubic:

Code Example – C# polynomial fit

```
int degree = 4;
var fit = new PolynomialLeastSquares( degree, x, y );
```

Code Example – VB polynomial fit

```
Dim Degree = 4
Dim Fit As New PolynomialLeastSquares(Degree, X, Y)
```

31.2 Properties of PolynomialLeastSquares

Once constructed, a **PolynomialLeastSquares** object provides the following properties:

- `FittedPolynomial` gets the fitted **Polynomial** object.
- `Coefficients` gets the coeffients of the fitted polynomial. The constant is at index `0`, and the leading coefficient is at index `Coefficients.Length - 1`.
- `Degree` gets the degree of the fitted polynomial.
- `LeastSquaresSolution` gets the **DoubleLeastSquares** object used to compute the coefficients.
- `DesignMatrix` gets the design matrix for the fit.

Finally, the `CoeffErrorEstimate()` method returns a vector of error estimates for the coefficients based on a given estimated error in the *y*-values. For example:

Code Example – C# polynomial fit

```
Console.WriteLine( fit.CoeffErrorEstimate(0.01) );
```

Code Example – VB polynomial fit

```
Console.WriteLine( Fit.CoeffErrorEstimate(0.01) )
```

CHAPTER 32. NONLINEAR LEAST SQUARES

NMath provides classes for solving nonlinear least squares problems.

Solving a nonlinear least squares problem means finding the best approximation to vector *y* with the model function that has nonlinear dependence on variables *x*, by minimizing the sum, *S*, of the squared residuals:

$$S = \sum_{i=1}^{n} r_i^2$$

where

$$r_i = y - f(x_i)$$

Unlike the linear least squares problem, non-linear least squares does not have a closed form solution, and is therefore solved by iterative refinement.

NMath provides nonlinear least squares classes for:

- solving nonlinear least squares problems, with or without linear boundary constraints, using the Trust-Region or Levenberg-Marquardt methods
- curve fitting, by finding a minimum in the curve parameter space in the sum of the squared residuals with respect to a set of data points
- surface fitting, by finding a minimum in the surface parameter space in the sum of the squared residuals with respect to a set of data points

This chapter describes how to use the nonlinear least squares classes.

32.1 Nonlinear Least Squares Interfaces

In **NMath**, classes which solve nonlinear least squares problems implement either the **INonlinearLeastSqMinimizer** interface or the **IBoundedNonlinearLeastSqMinimizer** interface.

Minimization

The **INonlinearLeastSqMinimizer** interface provides the `Minimize()` method for minimizing a given function encapsulated as a **DoubleMultiVariableFunction**, an abstract class for representing a multivariable function. Instances override the `Evaluate()` method and, optionally, the `Jacobian()` method. If the `Jacobian()` method is not overriden, a central differences approximation is used to calculate the Jacobian.

For example, this code encapsulates a function that has four input variables and twenty output variables:

Code Example – C# nonlinear least squares

```
public class MyFunction : DoubleMultiVariableFunction
{
  DoubleVector yi = new DoubleVector( 20 );
  DoubleVector ti = new DoubleVector( 20 );
  DoubleVector p = new DoubleVector( 4 );

  public MyFunction() : base(4, 20)
  {
    p[0] = -4;
    p[1] = -5;
    p[2] = 4;
    p[3] = -4;

    for ( int i = 0; i < yi.Length; i++ )
    {
      ti[i] = i;
      yi[i] = p[2]*Math.Exp( p[0]*i ) + p[3]*Math.Exp( p[1]*i );
    }
  }

  public override void Evaluate(DoubleVector x, ref DoubleVector y)
  {
    if ( x.Length != 4 || y.Length != 20 ) throw
      new InvalidArgumentException( "bad length" );

    for ( int i = 0; i < ti.Length; i++ )
    {
      y[i] = yi[i] - x[2] * Math.Exp( x[0] * ti[i] )
                   - x[3] * Math.Exp( x[1] * ti[i] );
    }
  }
}
```

Code Example – VB nonlinear least squares

```
Public Class MyFunction
  Inherits DoubleMultiVariableFunction

  Private YI As As New DoubleVector( 20 )
  Private TI As New DoubleVector(20)
  Private P As New DoubleVector(4)

  Public Sub New()
    MyBase.New(4, 20)

    P(0) = -4
    P(1) = -5
    P(2) = 4
    P(3) = -4

    For I As Integer = 0 To YI.Length - 1
      TI(I) = I
      yi(I) = P(2) * Math.Exp(P(0) * I) + P(3) * Math.Exp(P(1) * I)
    Next

  End Sub


  Public Overrides Sub Evaluate(X As DoubleVector,
    ByRef Y As DoubleVector)

    If X.Length <> 4 Or Y.Length <> 20 Then
      Throw New InvalidArgumentException("bad length")
    End If

    For I As Integer = 0 To TI.Length - 1
      Y(I) = yi(I) - X(2) * Math.Exp(X(0) * TI(I)) - X(3) *
        Math.Exp(X(1) * TI(I))
    Next

  End Sub

End Class
```

The `Minimize()` method takes:

- the function to minimize, encapsulated as a **DoubleMultiVariableFunction**
- the starting point

The **IBoundedNonlinearLeastSqMinimizer** interface extends **INonlinearLeastSqMinimizer** to provide an overload of the `Minimize()` method which also accepts lower and upper linear bounds on the solution.

Minimization Results

The `Minimize()` method returns the solution found by the minimization:

Code Example – C# nonlinear least squares

```
DoubleVector solution = minimizer.Minimize( f, start );
```

Code Example – VB nonlinear least squares

```
Dim Solution As DoubleVectorn = Minimizer.Minimize(F, Start)
```

Additional information about the last performed fit is available from properties in the **INonlinearLeastSqMinimizer** interface:

- `InitialResidual` gets the residual associated with the starting point.
- `FinalResidual` gets the residual associated with the last computed solution.
- `Iterations` gets the number of iterations used in the last computed solution.
- `MaxIterations` gets and sets the maximum number of iterations used in computing minima estimates.
- `MaxIterationsMet` returns `true` if the minimum just computed stopped because the maximum number of iterations was reached; otherwise, `false`.

For example:

Code Example – C# nonlinear least squares

```
double initialResidual = minimizer.InitialResidual;
double finalResidual = minimizer.FinalResidual;
int iterations = minimizer.Iterations;
```

Code Example – VB nonlinear least squares

```
Dim InitialResidual As Double = Minimizer.InitialResidual
Dim FinalResidual As Double = Minimizer.FinalResidual
Dim Iterations As Integer = Minimizer.Iterations
```

Implementations

NMath provides two implementations of the nonlinear least squares interfaces:

- Class **TrustRegionMinimizer** (Section 32.2) solves both constrained and unconstrained nonlinear least squares problems using the Trust-Region method, and implements the **IBoundedNonlinearLeastSqMinimizer** interface.

- Class **LevenbergMarquardtMinimizer** (Section 32.3) solves nonlinear least squares problems using the Levenberg-Marquardt method, and implements the **INonlinearLeastSqMinimizer** interface.

32.2 Trust-Region Minimization

NMath provides class **TrustRegionMinimizer** for solving both constrained and unconstrained nonlinear least squares problems using the Trust-Region method. **TrustRegionMinimizer** implements the **IBoundedNonlinearLeastSqMinimizer** interface.

The Trust-Region method maintains a region around the current search point where a quadratic model is “trusted” to be correct. If an adequate model of the objective function is found within the trust region, the region is expanded. Otherwise, the region is contracted.

The Trust-Region algorithm requires the partial derivatives of the function, but a numerical approximation may be used if the closed form is not available.

Constructing a TrustRegionMinimizer

Instances of **TrustRegionMinimizer** are constructed by specifying an error tolerance and a maximum number of iterations, or by accepting the defaults for these values. For example, this code constructs a **TrustRegionMinimizer** using the default tolerance and a maximum of 1000 iterations:

Code Example – C# trust region minimization

```
int iter = 1000;
var minimizer = new TrustRegionMinimizer( iter );
```

Code Example – VB nonlinear least squares

```
Dim Iter As Integer = 1000
Dim Minimizer As New TrustRegionMinimizer(Iter)
```

Minimization

Class **TrustRegionMinimizer** provides the `Minimize()` method for minimizing a given multivariable function. Functions may be multidimensional in both their domain, *x*, and range, *y*.

The `Minimize()` method takes:

- the function, *f*, to minimize, encapsulated as a **DoubleMultiVariableFunction**, as described in Section 32.1
- the starting point
- (optionally) lower and upper bounds on the solution

NOTE—The dimensionality of y must be greater than or equal to the dimensionality of x, or the least squares problem is under constrained.

Thus, this code minimizes the function **MyFunction**, starting at the specified point:

Code Example – C# trust region minimization

```
public class MyFunction : DoubleMultiVariableFunction
{
  public MyFunction() : base(4,4) {;}

  public override void Evaluate( DoubleVector x,
                                 ref DoubleVector y )
  {
    for (int i = 0; i < (x.Length) / 4; i++)
    {
      y[4 * i] = x[4 * i] + 10.0 * x[4 * i + 1];
      y[4 * i + 1] = 2.2360679774997896964091736687313 *
        (x[4 * i + 2] - x[4 * i + 3]);
      y[4 * i + 2] = (x[4 * i + 1] - 2.0 * x[4 * i + 2]) *
        (x[4 * i + 1] - 2.0 * x[4 * i + 2]);
      y[4 * i + 3] = 3.1622776601683793319988935444327 *
        (x[4 * i] - x[4 * i + 3]) * (x[4 * i] - x[4 * i + 3]);
    }
  }
}

var f = new MyFunction();
var start = new DoubleVector("3.0 -1.0 0.0 1.0");

var minimizer = new TrustRegionMinimizer();
DoubleVector solution = minimizer.Minimize( f, start );
```

Code Example – VB trust region minimization

```
Public Class MyFunction
  Inherits DoubleMultiVariableFunction

  Public Sub New()
    MyBase.New(4, 4)
  End Sub

  Public Overrides Sub Evaluate(X As DoubleVector, ByRef Y As
DoubleVector)

    For I As Integer = 0 To (X.Length / 4) - 1
      Y(4 * I) = X(4 * I) + 10.0 * X(4 * I + 1)
      Y(4 * I + 1) = 2.23606797749979 *
       (X(4 * I + 2) - X(4 * I + 3))
      Y(4 * I + 2) = (X(4 * I + 1) - 2.0 *
        X(4 * I + 2)) * (X(4 * I + 1) - 2.0 * X(4 * I + 2))
      Y(4 * I + 3) = 3.1622776601683795 *
       (X(4 * I) - X(4 * I + 3)) * (X(4 * I) - X(4 * I + 3))
    Next
  End Sub

End Class

Dim F As New MyFunction()
Dim Start As New DoubleVector("3.0 -1.0 0.0 1.0")

Dim Minimizer As New TrustRegionMinimizer()
Dim Solution As DoubleVector = Minimizer.Minimize(F, Start)
```

Since problems can have multiple local minima, trying different starting points is recommended for better solutions.

NOTE—The Trust-Region algorithm requires the partial derivatives of the function being minimized. A numerical approximation is used by default, but you can also implement the Jacobian() method on your DoubleMultiVariableFunction.

Linear Bound Constraints

The `Minimize()` method also accepts linear bound constraints on the solution, such that:

$$\text{lower}_i \leq x_i \leq \text{upper}_i, \quad i = 1, \ldots, n$$

For instance, this code specifies lower and upper bounds:

Code Example – C# trust region minimization

```
var f = new MyFunction();

var start = new DoubleVector("3.0 -1.0 0.0 1.0");
var lowerBounds = new DoubleVector("0.1 -20.0 -1.0 -1.0");
var upperBounds = new DoubleVector("100.0 20.0 1.0 50.0");

var minimizer = new TrustRegionMinimizer();
DoubleVector solution = minimizer.Minimize( f, start, lowerBounds,
  UpperBounds );
```

Code Example – VB trust region minimization

```
Dim F As New MyFunction()

Dim Start As New DoubleVector("3.0 -1.0 0.0 1.0")
Dim LowerBounds As New DoubleVector("0.1 -20.0 -1.0 -1.0")
Dim UpperBounds As New DoubleVector("100.0 20.0 1.0 50.0")

Dim Minimizer As New TrustRegionMinimizer()
Dim Solution As DoubleVector = Minimizer.Minimize(F, Start,
  LowerBounds, UpperBounds)
```

Minimization Results

The `Minimize()` method returns the solution found by the minimization:

Code Example – C# trust region minimization

```
DoubleVector solution = minimizer.Minimize( f, start );
```

Code Example – VB trust region minimization

```
Dim Solution As DoubleVector = Minimizer.Minimize(F, Start)
```

Additional information about the last performed fit is available from properties implemented as part of the **INonlinearLeastSqMinimizer** interface (Section 32.1). Class **TrustRegionMinimizer** also provides property `StopCriterion` which return the reason for stopping. The stopping criterion is returned as a value from the `TrustRegionMinimizer.Criterion` enumeration, shown in Table 22.

Table 22 – Stopping Criterion

Criterion	Description
MaxIterationsExceeded	The maximum number of iterations was exceeded.
TrustRegionWithinTolerance	The area of the trust region was within tolerance.
FunctionValueWithinTolerance	The function value was within tolerance.
JacobianWithinTolerance	The value of the Jacobian matrix, *A*, at *x* was within tolerance for all *A[i,j]*.
TrialStepWithinTolerance	The size of the trial step was within tolerance.
ImprovementWithinTolerance	The magnitude of the improvement between steps was within tolerance. The magnitude of the improvement between steps is $\|\|F(x)\|\|- \|\|F(x) - A(x)s\|\|$, where *F(x)* is the value of the function at *x*, *A* is the Jacobian matrix, and *s* is the trial step.

Note that by default, the general tolerance supplied when your construct a **TrustRegionMinimizer** instance is used for all tolerance-related stopping criteria. However, tolerances can also be specified individually for each criterion. For example, this code sets the trial step tolerance to `1e-12`:

Code Example – C# trust region minimization

```
minimizer.ToleranceTrialStep = 1e-12;
```

Code Example – VB trust region minimization

```
Minimizer.ToleranceTrialStep = "1e-12"
```

The `SetAllTolerances()` method can be used after construction to set all tolerances to the same value.

32.3 Levenberg-Marquardt Minimization

NMath provides class **LevenbergMarquardtMinimizer** for solving nonlinear least squares problems using the Levenberg-Marquardt method.

LevenbergMarquardtMinimizer implements the **INonlinearLeastSqMinimizer** interface.

Constructing a LevenbergMarquardtMinimizer

Instances of **LevenbergMarquardtMinimizer** are constructed by specifying a maximum number of iterations, gradient tolerance, and a solution tolerance, or by accepting the defaults for these values. Iteration stops when the infinity norm of the gradient used in calculating the next step falls below the gradient tolerance, or then the L2 norm of the step size falls below the solution tolerance. For example:

Code Example – C# Levenberg-Marquardt minimization

```
int maxIterations = 1000;
double gradientTolerance = 1e-14;
double solutionTolerance = 1e-14;
var lm = new LevenbergMarquardtMinimizer(
  maxIterations, gradientTolerance, solutionTolerance );
```

Code Example – VB Levenberg-Marquardt minimization

```
Dim MaxIterations As Integer = 1000
Dim GradientTolerance As Double = "1e-14"
Dim SolutionTolerance As Double = "1e-14"
Dim LM As New LevenbergMarquardtMinimizer(MaxIterations,
  GradientTolerance, SolutionTolerance)
```

Minimization

Class **LevenbergMarquardtMinimizer** provides the `Minimize()` method for minimizing a given multivariable function, encapsulated as a **DoubleMultiVariableFunction**, as described in Section 32.1.

Minimization Results

The `Minimize()` method returns the solution found by the minimization:

Code Example – C# Levenberg-Marquardt minimization

```
DoubleVector solution = minimizer.Minimize( f, start );
```

Code Example – VB Levenberg-Marquardt minimization

```
Dim Solution As DoubleVector = Minimizer.Minimize(F, Start)
```

Additional information about the last performed fit is available from properties implemented as part of the **INonlinearLeastSqMinimizer** interface (Section 32.1).

32.4 Nonlinear Least Squares Curve Fitting

NMath provides classes **OneVariableFunctionFitter** and **BoundedOneVariableFunctionFitter** for fitting *generalized* one variable functions to a set of points. In the space of the function parameters, beginning at a specified starting point, these classes finds a minimum (possibly local) in the sum of the squared residuals with respect to a set of data points. Minimization is performed by an implementation of the **INonlinearLeastSqMinimizer** or **IBoundedNonlinearLeastSqMinimizer** interface (Section 32.1), respectively. You must supply at least as many data points to fit as your function has parameters.

BoundedOneVariableFunctionFitter derives from **OneVariableFunctionFitter**, and accepts linear bounds on the solution.

Generalized One Variable Functions

A one variable function takes a single double *x*, and returns a double *y*:

$$y = f(x)$$

A *generalized* one variable function additionally takes a set of parameters, *p*, which may appear in the function expression in arbitrary ways:

$$y = f(p_1, p_2, \ldots, p_n; x)$$

For example, this code computes $y = a\sin(bx + c)$:

Code Example – C# nonlinear least squares fit

```
public double MyFunction( DoubleVector p, double x )
{
  return p[0] * Math.Sin( p[1] * x + p[2] );
}
```

Code Example – VB nonlinear least squares fit

```
Public Function MyFunction(P As DoubleVector, X As Double) As
  Double
  Return P(0) * Math.Sin(P(1) * X + P(2))
End Function
```

Encapsulating One Variable Functions

In **NMath**, generalized one variable functions can be encapsulated in two ways:

- By extending the abstract class **DoubleParameterizedFunction**, and implementing the `Evaluate()` method. The `GradientWithRespectToParams()` can also be implemented to compute the gradient with respect to the parameters; otherwise, a numerical approximation is used.
- By wrapping a `Func<DoubleVector, double, double>` delegate in a **DoubleParameterizedDelegate**. An `Action<DoubleVector, double, DoubleVector>` delegate can also be provided for computing the gradient with respect to the parameters; otherwise a numerical approximation is used.

For example, this code encapsulates $y = a\sin(bx + c)$ using a **DoubleParameterizedFunction**:

Code Example – C# nonlinear least squares fit

```
public class MyFunction : DoubleParameterizedFunction
{
  public MyFunction()
  {}

  public override double Evaluate( DoubleVector p, double x )
  {
    return p[0] * Math.Sin( p[1] * x + p[2] );
  }
}

DoubleParameterizedFunction f = new MyFunction();
```

Code Example – VB nonlinear least squares fit

```
Public Class MyFunction
  Inherits DoubleParameterizedFunction

  Public Sub New()
  End Sub

  Public Overrides Function Evaluate(P As DoubleVector,
    X As Double) As Double
    Return P(0) * Math.Sin(P(1) * X + P(2))
  End Function

End Class

Dim F As DoubleParameterizedFunction = New MyFunction()
```

This code encapsulates the same function using a **DoubleParameterizedDelegate**:

Code Example – C# nonlinear least squares fit

```
public double MyFunction( DoubleVector p, double x )
{
  return p[0] * Math.Sin( p[1] * x + p[2] );
}

var f = new DoubleParameterizedDelegate( MyFunction );
```

Code Example – VB nonlinear least squares fit

```
Public Function MyFunction(P As DoubleVector, X As Double) As
  Double
  Return P(0) * Math.Sin(P(1) * X + P(2))
End Function

Dim F As New DoubleParameterizedDelegate(AddressOf MyFunction)
```

This code demonstrates implementing `GradientWithRespectToParams()` as well as `Evaluate()` in a **DoubleParameterizedFunction** which encapsulates $y = a\cos(bx) + b\sin(ax)$:

Code Example – C# nonlinear least squares fit

```
public class MyFunction : DoubleParameterizedFunction
{
  public MyFunction()
  {}

  public override double Evaluate( DoubleVector p, double x )
  {
    double a = p[0];
    double b = p[1];
    return a*Math.Cos( b*x ) + b*Math.Sin( a*x );
  }

  public override void GradientWithRespectToParams( DoubleVector p,
    double x, ref DoubleVector grad )
  {
    double a = p[0];
    double b = p[1];
    grad[0] = Math.Cos( b*x ) + b*x*Math.Cos( a*x );
    grad[1] = -a*x*Math.Sin( b*x ) + Math.Sin( a*x );
  }
}
```

Code Example – VB nonlinear least squares fit

```
Public Class MyFunction
  Inherits DoubleParameterizedFunction

  Public Sub New()
  End Sub

  Public Overrides Function Evaluate(P As DoubleVector, X As
    Double) As Double
    Dim A As Double = P(0)
    Dim B As Double = P(1)
    Return a * Math.Cos(b * x) + b * Math.Sin(a * x)
  End Function

  Public Overrides Sub GradientWithRespectToParams(P As
    DoubleVector, X As Double, ByRef Grad As DoubelVector)
    Dim A As Double = P(0)
    Dim B As Double = P(1)
    Grad(0) = Math.Cos(B * X) + B * X * Math.Cos(A * X)
    Grad(1) = -A * X * Math.Sin(B * X) + Math.Sin(A * X)
  End Sub
End Class
```

Predefined Functions

For convenience, class **AnalysisFunctions** includes a selection of common generalized one variable functions, as shown in Table 23.

Table 23 – Predefined Generalized One Variable Functions

Delegate	Function
TwoParameterAsymptotic	$y = a + \frac{b}{x}$
ThreeParameterExponential	$y = ae^{bx} + c$
ThreeParameterSine	$y = a\sin(bx + c)$
FourParameterLogistic	$y = d + \frac{a - d}{1 + \left(\frac{x}{c}\right)^b}$
FiveParameterLogistic	$y = d + \frac{a - d}{\left[1 + \left(\frac{x}{c}\right)^b\right]^g}$

Instances of **DoubleParameterizedDelegate** can be constructed from these functions. For example:

Code Example – C# nonlinear least squares fit

```
var f = new DoubleParameterizedDelegate(
  AnalysisFunctions.FourParameterLogistic );
```

Code Example – VB nonlinear least squares fit

```
Dim F As New DoubleParameterizedDelegate(
  AnalysisFunctions.FourParameterLogistic)
```

Constructing a OneVariableFunctionFitter

Class **OneVariableFunctionFitter** is templatized on **INonlinearLeastSqMinimizer**, and **BoundedOneVariableFunctionFitter** is templatized on **IBoundedNonlinearLeastSqMinimizer** (Section 32.1). Instances are constructed from an encapsulated, generalized one variable function. For example, this code uses one of the predefined curves in **AnalysisFunctions**:

Code Example – C# nonlinear least squares fit

```
var f = new DoubleParameterizedDelegate(
  AnalysisFunctions.FourParameterLogistic );

var fitter =
  new OneVariableFunctionFitter<TrustRegionMinimizer>( f );
```

Code Example – VB nonlinear least squares fit

```
Dim F As New DoubleParameterizedDelegate(
  AnalysisFunctions.FourParameterLogistic)

Dim Fitter As New OneVariableFunctionFitter(
  Of TrustRegionMinimizer)(F)
```

As a convenience, there is a constructor that takes a `Func<DoubleVector, double, double>` delegate directly:

Code Example – C# nonlinear least squares fit

```
BoundedOneVariableFunctionFitter<TrustRegionMinimizer> fitter =
  new BoundedOneVariableFunctionFitter<TrustRegionMinimizer>(
    AnalysisFunctions.FourParameterLogistic );
```

Code Example – VB nonlinear least squares fit

```
Dim Fitter As New BoundedOneVariableFunctionFitter(
  Of TrustRegionMinimizer)(AnalysisFunctions.FourParameterLogistic)
```

An existing minimizer instance can also be passed to the constructor:

Code Example – C# nonlinear least squares fit

```
var minimizer = new LevenbergMarquardtMinimizer();
minimizer.GradientTolerance = 1e-6;

var fitter =
  new OneVariableFunctionFitter<LevenbergMarquardtMinimizer>(
    AnalysisFunctions.FourParameterLogistic, minimizer );
```

Code Example – VB nonlinear least squares fit

```
Dim Minimizer As New LevenbergMarquardtMinimizer()
Minimizer.GradientTolerance = "1e-6"

Dim Fitter As New OneVariableFunctionFitter(
  Of LevenbergMarquardtMinimizer)(
    AnalysisFunctions.FourParameterLogistic, Minimizer)
```

Fitting Data

Once you've constructed an instance of **OneVariableFunctionFitter** or **BoundedOneVariableFunctionFitter** containing a function, you can fit that function to a set of points using the `Fit()` method.

The `Fit()` method on **OneVariableFunctionFitter** takes vectors of *x* and *y* values representing the data points, and a starting position in the function parameter space. For instance:

Code Example – C# nonlinear least squares fit

```
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 );
```

Code Example – VB nonlinear least squares fit

```
Dim X As New DoubleVector(0.0, 0.0, 0.0, 0.0, 0.0,
                          0.0, 0.94, 0.94, 0.94, 1.88,
                          1.88, 1.88, 3.75, 3.75, 3.75,
                          7.5, 7.5, 7.5, 15.0, 15.0,
                          15.0, 30.0, 30.0, 30.0)

Dim Y As New DoubleVector(7.58, 8.0, 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)

Dim Start As New DoubleVector("0.1 0.1 0.1 0.1")

Dim Solution As DoubleVector = Fitter.Fit(X, Y, Start)
```

In the space of the function parameters, beginning at a specified `start` point, `Fit()` finds a minimum (possibly local) in the sum of the squared residuals with respect to the given `x` and `y` values.

NOTE—You must supply at least as many data points to fit as your function has parameters.

The `Fit()` method on **BoundedOneVariableFunctionFitter** additionally accepts linear bounds on the solution:

Code Example – C# nonlinear least squares fit

```
var lowerBounds = new DoubleVector( 1.1, 1.8 );
var upperBounds = new DoubleVector( 2.1, 3.9 );
DoubleVector solution =
  fitter.Fit( x, y, start, lowerBounds, upperBounds );
```

Code Example – VB nonlinear least squares fit

```
Dim LowerBounds As New DoubleVector(1.1, 1.8)
Dim UpperBounds As New DoubleVector(2.1, 3.9)
Dim Solution As DoubleVector = fitter.Fit(X, Y, Start, LowerBounds,
  UpperBounds)
```

Trying different initial starting points is recommended for better solutions. If possible, use starting points based on *a priori* information about the curve shape and the data being fit. Otherwise, random value close to zero are usually a good choice.

Fit Results

The `Fit()` method returns the solution found by the minimization. To compute the residuals relative to the data points at the solution, use the `ResidualVector()` method:

Code Example – C# nonlinear least squares fit

```
DoubleVector residuals = fitter.ResidualVector( x, y, solution );
```

Code Example – VB nonlinear least squares fit

```
Dim Residuals As DoubleVector =
  fitter.ResidualVector(X, Y, solution)
```

Additional information about the last performed fit is available from the underlying minimizer instance, accessible using the `Minimizer` property. For example, this code gets the sum of the squared residuals at the starting point and at the solution, the number of iterations performed, and the stop criterion:

Code Example – C# nonlinear least squares fit

```
INonlinearLeastSqMinimizer minimizer = fitter.Minimizer;

double initialResidual = minimizer.InitialResidual;
double finalResidual = minimizer.FinalResidual;
int iterations = minimizer.Iterations;
```

Code Example – VB nonlinear least squares fit

```
Dim Minimizer As INonlinearLeastSqMinimizer = Fitter.Minimizer

Dim InitialResidual As Double = Minimizer.InitialResidual
Dim FinalResidual As Double = Minimizer.FinalResidual
Dim Iterations As Integer = Minimizer.Iterations
```

NOTE—For testing the goodness of fit of OneVariableFunctionFitter solutions, see class GoodnessOfFit. Available statistics include the residual standard error, the coefficient of determination (R2 and "adjusted" R2), the F-statistic for the overall model with its numerator and denominator degrees of freedom, and standard errors, t-statistics, and finally corresponding (two-sided) p-values for the model parameters.

32.5 Nonlinear Least Squares Surface Fitting

NMath provides classes **MultiVariableFunctionFitter** and **BoundedMultiVariableFunctionFitter** for fitting *generalized* multivariable functions to a set of points. The interface is analogous to **OneVariableFunctionFitter** and **BoundedOneVariableFunctionFitter** (Section 32.4), with only a couple changes to accommodate multivariate data. Again, you must supply at least as many data points to fit as your function has parameters.

Generalized Multivariable Functions

A multivariable function takes a vector of *x* values, and returns a double *y*:

$$y = f(x_1, x_2, \ldots, x_n)$$

A *generalized* multivariable function additionally takes a set of parameters, *p*, which may appear in the function expression in arbitrary ways:

$$y = f(p_1, p_2, \ldots, p_m; x_1, x_2, \ldots, x_n)$$

For example, this code computes $y = ax_1^2x_2 + b\sin(x_1) + cx_2^3$:

Code Example – C# nonlinear least squares surface fit

```
public double MyFunction( DoubleVector p, DoubleVector x )
{
  return p[0] * Math.Pow( x[0], 2.0 ) * x[1] +
         p[1] * Math.Sin( x[0] ) +
         p[2] * Math.Pow( x[1], 3.0 );
};
```

Code Example – VB nonlinear least squares surface fit

```
Public Function MyFunction(P As DoubleVector, X As DoubleVector) As
  Double
  Return P(0) * Math.Pow(X(0), 2.0) * X(1) +
         P(1) * Math.Sin(X(0)) +
         P(2) * Math.Pow(X(1), 3.0)
End Function
```

Encapsulating Generalized Multivariable Functions

In **NMath**, generalized multivariable functions can be encapsulated in two ways:

- By extending the abstract class **DoubleParameterizedFunctional**, and implementing the `Evaluate()` method. The `GradientWithRespectToParams()` can also be implemented to compute the gradient with respect to the parameters; otherwise, a numerical approximation is used.
- By wrapping a `Func<DoubleVector, DoubleVector, double>` delegate in a **DoubleVectorParameterizedDelegate**. An `Action<DoubleVector, DoubleVector, DoubleVector>` delegate can also be provided for computing the gradient with respect to the parameters; otherwise a numerical approximation is used.

For example, this code encapsulates a multivariable function using a **DoubleParameterizedFunctional**:

Code Example – C# nonlinear least squares surface fit

```
public class MyFunction : DoubleParameterizedFunctional
{
  public MyFunction()
    : base (2)
  {}

  public override double Evaluate( DoubleVector p, DoubleVector x )
  {
    // z = ayx^2 + bsin(x) + cy^3
    return p[0] * x[0] * Math.Pow( x[1], 2.0 ) +
           p[1] * Math.Sin( x[0] ) +
           p[2] * Math.Pow( x[1], 3.0 );
  }
}

DoubleParameterizedFunctional f = new MyFunction();
```

Code Example – VB nonlinear least squares surface fit

```
Public Class MyFunction
  Inherits DoubleParameterizedFunctional

  Sub New()
    MyBase.New(2)
  End Sub

  Public Overrides Function Evaluate(P As DoubleVector, X As
    DoubleVector) As Double
    ' z = ayx^2 + bsin(x) + cy^3
    Return P(0) * X(0) * Math.Pow(X(1), 2.0) +
           P(1) * Math.Sin(X(0)) +
           P(2) * Math.Pow(X(1), 3.0)
  End Function

End Class

DoubleParameterizedFunctional F As New MyFunction()
```

This code encapsulates the same function using a **DoubleVectorParameterizedDelegate**:

Code Example – C# nonlinear least squares surface fit

```
public double MyFunction( DoubleVector p, DoubleVector x )
{
    // z = ayx^2 + bsin(x) + cy^3
    return p[0] * x[0] * Math.Pow( x[1], 2.0 ) +
           p[1] * Math.Sin( x[0] ) +
           p[2] * Math.Pow( x[1], 3.0 );
}

var f = new DoubleVectorParameterizedDelegate( MyFunction );
```

Code Example – VB nonlinear least squares surface fit

```
Public Function MyFunction(P As DoubleVector, X As DoubleVector) As
  Double
  Return P(0) * Math.Pow(X(0), 2.0) * X(1) +
         P(1) * Math.Sin(X(0)) +
         P(2) * Math.Pow(X(1), 3.0)
End Function

Dim F As New DoubleVectorParameterizedDelegate( MyFunction )
```

Constructing a MultiVariableFunctionFitter

Class **MultiVariableFunctionFitter** is templatized on **INonlinearLeastSqMinimizer**, and class **BoundedMultiVariableFunction** is templatized on **IBoundedNonlinearLeastSqMinimizer** (Section 32.1). Instances are constructed from an encapsulated generalized multivariable function. For example:

Code Example – C# nonlinear least squares surface fit

```
Func<DoubleVector, DoubleVector, double> myDelegate =
  delegate( DoubleVector p, DoubleVector x )
  {
    // z = ayx^2 + bsin(x) + cy^3
    return p[0] * x[0] * Math.Pow( x[1], 2.0 ) +
           p[1] * Math.Sin( x[0] ) +
           p[2] * Math.Pow( x[1], 3.0 );
  };

DoubleVectorParameterizedDelegate f =
  new DoubleVectorParameterizedDelegate( myDelegate );

MultiVariableFunctionFitter<TrustRegionMinimizer> fitter =
  new MultiVariableFunctionFitter<TrustRegionMinimizer>( f );
```

Again, an existing minimizer instance can also be passed to the constructor:

Code Example – C# nonlinear least squares surface fit

```
var minimizer = new LevenbergMarquardtMinimizer();
minimizer.DefaultTolerance = 1e-6;

var fitter =
  new MultiVariableFunctionFitter<LevenbergMarquardtMinimizer>(
    f, minimizer );
```

Code Example – VB nonlinear least squares surface fit

```
Dim Minimizer As New LevenbergMarquardtMinimizer()
Minimizer.GradientTolerance = "1e-6"

Dim Fitter As New MultiVariableFunctionFitter(
  Of LevenbergMarquardtMinimizer)(F, Minimizer)
```

Fitting Data

Once you've constructed an instance of **MultiVariableFunctionFitter** or **BoundedMultiVariableFunctionFitter** containing a function, you can fit that function to a set of points using the `Fit()` method.

The `Fit()` method on **MultiVariableFunctionFitter** takes a **DoubleMatrix** of *x* values, where each *row* in the matrix represents a point, a **DoubleVector** of *y* values representing the data points, and a starting position in the function parameter space. For instance:

Code Example – C# nonlinear least squares surface fit

```
var x = new DoubleMatrix(10, 2);
x[Slice.All, 0] = new DoubleVector("3.6 7.7 9.3 4.1 8.6
                                    2.8 1.3 7.9 10.0 5.4");
x[Slice.All, 1] = new DoubleVector("16.5 150.6 263.1 24.7 208.5
                                    9.9 2.7 163.9 325.0 54.3");

var y = new DoubleVector("95.09 23.11 60.63 48.59 89.12
                          76.97 45.68 1.84 82.17 44.47");

var start = new DoubleVector("10 10 10");

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

Code Example – VB nonlinear least squares surface fit

```
Dim X As New DoubleMatrix(10, 2)
X(Slice.All, 0) = New DoubleVector("3.6 7.7 9.3 4.1 8.6" & _
                                    "2.8 1.3 7.9 10.0 5.4")
X(Slice.All, 1) = New DoubleVector("16.5 150.6 263.1 24.7 208.5" &
_
                                        "9.9 2.7 163.9 325.0 54.3")

Dim Y As New DoubleVector("95.09 23.11 60.63 48.59 89.12" & _
                          "76.97 45.68 1.84 82.17 44.47")

Dim Start As New DoubleVector("10 10 10")

Dim Solution As DoubleVector = Fitter.Fit(X, Y, Start)
```

In the space of the function parameters, beginning at a specified `start` point, `Fit()` finds a minimum (possibly local) in the sum of the squared residuals with respect to the given `x` and `y` values.

NOTE—You must supply at least as many data points to fit as your function has parameters.

The `Fit()` method on **BoundedMultiVariableFunctionFitter** additionally accepts linear bounds on the solution:

Code Example – C# nonlinear least squares surface fit

```
var lowerBounds = new DoubleVector( "[0 -18 0]" );
var upperBounds = new DoubleVector( "[.007 -3 1]" );
DoubleVector solution =
  fitter.Fit( x, y, start, lowerBounds, upperBounds );
```

Code Example – VB nonlinear least squares surface fit

```
Dim LowerBounds As New DoubleVector("[0 -18 0]")
Dim UpperBounds As New DoubleVector("[.007 -3 1]")
Dim Solution As DoubleVector =
  Fitter.Fit(X, Y, Start, LowerBounds, UpperBounds)
```

Trying different initial starting points is recommended for better solutions. If possible, use starting points based on *a priori* information about the curve shape and the data being fit. Otherwise, random value close to zero are usually a good choice.

Fit Results

The `Fit()` method returns the solution found by the minimization. To compute the residuals relative to the data points at the solution, use the `ResidualVector()` method:

Code Example – C# nonlinear least squares surface fit

```
DoubleVector residuals = fitter.ResidualVector( x, y, solution );
```

Code Example – VB nonlinear least squares surface fit

```
Dim Residuals As DoubleVector =
  Fitter.ResidualVector(X, Y, solution)
```

Additional information about the last performed fit is available from the underlying minimizer instance, accessible using the `Minimizer` property. For example, this code gets the sum of the squared residuals at the starting point and at the solution, the number of iterations performed, and the stop criterion:

Code Example – C# nonlinear least squares surface fit

```
INonlinearLeastSqMinimizer minimizer = fitter.Minimizer;

double initialResidual = minimizer.InitialResidual;
double finalResidual = minimizer.FinalResidual;
int iterations = minimizer.Iterations;
```

Code Example – VB nonlinear least squares surface fit

```
Dim Minimizer As INonlinearLeastSqMinimizer = Fitter.Minimizer

Dim InitialResidual As Double = Minimizer.InitialResidual
Dim FinalResidual As Double = Minimizer.FinalResidual
Dim Iterations As Integer = Minimizer.Iterations
```

CHAPTER 33.

FINDING ROOTS OF UNIVARIATE FUNCTIONS

NMath includes classes for finding roots of univariate functions. A root-finding algorithm finds a value x for a given function f, such that $f(x) = 0$.

All **NMath** root-finding classes derive from the abstract base class **RootFinderBase**. The interface and behavior is the same as for **MinimizerBase** (Section 26.1)—iteration stops when either the decrease in function value is less than a specified error tolerance, or the specified maximum number of iterations is reached. The root-finding classes also implement one of the following interfaces:

- Classes that implement the **IOneVariableRootFinder** interface require only function evaluations to find roots.
- Classes that implement the **IOneVariableDRootFinder** interface also require evaluations of the derivative of a function.

This chapter describes how to use the root-finding classes.

33.1 Finding Function Roots Without Calculating the Derivative

NMath provides several classes that implement the **IOneVariableRootFinder** interface, and find roots of univariate functions using only function evaluations:

- Class **SecantRootFinder** finds roots of univariate functions using the *secant method*. The secant method assumes that the function is approximately linear in the local region of interest and uses the zero-crossing of the line connecting the limits of the interval as an estimate of the root. The function is evaluated at the estimate, a new line is formed, and the process is repeated.

- Class **RiddersRootFinder** finds roots of univariate functions using *Ridders' Method*. Ridders' Method first evaluates the function at the midpoint of the interval, then factors out the unique exponential function which turns the residual function into a straight line.

- Class **FZero** finds roots of univariate functions using the `zeroin()` root finder published originally in *Computer Methods for Mathematical Computations* by Forsythe, Malcolm and Moler in 1977. This class is similar to MATLAB's f`zero()` function.

Instances are constructed by specifying an error tolerance and a maximum number of iterations, or by accepting the defaults for these values. For example, this code constructs a **SecantRootFinder** using the default tolerance and a maximum of `50` iterations:

Code Example – C# root finding

```
int maxIter = 50;
var finder = new SecantRootFinder( maxIter );
```

Code Example – VB root finding

```
Dim MaxIter As Integer = 50
Dim Finder As New SecantRootFinder(MaxIter)
```

Instances provide `Find()` methods for minimizing a given function within a given interval. For instance, the cosine function has a root at $\pi/2$:

Code Example – C# root finding

```
var f = new OneVariableFunction(
  new Func<double, double>( Math.Cos ) );

var finder = new RiddersRootFinder();
double lower = 0;
double upper = Math.PI;
double root = finder.Find( f, lower, upper );
```

Code Example – VB root finding

```
Dim F As New OneVariableFunction(
  New Func(Of Double, Double)(AddressOf Math.Cos))

Dim Finder As New RiddersRootFinder()
Dim Lower As Double = 0
Dim Upper As Double = Math.PI
Dim Root As Double = Finder.Find(F, Lower, Upper)
```

33.2 Finding Function Roots of Derivable Functions

Class **NewtonRalphsonRootFinder** implements the **IOneVariableDRootFinder** interface and finds roots of univariate functions using the *Newton-Raphson Method*. The Newton-Raphson algorithm finds the slope of the function at the current point and uses the zero of the tangent line as an estimate of the root.

Like **SecantRootFinder** and **RiddersRootFinder** (Section 33.1), instances of **NewtonRalphsonRootFinder** are constructed by specifying an error tolerance and a maximum number of iterations, or by accepting the defaults for these values. For example:

Code Example – C# root finding

```
double tol = 1e-8;
int maxIter = 100;
var finder = new NewtonRaphsonRootFinder( tol, maxIter );
```

Code Example – VB root finding

```
Dim Tol As Double = "1e-8"
Dim MaxIter As Integer = 100
Dim Finder As New NewtonRaphsonRootFinder(Tol, MaxIter)
```

Once you have constructed a **NewtonRalphsonRootFinder** instance, you can use the `Find()` method to find a root within a given interval. 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)`:

Code Example – C# root finding

```
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 );
```

Code Example – VB root finding

```
Dim P As New Polynomial(New DoubleVector(-2.0, -5.0, 9.0, -2.0))
Dim Finder As New NewtonRaphsonRootFinder()
Dim Lower As Double = 0
Dim Upper As Double = 3
Dim Root As Double = Finder.Find(P, P.Derivative(), Lower, Upper)
```

CHAPTER 34.

INTEGRATING MULTIVARIABLE FUNCTIONS

The `CenterSpace.NMath.Core` namespace includes classes for computing an approximation of the integral of a **OneVariableFunction** over some interval (Chapter 13). These classes include **RombergIntegrator** and **GaussKronrodIntegrator**, which implement the **IIntegrator** interface.

Also the class **TwoVariableIntegrator** 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.

The chapter describes how to use class **TwoVariableIntegrator**.

34.1 Creating TwoVariableIntegrators

A **TwoVariableIntegrator** has two instances of **IIntegrator**: one for the *x* dimension, and one for the *y* dimension. This code constructs a **TwoVariableIntegrator** with the default univariate integrators:

Code Example – C# integration

```
var integrator = new TwoVariableIntegrator();
```

Code Example – VB integration

```
Dim Integrator As New TwoVariableIntegrator()
```

Instances of **GaussKronrodIntegrator** are used by default. Alternatively, you can provide non-default univariate integrators:

Code Example – C# integration

```
var gauss1 = new GaussKronrodIntegrator();
var gauss2 = new GaussKronrodIntegrator();
gauss2.Tolerance = 1e-6;
var integrator = new TwoVariableIntegrator( gauss1, gauss2 );
```

Code Example – VB integration

```
Dim Gauss1 As New GaussKronrodIntegrator()
Dim Gauss2 As New GaussKronrodIntegrator()
Gauss2.Tolerance = "1e-6"
```

```
Dim Integrator As New TwoVariableIntegrator(Gauss1, Gauss2)
```

Class **TwoVariableIntegrator** also provides properties `DxIntegrator` and `DyIntegrator` for getting and setting the *x* and *y* univariate integrators on a **TwoVariableIntegrator** instance post-construction.

34.2 Integrating Functions of Two Variables

The `Integrate()` method on **TwoVariableIntegrator** integrates a given two-variable function over a given region. For example, to compute the double integral:

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

First write the function:

Code Example – C# integration

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

Code Example – VB integration

```
Function F(V As DoubleVector) As Double
  Return 1.0 / (1.0 - (V(0) * V(0) * V(1) * V(1)))
End Function
```

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

Code Example – C# integration

```
var function = new MultiVariableFunction(
  new Func<DoubleVector, double>( F ) );
```

Code Example – VB integration

```
Dim MultiFunction As New MultiVariableFunction(
  New Func(Of DoubleVector, Double)(F))
```

Finally, compute the integral:

Code Example – C# integration

```
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 );
```

Code Example – VB integration

```
Dim Integrator As New TwoVariableIntegrator()
Dim XLower As Double = 0
Dim XUpper As Double = 1
Dim YLower As Double = 0
Dim YUpper As Double = 1
Dim Integral As Double = integrator.Integrate(MultiFunction,
  XLower, XUpper, YLower, YUpper)
```

The code above explicitly sets the *x* and *y* bounds. You can also set the *y* lower bound, *y* upper bound, or both, as a function of *x*. For example, to compute this double integral:

$$\int_{-3}^{3} \int_{-\sqrt{9-x^2}}^{\sqrt{9-x^2}} (9x^2 - 3y)\,dx\,dy$$

First define the function:

Code Example – C# integration

```
private double F( DoubleVector v )
{
  return ( 9.0 * v[0] * v[0] ) - ( 3.0 * v[1] );
}
```

Code Example – VB integration

```
Function F(V As DoubleVector) As Double
  Return (9.0 * V(0) * V(0)) - (3.0 * V(1))
End Function
```

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

Code Example – C# integration

```
var function = new MultiVariableFunction(
  new Func<DoubleVector, double>( F ) );
```

Code Example – VB integration

```
Dim function As New MultiVariableFunction(
  New Func(Of DoubleVector, Double)(F))
```

Then define the *y* bounding functions and encapsulate them as **OneVariableFunction** objects:

Code Example – C# integration

```
private double YUpperF( double x )
{
  return Math.Sqrt( 9.0 - ( x * x ) );
}

private double YLowerF( double x )
{
  return -YUpperF( x );
}

var yLowerFunction = new OneVariableFunction(
  new NMathFunctions.DoubleUnaryFunction( YLowerF ) );
var yUpperFunction = new OneVariableFunction(
  new NMathFunctions.DoubleUnaryFunction( YUpperF ) );
```

Code Example – VB integration

```
Function YUpperF(X As Double) As Double
  Return Math.Sqrt(9.0 - (X * X))
End Function

Function YLowerF(X As Double) As Double
  Return -YUpperF(X)
End Function

Dim YLowerFunction As New OneVariableFunction(
  New Func(Of Double, Double)(AddressOf YLowerF))
Dim YUpperFunction As New OneVariableFunction(
  New Func(Of Double, Double)(AddressOf YUpperF))
```

Finally, compute the integral:

Code Example – C# integration

```
var integrator = new TwoVariableIntegrator();
double xLower = -3;
double xUpper = 3;
double integral = integrator.Integrate( function, xLower, xUpper,
  yLowerFunction, yUpperFunction );
```

Code Example – VB integration

```
Dim Integrator As New TwoVariableIntegrator()
Dim XLower As Double = -3
Dim Xupper As Double = 3
Dim Integral As Double = integrator.Integrate(MultiFunction,
  XLower, XUpper, YLowerFunction, YUpperFunction)
```

CHAPTER 35.
DIFFERENTIAL EQUATIONS

NMath provides classes for solving first order initial value differential equations by the Runge-Kutta method.

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*.

NMath includes:

- Class **FirstOrderInitialValueProblem** encapsulates a first order initial value differential equation.
- Class **RungeKuttaSolver** solves an initial value ODE by the common Runge-Kutta method.
- Class **RungeKutta45OdeSolver** solves an initial value ODE using an explicit Runge-Kutta `(4,5)` formula known as the Dormand-Prince pair.
- Class **RungeKutta5OdeSolver** solves an initial value ODE using a non-adaptive explicit Runge-Kutta formula of order `5`.
- Class **VariableOrderOdeSolver** solves stiff and non-stiff ordinary differential equations. The algorithm uses higher order methods and smaller step size when the solution varies rapidly.

The chapter describes how to use class these classes.

35.1 Encapsulating Differential Equations

Class **FirstOrderInitialValueProblem** represents a first order initial value differential equation. If $y = y(x)$ is the unknown function, the first order initial value problem may be stated as

$$y' = F(x, y), y(x_0) = y_0$$

where y' denotes the first derivative of *y* with respect to *x*, *F* is a continuous function with bounded partial derivatives, and y_0 is the value of the unknown function *y* at the point x_0.

A **FirstOrderInitialValueProblem** instance is constructed from a function, *F*, and initial value, x_0 and y_0. The function *F* is encapsulated as a `Func<double, double, double>`, a delegate which takes two doubles and returns a double.

For example, the following code constructs a **FirstOrderInitialValueProblem** where $y' = x^2, y(0) = 1$:

Code Example – C# ordinary differential equations (ODE)

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

double x0 = 0.0;
double y0 = 1.0;

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

35.2 Solving Differential Equations

Class **RungeKuttaSolver** solves first order initial value differential equations by the Runge-Kutta method. The solver computes the unknown function *y* as set of tabulated values $\{x_i, y_i\}$ such that $y(x_i) = y_i$.

Constructing RungeKuttaSolver Instances

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 $i=0,1,\ldots,n$.

For instance:

Code Example – C# ordinary differential equations (ODE)

```
int n = 2000;
double delta = .001;
var solver = new RungeKuttaSolver( n, delta );
```

Code Example – VB ordinary differential equations (ODE)

```
Dim N As Integer = 2000
Dim Delta As Double = 0.001
Dim Solver As New RungeKuttaSolver(N, Delta)
```

Optionally, the order of the Runge-Kutta method may also be specified, using the enum `RungeKutterSolver.SolverOrder`.

Code Example – C# ordinary differential equations (ODE)

```
var solver = new RungeKuttaSolver( n, delta, SolverOrder.First );
```

Code Example – VB ordinary differential equations (ODE)

```
Dim Solver As New RungeKuttaSolver(N, Delta, SolverOrder.First)
```

By default, the fourth order Runge-Kutta method (RK4) is used.

Solving First Order Initial Value Problems

The `Solve()` method on **RungeKuttaSolver** solves a given **FirstOrderInitialValueProblem** by the common Runge-Kutta method. The `Solve()` method computes the unknown function *y* as set of tabulated values $\{x_i, y_i\}$ such that $y(x_i) = y_i$. The tabulated values are returned either as a `KeyValuePair<double[], double[]>`, or as a `CenterSpace.NMath.Core.TabulatedFunction` passed by reference. (See Section 13.5 for more information on tabulated functions.)

For example:

Code Example – C# ordinary differential equations (ODE)

```
KeyValuePair<double[], double[]> tabulatedValues =
  solver.Solve( prob );
```

Code Example – VB ordinary differential equations (ODE)

```
Dim TabulatedValues As KeyValuePair(Of Double(), Double()) =
  Solver.Solve(Prob)
```

or

Code Example – C# ordinary differential equations (ODE)

```
TabulatedFunction ftab = null;
solver.Solve( prob, ref ftab );
```

Code Example – VB ordinary differential equations (ODE)

```
Dim FTab As TabulatedFunction = Nothing
Solver.Solve(Prob, FTab)
```

Similarly, in the following code, results are returned as a **LinearSpline**, a subclass of **TabulatedFunction** that provides linear interpolation between tabulated values:

Code Example – C# ordinary differential equations (ODE)

```
var spline = new LinearSpline();
```

```
solver.Solve( prob, ref spline );
```

Code Example – VB ordinary differential equations (ODE)

```
Dim Spline As New LinearSpline()
Solver.Solve(Prob, Spline)
```

35.3 Dormand–Prince Method

The Dormand–Prince (DOPRI) method is a member of the Runge–Kutta family of ODE solvers. It uses six function evaluations to calculate fourth- and fifth-order accurate solutions. Dormand–Prince is currently the default method in MATLAB's `ode45` solver. **NMath** provides two solvers which use Dormand-Prince methods:

- Class **RungeKutta45OdeSolver** solves an initial value ODE using an explicit Runge-Kutta `(4,5)` formula known as the Dormand-Prince pair.
- Class **RungeKutta5OdeSolver** solves an initial value ODE using a non-adaptive explicit Runge-Kutta formula of order `5`. This is a non-adaptive solver. The step sequence is determined by given vector of time values, but the derivative function is evaluated multiple times per step. The solver implements the Dormand-Prince method of order 5 in a general framework of explicit Runge-Kutta methods.

For example, the following code shows how to use the **RungeKutta45OdeSolver** to solve a non-stiff system of equations describing the motion of a rigid body without external forces:

```
y1' = y2*y3,        y1(0)  = 0
y2' = -y1*y3,       y2(0)  = 1
y3' = -0.51*y1*y2, y3(0)  = 1
```

This function describes the system of differential equations:

Code Example – C# ordinary differential equations (ODE)

```
static DoubleVector Rigid( double t, DoubleVector y )
{
  var dy = new DoubleVector( 3 );
  dy[0] = y[1] * y[2];
  dy[1] = -y[0] * y[2];
  dy[2] = -0.51 * y[0] * y[1];
  return dy;
}
```

Code Example – VB ordinary differential equations (ODE)

```
Shared Function Rigid(T As Double, Y As DoubleVector)
  Dim DY As New DoubleVector(3)
```

```
  DY(0) = Y(1) * Y(2)
  DY(1) = -Y(0) * Y(2)
  DY(2) = -0.51 * Y(0) * Y(1)
  Return DY
End Function
```

Parameter *t* is the time parameter, and *y* is the state vector.

First, construct the solver:

Code Example – C# ordinary differential equations (ODE)

```
var solver = new RungeKutta45OdeSolver();
```

Code Example – VB ordinary differential equations (ODE)

```
Dim Solver As New RungeKutta45OdeSolver()
```

Next, construct the time span vector. If this vector contains exactly two points, the solver interprets these to be the initial and final time values. Step size and function output points are provided automatically by the solver. Here the initial time value *t0* is `0.0` and the final time value is `12.0`.

Code Example – C# ordinary differential equations (ODE)

```
var timeSpan = new DoubleVector( 0.0, 12.0 );
```

Code Example – VB ordinary differential equations (ODE)

```
Dim TimeSpan As New DoubleVector(0.0, 12.0)
```

Specify the initial *y* vector.

Code Example – C# ordinary differential equations (ODE)

```
var y0 = new DoubleVector( 0.0, 1.0, 1.0 );
```

Code Example – VB ordinary differential equations (ODE)

```
Dim Y0 As New DoubleVector(0.0, 1.0, 1.0)
```

Optionally, construct solver options. Here we set the absolute and relative tolerances to use.

Code Example – C# ordinary differential equations (ODE)

```
var solverOptions =
  new RungeKutta45OdeSolver.Options
  {
    AbsoluteTolerance = new DoubleVector( 1e-4, 1e-4, 1e-5 ),
    RelativeTolerance = 1e-4,
    Refine = 1
  };
```

Code Example – VB ordinary differential equations (ODE)

```
Dim SolverOptions As New RungeKutta45OdeSolver.Options()
SolverOptions.AbsoluteTolerance =
  New DoubleVector(0.0001, 0.0001, 0.00001)
SolverOptions.RelativeTolerance = 0.0001
SolverOptions.Refine = 1
```

Construct the delegate representing our system of differential equations.

Code Example – C# ordinary differential equations (ODE)

```
var odeFunction =
  new Func<double, DoubleVector, DoubleVector>( Rigid );
```

Code Example – VB ordinary differential equations (ODE)

```
Dim ODEFunction As New Func(Of Double, DoubleVector,
  DoubleVector)(AddressOf Rigid)
```

Finally, solve the problem. The solution is returned as a key/value pair. The first element of the pair is the time span vector, the second element is the corresponding solution values. That is, if the computed solution function is y then `y(soln.Key[i]) = soln.Value[i]`.

Code Example – C# ordinary differential equations (ODE)

```
RungeKutta45OdeSolver.Solution<DoubleMatrix> soln =
  solver.Solve( odeFunction, timeSpan, y0, solverOptions );
Console.WriteLine( "T = " + soln.T );
Console.WriteLine( "Y = " );
Console.WriteLine( soln.Y.ToTabDelimited() );
```

Code Example – VB ordinary differential equations (ODE)

```
Dim Soln As RungeKutta45OdeSolver.Solution(Of DoubleMatrix) =
Solver.Solve(ODEFunction, TimeSpan, Y0, SolverOptions)
Console.WriteLine("T = {0}", Soln.T)
Console.WriteLine("Y = ")
Console.WriteLine(Soln.Y.ToTabDelimited())
```

35.4 Stiff Equations

A *stiff* equation is a differential equation for which common numerical methods for solving the equation are numerically unstable, unless the step size is taken to be extremely small. In **NMath**, class **VariableOrderOdeSolver** solves stiff and non-stiff ordinary differential equations. The algorithm uses higher order methods and smaller step size when the solution varies rapidly.

The `Solve()` method solves the given initial value problem of ordinary differential equation of the form

$$y' = f(t, y)$$

or

$$y' = M(t, y)f(t, y)$$

for problems that involve a mass matrix M.

The function takes

- A delegate which evaluates the right hand side of the differential equations.
- A timespan vector specifying the interval of integration `[t0, tf]`. The solver imposes initial conditions at `t0` and integrates from `t0` to `tf`. If the timespan vector contains two elements, the solver returns the solution evaluated at every integration step. If the timespan vector contains more than two points, the solver returns the solution evaluated at the given points. The time values must be in order, either increasing or decreasing.
- The initial value for problem—the value of the unknown function y at the initial time value.

For example, the van der Pol equations in relaxation oscillation provide an example of a stiff system.[4] The limit cycle has portions where the solution

[4] https://www.mathworks.com/help/matlab/ref/ode15s.html

components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$y'_0 = y_1$$
$$y_0(0) = 2$$
$$y'_1 = 1000(1 - y_0^2)y_1 - y_0$$
$$y_1(0) = 0$$

To simulate this system, first create a function containing the equations. The following code creates the parameterized Van der Pol equation with parameter $\mu = 1000$. Parameter *t* is the time value, and *y* is the state value.

Code Example – C# stiff ODE solver

```
public static DoubleVector vdp1000( double t, DoubleVector y)
{
  DoubleVector dy = new DoubleVector( 2 );
  double y0 = y[0];
  double y1 = y[1];
  dy[0] = y1;
  dy[1] = 1000 * ( 1 - (y0 * y0) ) * y1 - y0;
  return dy;
}
```

Next, create a function that returns the Jacobian at given *t* and *y* values.

Code Example – C# stiff ODE solver

```
public static DoubleMatrix vdp1000Jac(double t, DoubleVector y)
{
  DoubleMatrix J = new DoubleMatrix( 2, 2 );
  J[0, 0] = 0;
  J[0, 1] = 1;
  J[1, 0] = -2 * 1000 * y[0] * y[1] - 1;
  J[1, 1] = 1000 * ( 1.0 - y[0] * y[0] );
  return J;
}
```

Create the initial values and time interval, and encapsulate the ODE function.

Code Example – C# stiff ODE solver

```
var y0 = new DoubleVector( 2.0, 0.0 );
var timespan = new DoubleVector( 0.0, 3000.0 );
var odeFunc =
  new Func<double, DoubleVector, DoubleVector>( vdp1000 );
```

Create the solver object and set up the solver options. Here we use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively). Also, since we have an explicit form for the Jacobian function, we set this in the solver options too.

Code Example – C# stiff ODE solver

```
var ode15s = new VariableOrderOdeSolver();
var options = new VariableOrderOdeSolver.Options();
options.JacobianFunction =
  new Func<double, DoubleVector, DoubleMatrix>( vdp1000Jac );
```

Finally, solve the equation and display the solution.

Code Example – C# stiff ODE solver

```
VariabeOrderOdeSolver.Solution<DoubleMatrix> soln =
  ode15s.Solve( vdp1000, timespan, y0, options );

Console.WriteLine( "t = " + NMathFunctions.Round(soln.T, 4) );
Console.WriteLine();
Console.WriteLine( "y = " );
Console.WriteLine(
  NMathFunctions.Round( soln.Y, 4 ).ToTabDelimited() );
```

PART V - STATISTICS

CHAPTER 36. STATISTICS INTRODUCTION

NMath's statistical suite is fully integrated into CenterSpace Software's **NMath**™ product. **NMath** provides object-oriented components for mathematical, engineering, scientific, and financial applications and includes statistical components for descriptive statistics, probability distributions, combinatorial functions, multiple linear regression, hypothesis testing, and analysis of variance all optimized for the .NET platform.

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

36.1 Product Features

The statistical features of **NMath** include:

- 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.
- 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.

36.2 Namespaces

All types in **NMath** are in the `CenterSpace.NMath.Core` namespace. To avoid using fully qualified names, preface your code with the namespace statement. For example:

Code Example – C#

```
using CenterSpace.NMath.Core;
```

Code Example – VB

```
Imports CenterSpace.NMath.Core
```

All **NMath** code shown in this manual assumes the presence of such a namespace statement.

CHAPTER 37.
DATA FRAMES

Statistical functions generally support the **NMath** types **DoubleVector** and **DoubleMatrix**, as well as native 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.

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

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. Column names and row keys do not need to be unique. For example, this output shows a formatted string representation of data from a sample data frame:

```
#             State  Weight  Married
John Smith    OR     165     true
Ruth Barnes   WA     147     true
Jane Jones    VT     115     false
Tim Travis    AK     230     false
Betsy Young   MA     130     true
Arthur Smith  CA     152     false
Emma Allen    OK     135     false
Roy Wilkenson WI     182     true
```

This data frame contains three columns: column `0`, named `State`, contains string data; column `1`, named `Weight`, contains integer data; column `2`, named `Married`, contains boolean data. There are eight rows of data in this data frame, and the subjects' names are used as row keys.

This chapter describes how to use the **DataFrame** class.

37.1 Column Types

A **DataFrame** may contain columns of different types—the only constraint is that the columns must be of the same length. **DFColumn**, which implements the **IDFColumn** interface, is the abstract base class for data frame columns. **NMath** provides the following derived classes for column types:

- **DFBoolColumn** represents a column of logical data.
- **DFDateTimeColumn** represents a column of temporal data.
- **DFGenericColumn** represents a column of generic data.
- **DFIntColumn** represents a column of integer data.
- **DFNumericColumn** represents a column of double-precision floating point data.
- **DFStringColumn** represents a column of string data.

Creating Columns

Empty columns are constructed by simply supplying a name for the column. For example:

Code Example – C#

```
var col = new DFDateTimeColumn( "myCol" );
```

Code Example – VB

```
Dim Col As New DFDateTimeColumn("myCol")
```

The name of a column can be used to access the column in a data frame. Once a column instance is constructed, the name cannot be changed.

NOTE—Columns also provide a modifiable Label property for display purposes; see below.

Columns can also be initialized with an array of data at construction time:

Code Example – C#

```
var bArray =
  new bool[] { true, false, true, true, true, false, false };
var col = new DFBoolColumn( "myCol", bArray );
```

Code Example – VB

```
Dim BArray() As Boolean = {True, False, True, True, True, False,
False}
Dim Col As New DFBoolColumn("myCol", BArray)
```

Constructors that take an array of data use the `params` keyword, so values may also be passed as parameters:

Code Example – C#

```
var col =
  new DFStringColumn( "myCol", "Jane", "Joe", "Mary", "Bill" );
```

Code Example – VB

```
Dim Col As
  New DFStringColumn("myCol", "Jane", "Joe", "Mary", "Bill")
```

Some column types provide additional options for initializing data at construction time. For instance, this code initializes a numeric column with data from a **DoubleVector**:

Code Example – C#

```
var v = new DoubleVector( 50, 0, .1 );
var col = new DFNumericColumn( "myCol", v );
```

Code Example – VB

```
Dim V As New DoubleVector(50, 0, 0.1)
Dim Col As New DFNumericColumn("myCol", V)
```

This code initializes a generic column with data from an **ICollection**:

Code Example – C#

```
var list = new ArrayList( 3 );
list.Add( 3.14 );
list.Add( "Hello World" );
list.Add( DateTime.Now );
var col = new DFGenericColumn( "myCol", list );
```

Code Example – VB

```
Dim List As New ArrayList(3)
List.Add(3.14)
```

```
List.Add("Hello World")
List.Add(DateTime.Now)
Dim Col As New DFGenericColumn("myCol", List)
```

Lastly, you can create a column from another column. For example, this code creates a **DFIntColumn** from a **DFStringColumn**:

Code Example – C#

```
var col = new DFStringColumn( “Col1”, “1”, “2”, “3”, “4” );
var col2 = new DFIntColumn( “Col2”, col1 );
```

Code Example – VB

```
Dim Col As New DFStringColumn("Col1", "1", "2", "3", "4")
Dim Col2 As New DFIntColumn("Col2", Col1)
```

A **NMathFormatException** is raised if the data in the given column cannot be converted to the appropriate type.

Adding and Removing Data

Once a column is constructed you can add or remove data from it. The `Add()` method appends an element to the end of the column:

Code Example – C#

```
var col = new DFStringColumn( "Name" );
col.Add( "Joe Smith" );
col.Add( "Jane Doe" );
col.Add( "John Davis" );
```

Code Example – VB

```
Dim Col As New DFStringColumn("Name")
Col.Add("Joe Smith")
Col.Add("Jane Doe")
Col.Add("John Davis")
```

The `Insert()` method inserts an element into a column at a given index. For instance, this code insert a new element at the top of the column:

Code Example – C#

```
col.Insert( 0, "Sally Jones" );
```

Code Example – VB

```
Col.Insert(0, "Sally Jones")
```

The `RemoveAt()` method removes the element at a given index:

Code Example – C#

```
col.RemoveAt( 3 );
```

Code Example – VB

```
Col.RemoveAt(3)
```

Accessing Column Data

The data frame column classes provide standard indexing operators for getting and setting element values. Thus, `col[i]` always returns the *i*th element of the column:

Code Example – C#

```
var col =
  new DFStringColumn( "Names", "Jane", "Joe", "Mary", "Bill" );
col[0] = "Janet";
```

Code Example – VB

```
Dim Col As
  New DFStringColumn("Names", "Jane", "Joe", "Mary", "Bill")
Col(0) = "Janet"
```

The `GetEnumerator()` method returns an enumerator for the column data:

Code Example – C#

```
IEnumerator enumerator = col.GetEnumerator();
while ( enumerator.MoveNext() )
{
  // Do something with enumerator.Current
}
```

Code Example – VB

```
Dim Enumerator As IEnumerator = Col.GetEnumerator()
While (Enumerator.MoveNext())
  '' Do something with enumerator.Current
End While
```

Column Properties

Data frame column types provide the following properties:

- `ColumnType` gets the type of the objects held by the column.
- `Count` gets the number of ojects in the column.

- `IsNumeric` returns `true` if a column is of type **DFIntColumn** or **DFNumericColumn**.
- `Label` gets and sets the label in the header of the column.
- `MissingValue` gets and sets the value used to represent missing values in the column (see below).
- `Name` gets the name of the column.

NOTE—The Name of a column can only be set in a constructor. Once a column is constructed, the name cannot be changed. For a modifiable label, see the Label property.

Reordering Column Data

You can use the `Permute()` method to arbitrarily reorder the elements in a column. This method accepts a permutation array of element indices and reorders the elements such that `this[ permutation[i] ]` is set to the *i*th object in the original column.

For example, this code moves the last two elements to the head of the column:

Code Example – C#

```
var col =
   new DFStringColumn( "myCol", "a", "b", "c", "d", "e" );
col.Permute( 2, 3, 4, 0, 1 );
```

Code Example – VB

```
Dim Col As New DFStringColumn("myCol", "a", "b", "c", "d", "e")
Col.Permute(2, 3, 4, 0, 1)
```

Missing Values

All column types—except **DFBoolColumn**, which has only two valid values—support missing values. Most statistical functions in **NMath** are accompanied by a paired function that ignores missing values (Section 38.2).

NOTE—To represent missing values in boolean data, use a DFIntColumn. For example, use 1 for true, 0 for false, and -1 for missing.

At construction time, the missing value for a column is defined using a static variable in class **StatsSettings**, as shown in Table 24.

Table 24 – Default missing values for data frame column types

Column Type	StatsSettings Variable	Default Value
DFDateTimeColumn	`DateTimeMissingValue`	`DateTime.MinValue`
DFGenericColumn	`GenericMissingValue`	`null`
DFIntColumn	`IntegerMissingValue`	`int.MinValue`
DFNumericColumn	`NumericMissingValue`	`Double.NaN`
DFStringColumn	`StringMissingValue`	`“.”`

For instance, this code computes the mean of a column of integers, ignoring any missing values:

Code Example – C#

```
var col = new DFIntColumn( “myCol”, 5, 2, -1, 1, 0, 7 );
double mean = StatsFunctions.NaNMean( col );
```

Code Example – VB

```
Dim Col As New DFIntColumn("myCol", 5, 2, -1, 1, 0, 7)
Dim Mean As Double = StatsFunctions.NaNMean(Col)
```

By default, a missing value in a **DFIntColumn** is represented using the default setting of `StatsFunctions.IntegerMissingValue`, which is `int.MinValue`. You can change the way a missing value is represented for a particular column instance using the `MissingValue` property:

Code Example – C#

```
col.MissingValue = -1;
double mean = StatsFunctions.NaNMean( col );
```

Code Example – VB

```
Col.MissingValue = -1
Dim Mean As Double = StatsFunctions.NaNMean(Col)
```

In this example, all values in `col` equal to `-1` are ignored when computing the mean.

NOTE—For DFNumericColumn instances you can use the MissingValue property to indicate that missing values are represented by something other than the default value Double.NaN. However, Double.NaN will continue to be treated as missing, in addition to whatever value you set.

You can also change the default missing value for all columns of a particular type by setting the appropriate static variable in **StatsSettings**. Thus, this code sets the default missing value for integer columns to `-1` for all subsequently constructed **DFIntColumn** instances:

Code Example – C#

```
StatsSettings.IntegerMissingValue = -1;
```

Code Example – VB

```
StatsSettings.IntegerMissingValue = -1
```

The `Clean()` method returns a new column with missing values removed.

Transforming Column Data

NMath provides convenience methods for applying functions to elements of a column. Each of these methods takes a function delegate. The `Apply()` method returns a new column whose contents are the result of applying the given function to each element of the column. The `Transform()` method modifies a column object by applying the given function to each of its elements.

Suppose, for example, that you want to cap all numeric values in a **DFNumericColumn** at `100.0`. You could write a simple function like this:

Code Example – C#

```
private static double Cap( double x )
{
  return x > 100.0 ? 100.0 : x;
}
```

Code Example – VB

```
Private Shared Function Cap(X As Double) As Double
  If X > 100 Then
    Return 100
  Else
    Return X
  End If
End Function
```

Then encapsulate the function in a `Func<double, double>` delegate:

Code Example – C#

```
var capDelegate = new Func<double, double>( Cap );
```

Code Example – VB

```
Dim CapDelegate As New Func(Of Double, Double)(AddressOf Cap)
```

This code caps all numeric values in column `col`:

Code Example – C#

```
col.Transform( capDelegate );
```

Code Example – VB

```
Col.Transform( capDelegate )
```

A common use of the `Apply()` functions is to create a new column whose values are a function of values in one or two existing column. For example, suppose you have `FirstName` and `LastName` string columns in data frame `df`, and want to create a new column containing customers' full names. You could write a simple function like this:

Code Example – C#

```
private static string Cat( string first, string last )
{
  return first + " " + last;
}
```

Code Example – VB

```
Private Shared Function Cat(First As String, Last As String) As
String
  Return First & Last
End Function
```

Then encapsulate the function in a `Func<String, String, String>` delegate:

Code Example – C#

```
var catDelegate = new Func<String, String, String>( Cat );
```

Code Example – VB

```
Dim CatDelegate As New Func(Of String, String, String)(AddressOf
Cat)
```

This code creates a new column containing the concatenated names:

Code Example – C#

```
DFStringColumn col =
  ( (DFStringColumn)data["FirstName"] ).Apply( “FullName”,
    catDelegate, (DFStringColumn)data["LastName"] );
```

Code Example – VB

```
Dim First As DFStringColumn =
  CType(Data["FirstName"], DFStringColumn )
Dim Last As DFStringColumn =
  CType(Data["LastName"], DFStringcolumn )
```

```
Dim Col As DFStringColumn =
  First.Apply("FullName", CatDelegate, Last)
```

Exporting Column Data

Data from a column can be exported in various ways:

- `ToArray()` exports the contents of a column to a strongly-typed array.
- `ToDoubleArray()` extracts the contents of a column to an array of doubles (numeric columns only).
- `ToDoubleVector()` extracts the contents of a column to a **DoubleVector** (numeric columns only).
- `ToIntArray()` extracts the contents of a column to an array of integers (integer columns only).
- `ToString()` returns a formatted string representation of a column.
- `ToStringArray()` exports the contents of a column to an array of strings.

37.2 Creating DataFrames

Data frames can be constructed in a variety of ways.

Creating Empty DataFrames

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.

Code Example – C#

```
var df = new DataFrame();

// Add some columns
df.AddColumn( new DFStringColumn( "Sex" ));
df.AddColumn( new DFStringColumn( "AgeGroup" ));
df.AddColumn( new DFIntColumn( "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 );
df.AddRow( "Emma Allen", "F", "Child", 35 );
```

Code Example – VB

```
Dim DF As New DataFrame()

'' Add some columns
DF.AddColumn( New DFStringColumn( "Sex" ))
DF.AddColumn( New DFStringColumn( "AgeGroup" ))
DF.AddColumn( New DFIntColumn( "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 )
DF.AddRow( "Emma Allen", "F", "Child", 35 )
```

NOTE—The first parameter to the AddRow() method is the row key. See Section 37.3 and Section 37.4, respectively, for more information on adding columns and rows to a data frame.

Creating DataFrames from Arrays of Columns

You can also construct and populate columns independently, then combine them into a data frame:

Code Example – C#

```
var col1 = new DFNumericColumn( "Col1", 1.1, 2.2, 3.3, 4.4 );
var col2 = new DFBoolColumn ( "Col2", true, true, false, true );
var col3 =
  new DFStringColumn ( "Col3", "John", "Paulo", "Sam", "Becky" );
var cols = new DFColumn[] { col1, col2, col3 };
var df = new DataFrame( cols );
```

Code Example – VB

```
Dim Col1 As New DFNumericColumn("Col1", 1.1, 2.2, 3.3, 4.4)
Dim Col2 As New DFBoolColumn("Col2", True, True, False, True)
Dim Col3 As
  New DFStringColumn("Col3", "John", "Paulo", "Sam", "Becky")
Dim Cols As DFColumn() = {Col1, Col2, Col3}
Dim DF As New DataFrame(Cols)
```

An **InvalidArgumentException** is thrown if the columns are not all of the same length.

In this case, the row keys are set to nulls; they can later be initialized using the `SetRowKeys()` method. Alternatively, you can pass in a collection of row keys at construction time:

Code Example – C#

```
var keys = new object[] { "Row1", "Row2", "Row3", "Row4" };
var df = new DataFrame( cols, keys );
```

Code Example – VB

```
Dim Keys As Object() = {"Row1", "Row2", "Row3", "Row4"}
Dim DF As New DataFrame(Cols, Keys)
```

Creating DataFrames from Matrices

You can construct a data frame from a **DoubleMatrix** and an array of column names. A new **DFNumericColumn** is added for each column in the matrix. For instance, this code creates a data frame from an 8 x 3 matrix:

Code Example – C#

```
var A = new DoubleMatrix( 8, 3, 0, 1 );
var colNames = new string[] { "A", "B", "C" };
var df = new DataFrame( A, colNames );
```

Code Example – VB

```
Dim A As New DoubleMatrix(8, 3, 0, 1)
Dim ColNames As String() = {"A", "B", "C"}
Dim DF As New DataFrame(A, ColNames)
```

The number of column names must match the number of columns in the matrix.

Creating DataFrames from ADO.NET Objects

You can construct a data frame from an ADO.NET **DataTable**. For example, assuming `table` is a **DataTable** instance:

Code Example – C#

```
var df = new DataFrame( table );
```

Code Example – VB

```
Dim DF As New DataFrame(Table)
```

In this case, the row keys are set to the default `rowIndex + 1`—that is, `1...n`. You can also specify the row keys in various ways. This code passes in an array of row keys:

Code Example – C#

```
var keys = new object[] { "Row1", "Row2", "Row3", "Row4" };
var df = new DataFrame( table, keys );
```

Code Example – VB

```
Dim Keys As Object() { "Row1", "Row2", "Row3", "Row4" }
Dim DF As New DataFrame(Table, Keys)
```

Alternatively, you can indicate a column in the **DataTable**, either by column index or column name, to use for the row keys. This code uses column `ID` for row keys:

Code Example – C#

```
var df = new DataFrame( table, "ID" );
```

Code Example – VB

```
Dim DF As New DataFrame(Table, "ID")
```

Creating DataFrames from Strings

You can construct a data frame from a string representation. For example, if `str` is a tab-delimited string containing:

```
Key  Col1 Col2   Col3
Row1 1.1  true   A
Row2 2.2  true   B
Row3 3.3  false  A
Row4 4.4  true   C
```

Then you could construct a data frame like so:

Code Example – C#

```
var df = new DataFrame( str );
```

Code Example – VB

```
Dim DF As New DataFrame(Str)
```

For more control, you can also indicate:

- whether the first row of data contains column headers
- whether the first column of data contains row keys
- the delimiter used to separate columns

- whether to parse the column types, or to treat everything as string data

For example, if `str` is a comma-delimited string containing column headers but no row keys:

```
Col1,Col2,Col3
1.1,true,A
2.2,true,B
3.3,false,A
4.4,true,C
```

you could construct a data frame like so:

Code Example – C#

```
var df = new DataFrame( str, true, false, ",", true );
```

Code Example – VB

```
Dim DF As New DataFrame(Str, True, False, ",", True)
```

37.3 Adding and Removing Columns

The `AddColumn()` method adds a column to a data frame:

Code Example – C#

```
var df = new DataFrame();
var col = new DFNumericColumn( "myCol" );
df.AddColumn( col );
```

Code Example – VB

```
Dim DF As New DataFrame()
Dim Col As New DFNumericColumn("myCol")
DF.AddColumn( Col )
```

NOTE—The AddColumn() method raises a MismatchedSizeException if you attempt to add a column that is not the same length as any existing columns in a data frame.

You can also add all the columns from one data frame to another, optionally copying the data in the columns. For example, assuming `df` is a data frame, this code adds the columns of `df` to a new data frame and copies all the column data:

Code Example – C#

```
var df2 = new DataFrame();
df2.AddColumns( df, true );
```

Code Example – VB

```
Dim DF2 As New DataFrame()
DF2.AddColumns(DF, True)
```

Overloads of `AddColumn()` and `AddColumns()` accept ADO.NET **DataColumn** and **DataColumnCollection** instances, respectively. If the data frame already contains rows of data, you must also pass in a **DataRowCollection** of the same `Count` as the number of rows in the data frame.

`InsertColumn()` inserts a column at a given column index. This code adds a column in the first position:

Code Example – C#

```
var col = new DFStringColumn( "myCol" );
df.InsertColumn( 0, col );
```

Code Example – VB

```
Dim Col As New DFStringColumn("myCol")
DF.InsertColumn(0, Col)
```

`RemoveColumn()` removes the column at a given index:

Code Example – C#

```
df.RemoveColumn( 3 );
```

Code Example – VB

```
DF.RemoveColumn(3)
```

You can also identify a column by name:

Code Example – C#

```
df.RemoveColumn( "myCol" );
```

Code Example – VB

```
DF.RemoveColumn( "myCol" )
```

Because column names are not constrained to be unique, this method will remove *all* columns in the data frame with the given name.

`RemoveAllColumns()` removes all columns from a data frame, but preserves the existing row keys. `RemoveColumns()` removes the columns specified in a given subset or slice.

`Clear()` method removes all columns and rows from a data frame. `CleanCols()` returns a new data frame containing only those columns in a data frame that do not contain missing values.

37.4 Adding and Removing Rows

The `AddRow()` method adds a row of data to a data frame. The first parameter is the row key; subsequent parameters are the row data. For example:

Code Example – C#

```
var df = new DataFrame();
df.AddColumn( new DFStringColumn( "Col1" ));
df.AddColumn( new DFNumericColumn( "Col2" ) );
df.AddColumn( new DFNumericColumn( "Col3" ) );
df.AddRow( 1546, "Test1", 1.5445, 667.87 );
```

Code Example – VB

```
Dim DF As New DataFrame()
DF.AddColumn( New DFStringColumn( "Col1" ))
DF.AddColumn( New DFNumericColumn( "Col2" ) )
DF.AddColumn( New DFNumericColumn( "Col3" ) )
DF.AddRow( 1546, "Test1", 1.5445, 667.87 )
```

NOTE—The AddRow() method raises a MismatchedSizeException if the number of row elements does not match the number of columns in the data frame.

This example uses `1546` as an integer row key, perhaps representing some sort of ID. Row keys can be any object, and need not be unique.

Additional overloads of `AddRow()` accept data in various collections other than an array of objects. One overload takes an **ICollection**. For instance:

Code Example – C#

```
var myQ = new Queue();
myQ.Enqueue( "Hello" );
myQ.Enqueue( 47.0 );
myQ.Enqueue( -0.34 );
df.AddRow( "Row1", myQ );
```

Code Example – VB

```
Dim MyQ As New Queue()
MyQ.Enqueue( "Hello" )
MyQ.Enqueue( 47.0 )
MyQ.Enqueue( -0.34 )
DF.AddRow( "Row1", MyQ )
```

Another overload accepts an **IDictionary** in which the keys are the column names and the values are the row data:

Code Example – C#

```
var df = new DataFrame();
df.AddColumn( new DFNumericColumn( "V1" ) );
```

```
df.AddColumn( new DFBoolColumn( "V2" ) );
df.AddColumn( new DFStringColumn( "V3" ) );
var myHT = new Hashtable();
myHT.Add( "V1", 3.14 );
myHT.Add( "V3", "Hello");
myHT.Add( "V2", true );
df.AddRow( "Row1", myHT );
```

Code Example – VB

```
Dim DF As New DataFrame()
DF.AddColumn( New DFNumericColumn( "V1" ) )
DF.AddColumn( New DFBoolColumn( "V2" ) )
df.AddColumn( New DFStringColumn( "V3" ) )
Dim MyHT As New Hashtable()
MyHT.Add("V1", 3.14)
MyHT.Add("V3", "Hello")
MyHT.Add("V2", True)
DF.AddRow("Row1", MyHT)
```

If all of the columns in your data frame are numeric, you can add a row as a **DoubleVector**:

Code Example – C#

```
var v = new DoubleVector( 10, 0, 1 );
df.AddRow( "myKey", v );
```

Code Example – VB

```
Dim V As New DoubleVector(10, 0, 1)
DF.AddRow("myKey", V)
```

Other overloads of `AddRow()` and `AddRows()` accept ADO.NET **DataRow** and **DataRowCollection** instances, respectively.

`InsertRow()` inserts a row at a given row index. For example, this code inserts a row into the second position:

Code Example – C#

```
var df = new DataFrame();
df.AddColumn( new DFNumericColumn( "Col1" ) );
df.AddColumn( new DFNumericColumn( "Col2" ) );
df.AddColumn( new DFNumericColumn( "Col3" ) );
df.AddRow( "Row1", 2.5, 0.0, 3.4 );
df.AddRow( "Row2", 3.14, -.5, -.33 );
df.AddRow( "Row3", 0.1, 55.34, 12.02 );
df.AddRow( "Row4", 3.14, -33.2, 7.22 );
var myRow = object[] { 5.5, 9.05, -6.11 };
df.InsertRow( 1, "Row1a", myRow );
```

Code Example – VB

```
Dim DF As New DataFrame()
DF.AddColumn(New DFNumericColumn("Col1"))
DF.AddColumn(New DFNumericColumn("Col2"))
DF.AddColumn(New DFNumericColumn("Col3"))
DF.AddRow("Row1", 2.5, 0.0, 3.4)
DF.AddRow("Row2", 3.14, -0.5, -0.33)
DF.AddRow("Row3", 0.1, 55.34, 12.02)
DF.AddRow("Row4", 3.14, -33.2, 7.22)
Dim MyRow As Object() = {5.5, 9.05, -6.11}
DF.InsertRow(1, "Row1a", MyRow)
```

Again, overloads are provided for adding row data in various collection types.

`RemoveRow()` removes the row at a given index:

Code Example – C#

```
df.RemoveRow( 0 );
```

Code Example – VB

```
DF.RemoveRow(0)
```

You can also identify a row by key:

Code Example – C#

```
df.RemoveRow( “Row3” );
```

Code Example – VB

```
DF.RemoveRow("Row3")
```

Because row keys are not constrained to be unique, this method will remove *all* rows in the data frame with the given key.

`RemoveAllRows()` removes all rows from a data frame, but preserves the existing columns. `RemoveRows()` removes the rows specified in a given subset or slice.

`Clear()` method removes all rows and columns from a data frame. `CleanRows()` returns a new data frame containing only those rows in a data frame that do not contain missing values.

Modifying Row Keys

Unlike column names which are fixed at construction time, row keys can be changed at any time. The `SetRowKey()` method sets the key for a given row to a given value. Remember that row keys can be any object:

Code Example – C#

```
df.SetRowKey( 0, 1.14 );
df.SetRowKey( 1, “John Doe” );
df.SetRowKey( 2, true );
```

Code Example – VB

```
DF.SetRowKey(0, 1.14)
DF.SetRowKey(1, "John Doe")
DF.SetRowKey(2, True)
```

`SetRowKeys()` accepts a collection of row keys, and raises a **MismatchedSizeException** if if the number of elements in the collection does not equal the number of rows in this data frame:

Code Example – C#

```
object[] keys = { “Subject1”, “Subject2”, “Subject3” };
df.SetRowKeys( keys );
```

Code Example – VB

```
Dim Keys As Object() = {"Subject1", "Subject2", "Subject3"}
DF.SetRowKeys(Keys)
```

Finally, `IndexRowKeys()` resets the row keys for all rows to `rowIndex + 1`; that is, `1...n`.

37.5 Properties of DataFrames

The **DataFrame** class provides the following properties:

- `Cols` gets the number of columns.
- `ColumnNames` gets an array of the column names.
- `ColumnHeaders` gets and sets the array of column labels used for display purposes.
- `CreateDate` gets the creation datetime for the date frame.
- `Name` gets and sets the name of the data frame.
- `Rows` gets the number of rows.
- `RowKeyHeader` gets and sets the header for the row keys for display purposes. The default row key header is `#`.
- `RowKeys` gets an object array of the row keys.

- `StringRowKeys` gets a string array of the row keys.

37.6 Accessing DataFrames

Class **DataFrame** provides a wide range of indexers and member functions accessing individual elements, columns, or rows in a data frame.

NOTE—For information on getting arbitrary sub-frames from a data frame, see Section 37.8.

Accessing Elements

Class **DataFrame** provides a two-dimensional indexing operator for getting and setting individual element values. Thus, `df[i,j]` always returns the *i*th element of the *j*th column:

Code Example – C#

```
df[3,0] = 1.0;
```

Code Example – VB

```
DF(3, 0) = 1.0
```

Accessing Columns

The one-dimensional indexing operator `df[i]` always returns the *i*th column:

Code Example – C#

```
DFNumericColumn col = df[3];
```

Code Example – VB

```
Dim Col As DFNumericColumn = DF(3)
```

You can also access columns by name:

Code Example – C#

```
DFNumericColumn col = df[ "myCol" ];
```

Code Example – VB

```
Dim Col As DFNumericColumn = DF("myCol")
```

Because column names are not constrained to be unique, this returns the first column with the given name, or `null` if a column by that name is not found.

The `IndexOfColumn()` method returns the index of the first column with a given name, or `null` if a column by that name is not found. `IndicesOfColumn()` returns an array of all column indices for a given column name.

You can also check whether a column of a given name exists in a data frame using the `ContainsColumn()` method:

Code Example – C#

```
if ( df.ContainsColumn( "myCol" ) )
{
  // Do something here with df[ "myCol" ]
}
```

Code Example – VB

```
If (DF.ContainsColumn("myCol")) Then
  '' Do something here with DF( "myCol" )
End If
```

Finally, the `GetColumnDictionary()` method returns an **IDictionary** of the values in a given column. For instance, this code gets a dictionary of the values in column `2`:

Code Example – C#

```
IDictionary dict = df.GetColumnDictionary( 2 );
```

Code Example – VB

```
Dim Dict As IDictionary = DF.GetColumnDictionary(2)
```

The row keys are used as keys in the dictionary. Alternatively, you can specify two column indices—the first is used for the dictionary keys, the second for the dictionary values:

Code Example – C#

```
IDictionary dict = df.GetColumnDictionary( 0, 2 );
```

Code Example – VB

```
Dim Dict As IDictionary = DF.GetColumnDictionary(0, 2)
```

In this example, the elements in column `0` are used as the dictionary keys.

Accessing Rows

Because the one-dimensional indexer `df[i]` is already used for accessing data frame columns, class **DataFrame** provides `GetRow()` methods for accessing individual rows. Thus, `GetRow( i )` returns the data in the *i*th row as an array of objects:

Code Example – C#

```
object[] rowData = df.GetRow( 3 );
```

Code Example – VB

```
Dim RowData As Object() = DF.GetRow(3)
```

You can also access rows by key:

Code Example – C#

```
object[] rowData = df.GetRow( "myKey" );
```

Code Example – VB

```
Dim RowData As Object() = DF.GetRow("myKey")
```

Because row keys are not constrained to be unique, this returns the first row with the given key, or `null` if a row with that key is not found.

The `IndexOfKey()` method returns the index of the first row with a given key, or `null` if a row with that key is not found. `IndicesOfKey()` returns an array of all row indices for a given key.

You can also retrieve the indices of rows with a particular value in a given column. `IndexOf()` returns the first row with a particular value in a column; `IndicesOf()` returns all rows. For instance, this code gets an array of row indices for all rows which have the value `"John Doe"` in column `2`:

Code Example – C#

```
int[] rowIndices = df.IndicesOf( 2, "John Doe" );
```

Code Example – VB

```
Dim RowIndices As Integer() = DF.IndicesOf(2, "John Doe")
```

Lastly, the `GetRowDictionary()` method returns an **IDictionary** of the data in a given row, specified either by row index or row key. The column names are used as keys in the dictionary. Thus, this code gets a dictionary of the data in row `3`:

Code Example – C#

```
IDictionary dict = df.GetRowDictionary( 3 );
```

Code Example – VB

```
Dim Dict As IDictionary = DF.GetRowDictionary(3)
```

37.7 Subsets

In addition to accessors for individual elements, columns, or rows in a data frame (Section 37.6), class **DataFrame** provides a large number of indexers and member functions for accessing sub-frames containing any arbitrary subset of rows, columns, or both (Section 37.8).

Such indexers and methods accept the **NMath** types **Slice** and **Range** to indicate sets of row or column indices with constant spacing, as well as abstract values like `Slice.All` for indexing all elements.

In addition, **NMath** introduces a new 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.

This section describes the **Subset** class.

Creating Subsets

Subset instances can be constructed in a variety of ways. One constructor simply accepts an array of integers:

Code Example – C#

```
var sub = new Subset( new int[] { 5, 4, 0, 12 } );
```

Code Example – VB

```
Dim Subset As New Subset(New Integer() {5, 4, 0, 12})
```

Another constructor accepts an **ICollection** whose elements are all `System.Int32`.

A very useful constructor takes an array of boolean values and constructs a **Subset** containing the indices of all `true` elements in the array. This can used, for example, to create a subset from a **DataFrame** containing the indices of the rows or columns than meet a certain criteria.

Thus, this code creates a subset of row indices containing those rows where the value in column `2` is greater than the value in column `3`:

Code Example – C#

```
var bArray = new bool[ df.Rows ];
for ( int i = 0; i < df.Rows; i++ )
{
  bArray[i] = ( df[2][i] > df[3][i] );
}
var rowIndices = new Subset( bArray );
```

Code Example – VB

```
Dim BArray(DF.Rows) As Boolean
For I As Integer = 0 To DF.Rows - 1
  BArray(I) = DF(2)(I) > DF(3)(I)
Next
Dim RowIndices As New Subset(BArray)
```

This **Subset** could be use to access the sub-frame containing only those rows that meet the criterion, as described in Section 37.8.

A **Subset** can also be constructed from an array of other subsets. The subsets are simply concatenated. To created a sorted **Subset** of the unique indices, you can call `Unique()` on the constructed **Subset** (see below).

Lastly, constructors are provided that construct subsets with continuous spacing, like slices and ranges. For instance, this code creates a subset starting at `2`, with `5` total elements, and a stepsize of `1`:

Code Example – C#

```
var sub = new Subset( 2, 5, 1 );
```

Code Example – VB

```
Dim Subset As New Subset(2, 5, 1)
```

Properties of Subsets

Class **Subset** provides the following read-only properties:

- `First` gets the first index in the subset.
- `Length` gets the total number of indices in the subset.
- `Indices` gets the underlying array of integers.
- `Last` gets the last index in the subset.

Accessing Elements

Class **Subset** provides an indexing operator for getting and setting element values. Thus, `subset[i]` returns the *i*th element of the underlying array of integers.

Code Example – C#

```
sub[ 3 ] = 4;
```

Code Example – VB

```
Subset(3) = 4
```

NOTE—Indexing starts at 0.

The `Get( i )` method safely gets the index at a given position by looping around the end of the subset if `i` exceeds the length of the subset:

Code Example – C#

```
var sub = new Subset( new int[] { 3, 4, 5, 8, 9 } );
int index = sub.Get( 5 )
// index = 3
```

Code Example – VB

```
Dim Subset As New Subset(New Integer() {3, 4, 5, 8, 9})
Dim Index As Integer = Subset.Get(5)
'' index = 3
```

You can also create a **Subset** of a **Subset** using the indexing operator. For instance:

Code Example – C#

```
var sub1 = new Subset( new int[] { 1, 3, 4, 7, 9 } );
var sub2 = new Subset( new int[] { 0, 2, 4 } );
Subset sub3 = sub1[ sub2 ];
// sub3.Indices = 1, 4, 9
```

Code Example – VB

```
Dim Sub1 As New Subset(New Integer() {1, 3, 4, 7, 9})
Dim Sub2 As New Subset(New Integer() {0, 2, 4})
Dim Sub3 As Subset = Sub1(Sub2)
'' sub3.Indices = 1, 4, 9
```

Logical Operations on Subsets

Operator `==` tests for equality of two subsets, and returns `true` if both subsets are the same length and all elements are equal; otherwise, `false`. Following the convention of the .NET Framework, if both objects are `null`, they test equal. Operator `!=` returns the logical negation of `==`. The `Equals()` member function also tests for equality.

Arithmetic Operations on Subsets

NMath provides overloaded arithmetic operators for subsets with their conventional meanings for those .NET languages that support them, and equivalent named methods for those that do not. Table 25 lists the equivalent operators and methods.

Table 25 – Arithmetic operators for subsets

Operator	Equivalent Named Method
+	Add()
-	Subtract()
*	Multiply()
/	Divide()
Unary -	Negate()
++	Increment()
--	Decrement()
&	Intersection()
|	Union()

Manipulating Subsets

The `Append()` method adds an index to the end of a subset:

Code Example – C#

```
sub.Append( 5 );
```

Code Example – VB

```
Subset.Append(5)
```

`Remove()` removes the first occurence of a given index from a subset. `Reverse()` reverses the indices of a subset. `Unique()` sorts the indices in a subset and removes any repetitions. Thus:

Code Example – C#

```
var sub = new Subset( new int[] { 0,5,3,2,7,5 } );
sub.Remove( 3 );
// sub.Indices = 0, 5, 2, 7, 5
sub.Reverse();
// sub.Indices = 5, 7, 2, 5, 0
sub.Unique();
// sub.Indices = 0, 2, 5, 7
```

Code Example – VB

```
Dim Subset As New Subset(New Integer() {0, 5, 3, 2, 7, 5})
Subset.Remove(3)
'' sub.Indices = 0, 5, 2, 7, 5
```

```
Subset.Reverse()
'' sub.Indices = 5, 7, 2, 5, 0
Subset.Unique()
'' sub.Indices = 0, 2, 5, 7
```

Similarly, `ToReverse()` returns a new subset containing the indices of a subset in the reverse order; `ToUnique()` returns a new subset containing the sorted indices of a subset, with all repetitions removed.

The `Repeat()` method creates a new subset by repeating the source subset until a given length is reached. For instance:

Code Example – C#

```
var sub1 = new Subset( 3 );
// sub1.Indices = 0,1,2
Subset sub2 = sub1.Repeat( 11 );
// sub2.Indices = 0,1,2,0,1,2,0,1,2,0,1
```

Code Example – VB

```
Dim Sub1 As New Subset(3)
'' sub1.Indices = 0,1,2
Dim Sub2 As Subset = Sub1.Repeat(11)
'' sub2.Indices = 0,1,2,0,1,2,0,1,2,0,1
```

The `Split()` method splits a source subset into an arbitrary array of subsets. The parameters are the number of subsets into which to split the source subset, and another subset the same length as the source subset, the *i*th element of which indicates into which bin to place the *i*th element of the source subset. For example:

Code Example – C#

```
var sub = new Subset( 10 );
// sub.Indices = 0,1,2,3,4,5,6,7,8,9
var bins =
  new Subset( new int[] { 3, 1, 0, 2, 2, 1, 1, 2, 3, 0 } );
Subset[] subsetArray = sub.Split( 4, bins );
// subsetArray[0] = 2,9
// subsetArray[1] = 1,5,6
// subsetArray[2] = 3,4,7
// subsetArray[3] = 0,8
```

Code Example – VB

```
Dim Subset As New Subset(10)
'' sub.Indices = 0,1,2,3,4,5,6,7,8,9
Dim Bins As New Subset(New Integer() {3, 1, 0, 2, 2, 1, 1, 2, 3,
0})
Dim SubsetArray() As Subset = Subset.Split(4, Bins)
'' subsetArray[0] = 2,9
'' subsetArray[1] = 1,5,6
'' subsetArray[2] = 3,4,7
```

```
'' subsetArray[3] = 0,8
```

Lastly, the `ToString()` returns a comma-delimited string list of the indices in a subset.

Groupings

The static `GetGroupings()` methods on **Subset** create subsets from factors. 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. See Section 37.10 for more information on factors and the `GetGroupings()` methods.

Random Samples

The static method `Sample( n )` returns a random shuffle of `0..n-1`. The returned **Subset** can be used to randomly reorder the rows in a data frame, as described in Section 37.8.

37.8 Accessing Sub-Frames

In addition to accessing individual elements, columns, or rows in a data frame (Section 37.6), class **DataFrame** provides a large number of member functions and indexers for accessing sub-frames containing any arbitrary subset of rows, columns, or both. Such methods and indexers accept **Slice** and **Subset** objects to indicate which rows and columns to return. (See Section 37.7 for more information on the **Subset** class.)

For example, `GetColumns()` returns a new data frame containing the columns indicated by a given **Slice** or **Subset**. For instance, if `df` has `5` columns, this code creates a new data frame containing columns `0`, `4`, and `5`:

Code Example – C#

```
var colSubset = new Subset( new int[] { 0, 4, 5 } );
DataFrame subDF = df.GetColumns( colSubset );
```

Code Example – VB

```
Dim ColSubset As New Subset(New Integer() {0, 4, 5})
Dim SubDF As DataFrame = DF.GetColumns(ColSubset)
```

Similarly, `GetRows()` returns a new data frame containing the rows indicated by a given **Slice** or **Subset**. Thus, this code gets every other row in the source data frame:

Code Example – C#

```
var rowSubset = new Range( 0, df.Rows - 1, 2 );
DataFrame subDF = df.GetRows( rowSubset );
```

Code Example – VB

```
Dim RowSubset As New Range(0, DF.Rows - 1, 2)
Dim SubDF As DataFrame = DF.GetRows(RowSubset)
```

Class **DataFrame** also provides a wide range of indexers for accessing subframes:

Code Example – C#

```
this[int colIndex, Slice rowSlice]
this[int colIndex, Subset rowSubset]
this[Slice rowSlice, Slice colSlice]
this[Subset rowSubset, Subset colSubset]
this[Slice rowSlice, Subset colSubset]
this[Subset rowSubset, Slice colSlice]
```

Code Example – VB

```
Item(ColIndex As Integer, RowSlice As Slice)
Item(ColIndex As Integer, RowSubset As Subset)
Item(RowSlice As Slice, ColSlice As Slice)
Item(RowSubset As Subset, ColSlice As Slice)
Item(RowSlice As Slice, ColSubset As Subset)
Item(RowSubset As Subset, ColSlice As Slice)
```

These indexers can be used to return any portion of a data frame. For example, this code gets a new data frame containing columns `3-8` in reverse order, and all rows where column `0` equals `Test1`:

Code Example – C#

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

var bArray = new bool[ df.Rows ];
for ( int i = 0; i < df.Rows; i++ )
{
  bArray[i] = ( df[0][i] == "Test1" );
}
var rowSubset = new Subset( bArray );

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

Code Example – VB

```
Dim ColRange As New Range(8, 3, -1)
```

```
Dim BArray() As Boolean = New Boolean(DF.Rows) {}
For I As Integer = 0 To DF.Rows - 1
  BArray(I) = (DF(0)(I) = "Test1")
Next
Dim RowSubset As New Subset(BArray)

Dim DF2 As DataFrame = DF(RowSubset, ColRange)
```

Finally, there is the `GetSubRow()` method. Whereas `GetRow()` returns an entire row for a given row index, `GetSubRow()` returns the portion of the row indicated by the given column **Slice** or **Subset**:

Code Example – C#

```
var colSlice = new Slice( 0, 3, 1 );
object[] subRow = df.GetSubRow( 3, colSlice );
```

Code Example – VB

```
Dim ColSlice As New Slice(0, 3, 1)
Dim SubRow As Object() = DF.GetSubRow(3, ColSlice)
```

37.9 Reordering DataFrames

The **DataFrame** class provides method for both sorting rows, and for arbitrarily reordering rows and columns.

Sorting Rows

The `SortRows()` method sorts the rows in a data frame according to a given ordered array of column indices. The first index is the primarily sort column, the second index is the secondary sort column, and so forth. For instance:

Code Example – C#

```
df.SortRows( 3, 0, 1 );
```

Code Example – VB

```
DF.SortRows(3, 0, 1)
```

By default, all sorting is in ascending order.

For more control, you can also pass an array of **SortingType** enumerated values (`Ascending` or `Descending`):

Code Example – C#

```
int[] colIndices = { 3, 0, 1 };
SortingType[] sortingTypes = { SortingType.Ascending,
                               SortingType.Descending,
                               SortingType.Ascending };
df.SortRows( colIndices, sortingTypes );
```

Code Example – VB

```
Dim ColIndices As Integer() = {3, 0, 1}
Dim SortingTypes As SortingType() = {SortingType.Ascending,
                                     SortingType.Descending,
                                     SortingType.Ascending}
DF.SortRows(ColIndices, SortingTypes)
```

Finally, the `SortRowsByKeys()` method sorts the rows in a data frame by their row keys, in the specified order:

Code Example – C#

```
df.SortRowsByKeys( SortingType.Ascending );
```

Code Example – VB

```
DF.SortRowsByKeys(SortingType.Ascending)
```

NOTE—StatsSettings.Sorting specifies the default SortingType.

Permuting Rows and Columns

The `PermuteColumns()` and `PermuteRows()` methods enable you to arbitrarily reorder the columns and rows in a data frame, respectively. Each method takes an array of indices. The array must be same length as the number of columns or rows, and contain unique indices. In both cases:

Code Example – C#

```
new[ permutation[i] ] = old[ i ]
```

Code Example – VB

```
New( permutation(i) ) = Old( i )
```

For example, assuming `df` has `3` columns, this code switches the last two columns:

Code Example – C#

```
df.PermuteColumns( 0, 2, 1 );
```

Code Example – VB

```
DF.PermuteColumns(0, 2, 1)
```

Assuming `df` has `5` rows, this code moves the second and fourth rows to the top:

Code Example – C#

```
df.PermuteRows( 2, 0, 3, 1, 4 );
```

Code Example – VB

```
DF.PermuteRows(2, 0, 3, 1, 4)
```

37.10 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
- 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:

Code Example – C#

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

Code Example – VB

```
Dim Levels As Object() = {"A", "B", "C"}
Dim Data As Integer() = {0, 0, 2, 1, 0, 2, 1}
```

Factors are usually constructed from a data frame column using the `GetFactor()` method, but they can also be constructed independently.

Creating Factors

The `GetFactor()` method on **DataFrame** accepts a column index or name and returns a **Factor** with levels for the sorted, unique elements in the given column:

Code Example – C#

```
Factor myColFactor = df.GetFactor( "myCol" );
```

Code Example – VB

```
Dim ColFactor As Factor = DF.GetFactor("myCol")
```

Alternatively, you can provide the factor levels yourself. The order is preserved. Thus:

Code Example – C#

```
var levels = new object[] { "Q1", "Q2", "Q3", "Q4" };
Factor myColFactor = df.GetFactor( "myCol", levels );
```

Code Example – VB

```
Dim Levels As Object() = {"Q1", "Q2", "Q3", "Q4"}
Dim ColFactor As Factor = DF.GetFactor("myCol", Levels)
```

An **InvalidArgumentException** is raised if the specified column contains a value not present in the given array of levels.

You can also construct a **Factor** independently of a **DataFrame**. For example, you can construct a **Factor** from an array of values:

Code Example – C#

```
var strArray = new object[] { 1, 1, 3, 2, 1, 3, 2 };
var factor = new Factor( strArray );
```

Code Example – VB

```
Dim StrArray As Object() = {1, 1, 3, 2, 1, 3, 2}
Dim Factor As New Factor(StrArray)
```

Factor levels are constructed from a sorted list of unique values in the passed array.

Alternatively, you can construct a **Factor** from an array of factor levels, and a data array consisting of indices into the factor levels:

Code Example – C#

```
var levels = new object[] { 1, 2, 3 };
var data = new int[] { 0, 0, 2, 1, 0, 2, 1 };
var factor = new Factor( levels, data );
```

Code Example – VB

```
Dim Levels As Object() = {1, 2, 3}
Dim Data As Integer() = {0, 0, 2, 1, 0, 2, 1}
Dim Factor As New Factor(Levels, Data)
```

An **InvalidArgumentException** is thrown if the given data array contains an invalid index.

Properties of Factors

The **Factor** class provides the following properties:

- `Data` gets the categorical data for the factor. Each element in the returned integer array is an index into `Levels`.
- `Levels` gets the levels of the factor as an array of objects.
- `Length` gets the length of the `Data` in the factor.
- `Name` gets and set the name of the factor.
- `NumberOfLevels` gets the number of levels in the factor.

Accessing Factors

A standard indexer is provided for accessing the element at a given index:

Code Example – C#

```
string str = (string)factor[2];
```

Code Example – VB

```
Dim Str As String = CType(Factor(2), String)
```

The indexer returns `Levels[ Data[index] ]`—that is, it returns the level at the given position.

Creating Groupings with Factors

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, suppose we weigh human subjects based on sex and age group. The data for 15 subject might look like this:

Table 26 – Sample data

	Male	Female
Child	45, 42	30, 35, 60, 40
Adult	182, 170	115, 130, 110
Senior	142, 155	115, 123

In a **DataFrame**, each observation would be a row, like so:

Code Example – C#

```
var df = new DataFrame();
df.AddColumn( new DFStringColumn( "Sex" ) );
df.AddColumn( new DFStringColumn( "AgeGroup" ));
df.AddColumn( new DFIntColumn( "Weight" ) );

df.AddRow( "John Smith", "Male", "Child", 45 );
df.AddRow( "Ruth Barnes", "Female", "Senior", 115 );
df.AddRow( "Jane Jones", "Female", "Adult", 115 );
df.AddRow( "Timmy Toddler", "Male", "Child", 42 );
df.AddRow( "Betsy Young", "Female", "Adult", 130 );
df.AddRow( "Arthur Smith", "Male", "Senior", 142 );
df.AddRow( "Lucy Young", "Female", "Child", 30 );
df.AddRow( "Emma Allen", "Female", "Child", 35 );
df.AddRow( "Roy Wilkenson", "Male", "Adult", 182 );
df.AddRow( "Susan Schwarz", "Female", "Senior", 110 );
df.AddRow( "Ming Tao", "Female", "Senior", 123 );
df.AddRow( "Johanna Glynn", "Female", "Child", 60 );
df.AddRow( "Randall Harvey", "Male", "Adult", 170 );
df.AddRow( "Tom Howard", "Male", "Senior", 155 );
df.AddRow( "Jennifer Watson", "Female", "Child", 40 );
```

Code Example – VB

```
Dim DF As New DataFrame()
DF.AddColumn(New DFStringColumn("Sex"))
DF.AddColumn(New DFStringColumn("AgeGroup"))
DF.AddColumn(New DFIntColumn("Weight"))

DF.AddRow("John Smith", "Male", "Child", 45)
DF.AddRow("Ruth Barnes", "Female", "Senior", 115)
DF.AddRow("Jane Jones", "Female", "Adult", 115)
DF.AddRow("Timmy Toddler", "Male", "Child", 42)
DF.AddRow("Betsy Young", "Female", "Adult", 130)
DF.AddRow("Arthur Smith", "Male", "Senior", 142)
DF.AddRow("Lucy Young", "Female", "Child", 30)
```

```
DF.AddRow("Emma Allen", "Female", "Child", 35)
DF.AddRow("Roy Wilkenson", "Male", "Adult", 182)
DF.AddRow("Susan Schwarz", "Female", "Senior", 110)
DF.AddRow("Ming Tao", "Female", "Senior", 123)
DF.AddRow("Johanna Glynn", "Female", "Child", 60)
DF.AddRow("Randall Harvey", "Male", "Adult", 170)
DF.AddRow("Tom Howard", "Male", "Senior", 155)
DF.AddRow("Jennifer Watson", "Female", "Child", 40)
```

In this case, we're using the subjects' names as row keys.

It is natural to construct factors from the `Sex` and `AgeGroup` columns:

Code Example – C#

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

Code Example – VB

```
Dim Sex As Factor = DF.GetFactor("Sex")
Dim Age As Factor = DF.GetFactor("AgeGroup")
```

We can then use these factors in conjunction with the `GetGroupings()` methods on **Subset** to create subsets representing the original rows, columns, and cells in Table 26:

Code Example – C#

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

Code Example – VB

```
Dim SexGroups As Subset() = Subset.GetGroupings(Sex)
Dim AgeGroups As Subset() = Subset.GetGroupings(Age)
Dim CellGroups As Subset(,) = Subset.GetGroupings(Sex, Age)
```

These subsets can then be used to operate on the relevant portions of the data frame. For instance, this code prints out row means, column means, and cell means for Table 26:

Code Example – C#

```
Console.WriteLine( "\nTABLE ROW MEANS" );
for ( int i = 0; i < age.NumberOfLevels; i++ )
{
  double mean = StatsFunctions.Mean(
    df[ df.IndexOfColumn( "Weight" ), ageGroups[i] ] );
  Console.WriteLine( "Mean for {0} = {1}", age.Levels[i], mean );
}
```

```
Console.WriteLine( "\nTABLE COLUMN MEANS" );
for ( int i = 0; i < sex.NumberOfLevels; i++ )
{
  double mean = StatsFunctions.Mean(
    df[ df.IndexOfColumn( "Weight" ), sexGroups[i] ] );
  Console.WriteLine( "Mean for {0} = {1}", sex.Levels[i], mean );
}

Console.WriteLine( "\nTABLE CELL MEANS" );
for ( int i = 0; i < sex.NumberOfLevels; i++ )
{
  for ( int j = 0; j < age.NumberOfLevels; j++ )
  {
    double mean = StatsFunctions.Mean(
      df[ df.IndexOfColumn( "Weight" ), cellGroups[i,j] ] );
    Console.WriteLine( "Mean for {0} {1} = {2}",
      sex.Levels[i], age.Levels[j], mean );
  }
}
```

Code Example – VB

```
Console.WriteLine(Environment.NewLine & "TABLE ROW MEANS")
For I As Integer = 0 To Age.NumberOfLevels - 1
  Dim Mean As Double =
    StatsFunctions.Mean(DF(DF.IndexOfColumn("Weight"),
    AgeGroups(I)))
  Console.WriteLine("Mean for {0} = {1}", Age.Levels(I), Mean)
Next

Console.WriteLine(Environment.NewLine & "TABLE COLUMN MEANS")
For I As Integer = 0 To Sex.NumberOfLevels - 1
  Dim Mean As Double =
    StatsFunctions.Mean(DF(DF.IndexOfColumn("Weight"),
    SexGroups(I)))

  Console.WriteLine("Mean for {0} = {1}", Sex.Levels(I), Mean)
Next

Console.WriteLine(Environment.NewLine & "TABLE CELL MEANS")
For I As Integer = 0 To Sex.NumberOfLevels - 1
  For J As Integer = 0 To Age.NumberOfLevels - 1
    Dim Mean As Double =
      StatsFunctions.Mean(DF(DF.IndexOfColumn("Weight"),
      CellGroups(I, J)))
    Console.WriteLine("Mean for {0} {1} = {2}", Sex.Levels(I),
      Age.Levels(J), Mean)
  Next
Next
```

The output is:

```
TABLE ROW MEANS
Mean for Adult = 149.25
Mean for Child = 42
Mean for Senior = 129

TABLE COLUMN MEANS
Mean for Female = 84.2222222222222
Mean for Male = 122.666666666667

TABLE CELL MEANS
Mean for Female Adult = 122.5
Mean for Female Child = 41.25
Mean for Female Senior = 116
Mean for Male Adult = 176
Mean for Male Child = 43.5
Mean for Male Senior = 148.5
```

See also the `Tabulate()` convenience methods on class **DataFrame**, as described in Section 37.11.

37.11 Cross-Tabulation

As described in Section 37.10, the `DataFrame.GetFactor()` method can be used in conjunction with `Subset.GetGroupings()` to access "cells" of data based on one or two grouping factors. This is such a common operation that class **DataFrame** also provides the `Tabulate()` methods as a convenience. This method accepts one or two grouping columns, a data column, and a delegate to apply to each data column subset. The results are returned in a new data frame.

Column Delegates

Overloads of `Tabulate()` accept static **IDFColumn** function delegates that return various types. For instance, this code encapsulates the static `StatsFunctions.Mean()` function in a `Func<IDFColumn, double>`:

Code Example – C#

```
var mean = new Func<IDFColumn, double>(StatsFunctions.Mean);
```

Code Example – VB

```
Dim Mean As Func(Of IDFColumn, Double) = AddressOf
  StatsFunctions.Mean
```

Most of the static descriptive statistics functions on class **StatsFunctions** (Chapter 38) have overloads that accept an **IDFColumn** and return a double, and so can be encapsulated in this way. A few return integers.

For example, this code encapsulates `StatsFunctions.Count()`, which returns the number of items in a column, in a `Func<IDFColumn, int>`:

Code Example – C#

```
var count = new Func<IDFColumn, int>(StatsFunctions.Count);
```

Code Example – VB

```
Dim Count As Func(Of IDFColumn, Integer) = AddressOf
  StatsFunctions.Count
```

Applying Column Delegates to Tabulated Data

The following code fills a **DataFrame** with some sales data:

Code Example – C#

```
var df = new DataFrame();
df.AddColumn( new DFStringColumn( "Product" ) );
df.AddColumn( new DFStringColumn("Month") );
df.AddColumn( new DFIntColumn( "Quantity" ) );
df.AddColumn( new DFNumericColumn( "Price" ) );
df.AddColumn( new DFNumericColumn( "TotalSale" ) );

int rowID = 0;
df.AddRow( rowID++, "Squash", "Nov", 40, 1.50, 60.0 );
df.AddRow( rowID++, "Carrots", "Nov", 15, 1.20, 18.0 );
df.AddRow( rowID++, "Squash", "Nov", 37, 1.45, 53.65 );
df.AddRow( rowID++, "Carrots", "Nov", 18, 1.25, 22.50 );
df.AddRow( rowID++, "Squash", "Nov", 34, 1.39, 47.26 );
df.AddRow( rowID++, "Carrots", "Dec", 20, 1.30, 26.0 );
df.AddRow( rowID++, "Squash", "Dec", 31, 1.30, 40.30 );
df.AddRow( rowID++, "Carrots", "Dec", 25, 1.40, 35.0 );
df.AddRow( rowID++, "Squash", "Dec", 25, 1.25, 31.25 );
df.AddRow( rowID++, "Carrots", "Dec", 30, 1.45, 43.50 );
df.AddRow( rowID++, "Carrots", "Jan", 33, 1.50, 49.50 );
df.AddRow( rowID++, "Squash", "Jan", 19, 1.21, 22.99 );
df.AddRow( rowID++, "Carrots", "Jan", 40, 1.65, 66.0 );
df.AddRow( rowID++, "Squash", "Jan", 15, 1.11, 16.65 );
df.AddRow( rowID++, "Carrots", "Jan", 47, 1.80, 84.60 );
df.AddRow( rowID++, "Squash", "Jan", 10, 1.00, 10.0 );
```

Code Example – VB

```
Dim DF As New DataFrame()
DF.AddColumn(New DFStringColumn("Product"))
```

```
DF.AddColumn(New DFStringColumn("Month"))
DF.AddColumn(New DFIntColumn("Quantity"))
DF.AddColumn(New DFNumericColumn("Price"))
DF.AddColumn(New DFNumericColumn("TotalSale"))

Dim RowID As Integer = 0
RowID += 1
DF.AddRow(RowID, "Squash", "Nov", 40, 1.5, 60.0)
RowID += 1
DF.AddRow(RowID, "Carrots", "Nov", 15, 1.2, 18.0)
RowID += 1
DF.AddRow(RowID, "Squash", "Nov", 37, 1.45, 53.65)
RowID += 1
DF.AddRow(RowID, "Carrots", "Nov", 18, 1.25, 22.5)
RowID += 1
DF.AddRow(RowID, "Squash", "Nov", 34, 1.39, 47.26)
RowID += 1
DF.AddRow(RowID, "Carrots", "Dec", 20, 1.3, 26.0)
RowID += 1
DF.AddRow(RowID, "Squash", "Dec", 31, 1.3, 40.3)
RowID += 1
DF.AddRow(RowID, "Carrots", "Dec", 25, 1.4, 35.0)
RowID += 1
DF.AddRow(RowID, "Squash", "Dec", 25, 1.25, 31.25)
RowID += 1
DF.AddRow(RowID, "Carrots", "Dec", 30, 1.45, 43.5)
RowID += 1
DF.AddRow(RowID, "Carrots", "Jan", 33, 1.5, 49.5)
RowID += 1
DF.AddRow(RowID, "Squash", "Jan", 19, 1.21, 22.99)
RowID += 1
DF.AddRow(RowID, "Carrots", "Jan", 40, 1.65, 66.0)
RowID += 1
DF.AddRow(RowID, "Squash", "Jan", 15, 1.11, 16.65)
RowID += 1
DF.AddRow(RowID, "Carrots", "Jan", 47, 1.8, 84.6)
RowID += 1
DF.AddRow(RowID, "Squash", "Jan", 10, 1.0, 10.0)
```

This code displays the average sales for each product:

Code Example – C#

```
var mean =
  new Func<IDFColumn, double>(StatsFunctions.Mean);
Console.WriteLine( df.Tabulate( "Product", "TotalSale", mean ) );
```

Code Example – VB

```
Dim Mean As Func(Of IDFColumn, Double) = AddressOf
  StatsFunctions.Mean
Console.WriteLine(DF.Tabulate("Product", "TotalSale", Mean))
```

The `Product` column is used as a grouping column, `TotalSale` contains the data, and the `mean` delegate returns the mean of the value in each cell. The output is:

```
#          Results
Carrots    43.1375
Squash     35.2625
Overall    39.2000
```

The `Tabulate()` methods return a new data frame. If only one grouping factor is specified, as in this example, the row keys are the sorted, unique factor levels. The only column, named `Results`, contains the results of applying the given delegate to the values in the data column tabulated for each level of the factor. A final row is appended, with key `Overall`, containing the results of applying the given delegate to all values in the data column.

Similarly, this code displays the number of observations in each cell for every combination of `Product` and `Month`:

Code Example – C#

```
var count =
  new Func<IDFColumn, int>( StatsFunctions.Count );
Console.WriteLine(
  df.Tabulate( "Product", "Month", "TotalSale", count );
```

Code Example – VB

```
Dim Count As Func(Of IDFColumn, Integer) = AddressOf
  StatsFunctions.Count
Console.WriteLine(DF.Tabulate("Product", "Month", "TotalSale",
  Count))
```

The `Product` and `Month` columns are used as grouping columns, `TotalSale` contains the data, and the `count` delegate returns the number of items in each cell.

The output is:

```
#          Dec  Jan  Nov  Overall
Carrots    3    3    2    8
Squash     2    3    3    8
Overall    5    6    5    16
```

When two grouping factors are specified, as in this case, the returned data frame has row keys containing the sorted, unique levels of the first grouping factor as strings. The columns in the data frame are named using the sorted, unique levels of the second grouping factor.

NOTE—In this example the alphabetic sorting of the Month names has put them into non-chronological order. In the months had been stored as DateTime objects in an DFDateTimeColumn, they would have been ordered chronologically.

Each cell in the data frame contains the results of applying the given delegate to the values in the data column tabulated for the appropriate combination of the two factors. A final column is appended, named `Overall`, containing the overall results for each level of the first factor. A final row is appended, with key `Overall`, containing the overall results for each level of the second factor. The lower right corner cell, accessed by indexer `this["Overall","Overall"]`, contains the results of applying the given delegate to all values in the data column.

37.12 Exporting Data from DataFrames

The contents of a data frame can be exported in various ways.

Exporting to a Matrix

The `ToDoubleMatrix()` method exports all the numeric data in a data frame to a **DoubleMatrix**. Non-numeric columns are ignored. For example, this code constructs a **DataFrame** from a **DoubleMatrix**, adds a column of string data, then exports the contents of the data frame to another **DoubleMatrix**:

Code Example – C#

```
var A = new DoubleMatrix( 8, 3, 0, .1 );
df = new DataFrame( A, new string[] { "A", "B", "C" } );

var col4 = new DFStringColumn( "D",
  new String[] { "x", "x", "x", "x", "x", "x", "x", "x" } );
df.AddColumn( col4 );

DoubleMatrix B = df.ToDoubleMatrix();
```

Code Example – VB

```
Dim A As New DoubleMatrix(8, 3, 0, 0.1)
DF = New DataFrame(A, New String() {"A", "B", "C"})

Dim Col4 As New DFStringColumn("D",
  New String() {"x", "x", "x", "x", "x", "x", "x", "x"})
DF.AddColumn(Col4)

Dim B As DoubleMatrix = DF.ToDoubleMatrix()
```

The two matrices are equal (`A == B`); the string column is ignored.

Exporting to a String

The `ToString()` method returns a formatted string representation of a data frame:

Code Example – C#

```
string str = df.ToString();
```

Code Example – VB

```
Dim Str As String = DF.ToString()
```

For more control, you can also indicate:

- whether to export column headers (the default is `true`)
- whether to export row keys (the default is `true`)
- the delimiter to use to separate columns (the default is tab-delimited)

For instance, this code exports the column headers, but not the row keys, and uses a comma delimiter:

Code Example – C#

```
string str = df.ToString( true, false, “,” );
```

Code Example – VB

```
Dim Str As String = DF.ToString(True, False, ",")
```

Convenience methods are also provided for persisting a text representation of a data frame to a text file. `Save()` exports the contents of the data frame to a given filename:

Code Example – C#

```
df.Save( “myData.txt” );
```

Code Example – VB

```
DF.Save("myData.txt")
```

Again, you can also indicate whether to export column header or row keys, and specify the column delimiter:

Code Example – C#

```
df.Save( “myData.txt”, true, false, “,” );
```

Code Example – VB

```
DF.Save("myData.txt", True, False, ",")
```

The `LaunchSaveFileDialog()` method allows the end user to specify the filename. The `OpenInEditor()` method programmatically opens a data frame in the default text editor on the user's system. The user can then edit the contents of the data frame. Lastly, the static `Load()` method imports a data frame from a text file:

Code Example – C#

```
DataFrame df = DataFrame.Load(  "myData.txt" );
```

Code Example – VB

```
Dim DF As DataFrame = DataFrame.Load("myData.txt")
```

Again, you can indicate whether the text file includes column headers and row keys, and the delimiter used to separate the columns.

Exporting to an ADO.NET DataTable

The `ToDataTable()` method exports the data in a data frame to an ADO.NET **DataTable** object. The row keys are placed in a **DataColumn** named `DFRowKeys`. Thus, this code:

Code Example – C#

```
var df = new DataFrame();
df.AddColumn(
  new DFNumericColumn( "ids", new DoubleVector( 3, 3, -1 )));
df.AddColumn(
  new DFStringColumn( "names", "a", "b", "c" ));
df.AddColumn(
  new DFBoolColumn( "bools", true, false, true ));
df.SetRowKeys( new String[] { "Row1", "Row2", "Row3" } );
DataTable table = df.ToDataTable();
```

Code Example – VB

```
Dim DF As New DataFrame()
DF.AddColumn(
  New DFNumericColumn("ids", New DoubleVector(3, 3, -1)))
DF.AddColumn(
  New DFStringColumn("names", "a", "b", "c"))
DF.AddColumn(
  New DFBoolColumn("bools", True, False, True))
DF.SetRowKeys(New String() {"Row1", "Row2", "Row3"})
Dim Table As DataTable = DF.ToDataTable()
```

returns a **DataTable** that looks like this:

```
name: CenterSpace.NMath.Core.DataFrame
#  DFRowKeys ids     names   bools
1  Row1      3.0000  a       True
2  Row2      2.0000  b       False
```

```
3  Row3      1.0000  c       True
```

If no name is assigned to a data frame before `ToDataTable()` is called, the name of the **DataTable** is set to the type: `CenterSpace.NMath.Core.DataFrame`.

Binary and SOAP Serialization

Class **DataFrame** implements the **ISerializable** interface to control serialization and deserialization. Common Language Runtime (CLR) serialization **Formatter** classes call the provided `GetObjectData()` method at serialization time to populate a **SerializationInfo** object with all the data required to represent a **DataFrame**. For example, the **BinaryFormatter** class provides `Serialize()` and `Deserialize()` methods for persisting an object in binary format to a given stream. For example, this code serializes a data frame to a file:

Code Example – C#

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

FileStream binStream = File.Create( "myData.dat" );
var binFormatter = new BinaryFormatter();
binFormatter.Serialize( binStream, df );
binStream.Close();
```

Code Example – VB

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

Dim BinStream As FileStream = File.Create("myData.dat")
Dim BinFormatter As New BinaryFormatter()
BinFormatter.Serialize(BinStream, DF)
BinStream.Close()
```

This code restores the data frame from the file:

Code Example – C#

```
binStream = File.OpenRead( "myData.dat" );
DataFrame df2 = (DataFrame)binFormatter.Deserialize( binStream );
binStream.Close();
File.Delete( "myData.dat" );
```

Code Example – VB

```
BinStream = File.OpenRead("myData.dat")
Dim DF2 As DataFrame = CType(BinFormatter.Deserialize(BinStream),
DataFrame)
BinStream.Close()
File.Delete("myData.dat")
```

Similarly, the **SoapFormatter** class persists an object in SOAP format to a given stream. Thus:

Code Example – C#

```
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;

FileStream xmlStream = File.Create( "myData.xml" );
var xmlFormatter = new SoapFormatter();
xmlFormatter.Serialize( xmlStream, df );
xmlStream.Close();
```

Code Example – VB

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Soap

Dim XMLStream As FileStream = File.Create("myData.xml")
Dim XMLFormatter As New SoapFormatter()
XMLFormatter.Serialize(XMLStream, DF)
XMLStream.Close()
```

This code restores the data frame from the file:

Code Example – C#

```
xmlStream = File.OpenRead( "myData.xml" );
DataFrame df2 = (DataFrame)xmlFormatter.Deserialize( xmlStream )
xmlStream.Close();
File.Delete( "myData.xml" );
```

Code Example – VB

```
XMLStream = File.OpenRead("myData.xml")
Dim DF2 As DataFrame = CType(XMLFormatter.Deserialize(XMLStream),
DataFrame)
XMLStream.Close()
File.Delete("myData.xml")
```

CHAPTER 38.
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 column in a **DataFrame** (Chapter 37). For example:

Code Example – C#

```
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" ] );

// mean1 == mean2 == mean3
```

Code Example – VB

```
Dim DblArray() As Double = {1.12, -2.0, 3.88, 1.2, 15.345}
Dim Mean1 As Double = StatsFunctions.Mean(DblArray)

Dim V As New DoubleVector("1.12 -2.0 3.88 1.2 15.345")
Dim Mean2 As Double = StatsFunctions.Mean(V)

Dim DF As New DataFrame()
DF.AddColumn(New DFNumericColumn("myData", 1.12, -2.0, 3.88, 1.2,
  15.345))
Dim Mean3 As Double = StatsFunctions.Mean(DF("myData"))

'' mean1 == mean2 == mean3
```

In this chapter, where `data` is used in code examples, it should be understood to be an instance of any of these three types.

38.1 Column Types

Most functions in class **StatsFunctions** require numeric data, although they accept any instance of **IDFColumn**. If a column is not an instance of **DFIntColumn** or **DFNumericColumn**, an attempt is made to convert the data to double using `System.Convert.ToDouble()`.

NOTE—An NMathFormatException is raised if the data cannot be converted to double.

For instance, these functions will work with a **DFStringColumn** containing numbers represented as strings.

Code Example – C#

```
var col =
  new DFStringColumn( "Col1", "1.5", "2", "1.33", "4.76" );
double mean = StatsFunctions.Mean( col );
```

Code Example – VB

```
Dim Col As New DFStringColumn("Col1", "1.5", "2", "1.33", "4.76")
Dim Mean As Double = StatsFunctions.Mean(Col)
```

However, there is a processing penalty due to such type conversion. If you need to perform many statistical functions on a column, first create a new **DFIntColumn** or **DFNumericColumn** from your data column, so type conversion occurs only once. For example, if column `4` in data frame `df` is a **DFGenericColumn** containing decimal types, this works:

Code Example – C#

```
double mean = StatsFunctions.Mean( df[4] );
double stdev = StatsFunctions.StandardDeviation( df[4] );
```

Code Example – VB

```
Dim Mean As Double = StatsFunctions.Mean(DF(4))
Dim StdDev As Double = StatsFunctions.StandardDeviation(DF(4))
```

but the decimal data is converted to doubles twice. This code first creates a new **DFNumericColumn** containing doubles from the generic column, then computes the statistics:

Code Example – C#

```
var col = new DFNumericColumn( df[4].Name, df[4] );
double mean = StatsFunctions.Mean( col );
double stdev = StatsFunctions.StandardDeviation( col );
```

Code Example – VB

```
Dim Col As New DFNumericColumn(DF(4).Name, DFBoolColumn(4))
```

```
Dim Mean As Double = StatsFunctions.Mean(Col)
Dim StdDev As Double = StatsFunctions.StandardDeviation(Col)
```

In some cases, you may want to replace the original generic column in the data frame with the new **DFNumericColumn**:

Code Example – C#

```
df.RemoveColumn( 4 );
df.InsertColumn( 4, col );
double mean = StatsFunctions.Mean( df[4] );
double stdev = StatsFunctions.StandardDeviation( df[4] );
```

Code Example – VB

```
DF.RemoveColumn(4)
DF.InsertColumn(4, Col)
Dim Mean As Double = StatsFunctions.Mean(DF(4))
Dim StdDev As Double = StatsFunctions.StandardDeviation(DF(4))
```

Note that sometimes you may not even be aware that your data is stored in a generic column. (You can always return the type of a column using the `ColumnType` property.) This is most likely to occur when you read data from a text file or database directly into a **DataFrame**. For example, if your database stores data using SQL `NUMERIC` or `DECIMAL` types, these get mapped to `System.Decimal` in ADO. NMath does not silently convert decimals to doubles, because of the loss of precision, so they are stored in the dataframe as objects in a **DFGenericColumn**. If you intend to perform multiple statistical functions on the data, convert the column to a **DFNumericColumn** first, as shown above.

38.2 Missing Values

Most functions in class **StatsFunctions** are accompanied by a paired function that ignores missing values, such as `Double.NaN` in a **DoubleVector**, **DFNumericColumn**, or array of doubles. 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.

Code Example – C#

```
var v = new DoubleVector( "[ 3.2 1.0 Double.NaN 4.5 -1.2 ]");

double mean1 = StatsFunctions.Mean( v );
// mean1 = Double.NaN

double mean2 = StatsFunctions.NaNMean( v );
// mean2 = 1.875
```

Code Example – VB

```
Dim V As New DoubleVector("[ 3.2 1.0 Double.NaN 4.5 -1.2 ]")

Dim Mean1 As Double = StatsFunctions.Mean(V)
'' mean1 = Double.NaN

Dim Mean2 As Double = StatsFunctions.NaNMean(V)
'' mean2 = 1.875
```

The provided convenience method `NaNCheck()` returns `true` if a given data set contains any missing values. `NaNRemove()` creates a copy of a data set with missing values removed. For two-dimensional data sets, such as matrices and data frames, `NaNRemoveCols()` creates a copy with only those columns that do not contain missing values. `NaNRemoveRows()` removes any rows containing missing data. The `CleanCols()` and `CleanRows()` methods on class **DataFrame** have the same effect.

As described in Section 37.1, data frame column types enable you to specify how missing values are represented within a particular column instance, or for all columns of a particular type. For example, this column stores numeric data in a string column, and uses `NA` to indicate a missing value:

Code Example – C#

```
var col =
  new DFStringColumn( "myCol", "32.1", "NA", "6.0", "34" );
```

Code Example – VB

```
Dim Col As New DFStringColumn("myCol", "32.1", "NA", "6.0", "34")
```

This code identifies the missing value string, then computes the mean, ignoring missing values:

Code Example – C#

```
col.MissingValue = "NA";
double mean = StatsFunctions.NaNMean( col );
```

Code Example – VB

```
Col.MissingValue = "NA"
Dim Mean As Double = StatsFunctions.NaNMean(Col)
```

Because the column is not an instance of **DFIntColumn** or **DFNumericColumn**, an attempt is made to convert the data to double using `System.Convert.ToDouble()` (Section 38.1). If `StatsFunctions.Mean()` was used, instead of `StatsFunctions.NaNMean()`, or if `col.MissingValue` was set to something other than `NA` (for example, the default value is `"."`), an exception would be thrown.

38.3 Counts and Sums

The static `Count()` method on class **StatsFunctions** returns the number of elements in a data set:

Code Example – C#

```
int numElements = StatsFunctions.Count( data );
```

Code Example – VB

```
Dim NumElements As Integer = StatsFunctions.Count(Data)
```

`Counts()` returns an **IDictionary** of key-value pairs in which the keys are the unique elements in a given data set, and the values are the counts for each element.

`CountIf()` calculates how many elements in a data set return `true` when a logical function is applied. For example, suppose `MeetsThreshold()` is a method that returns `true` if a given numeric value is greater than `100`:

Code Example – C#

```
public bool MeetsThreshold( double x )
{
  return ( x > 100 );
}
```

Code Example – VB

```
Public Function MeetsThreshold(X As Double) As Boolean
  Return (X > 100)
End Function
```

This code counts the number of elements in a data set that meet the criterion:

Code Example – C#

```
int num = StatsFunctions.CountIf( data, new
  new Func<double, bool>( MeetsThreshold ) );
```

Code Example – VB

```
Dim Num As Integer = StatsFunctions.CountIf(DataArray,
  New Func(Of Double, Boolean)(AddressOf MeetsThreshold))
```

Similarly, the static `Sum()` method sums the elements in a data set. `SumIf()` sums the elements in a data set that return `true` when a logical function is applied:

Code Example – C#

```
double sum = StatsFunctions.SumIf( data, new
  new Func<double, bool>( MeetsThreshold ) );
```

Code Example – VB

```
Dim Sum As Double = StatsFunctions.SumIf(DataColumn,
  New Func(Of Double, Boolean)(AddressOf MeetsThreshold))
```

An overload of `SumIf()` sums the elements in one data set based on evaluating a logical function on another data set. For instance, this code sums the elements in `data2` that correspond to those elements in `data` where `MeetsThreshold()` returns `true`:

Code Example – C#

```
double sum = StatsFunctions.SumIf( data, function, data2 );
```

Code Example – VB

```
Dim Sum As Double = StatsFunctions.SumIf(DataVector, MyFunction,
  data2)
```

A **MismatchedSizeException** is raised if the two data sets do not have the same number of elements.

38.4 Min/Max Functions

Class **StatsFunctions** provides static min/max finding methods that return the integer index of the element in a data set that meets the appropriate criterion:

- `MaxIndex()` returns the index of the element with the greatest value.
- `MinIndex()` returns the index of the element with the smallest value.
- `MaxAbsIndex()` returns the index of the element with the greatest absolute value.
- `MinAbsIndex()` returns the index of the element with the smallest absolute value.

Min/max value methods `MaxValue()`, `MinValue()`, `MaxAbsValue()`, and `MinAbsValue()` return the value of the element that meets the appropriate criterion.

38.5 Ranks, Percentiles, Deciles, and Quartiles

The static `Ranks()` method on class **StatsFunctions** returns the rank of each element in a data set an as array of integers. For example:

Code Example – C#

```
int[] ranks  = StatsFunctions.Ranks( data );
```

Code Example – VB

```
Dim Ranks() As Integer = StatsFunctions.Ranks(MyData)
```

By default, the ranks are calculated using ascending order. Alternatively, you can specify a sort order using a value from the **SortingType** enumeration. Thus:

Code Example – C#

```
int[] ranks  =
  StatsFunctions.Ranks( data, SortingType.Descending );
```

Code Example – VB

```
Dim Ranks As Integer() = StatsFunctions.Ranks(MyData,
  SortingType.Descending)
```

NOTE—StatsSettings.Sorting specifies the default SortingType.

The `Rank()` method returns where a given value *would* rank within a data set, if it were part of the data set. Again, the sorting order can be specified using a value from the **SortingType** enumeration. For instance:

Code Example – C#

```
double x = 5.342;
int rank = StatsFunctions.Rank( data, x, SortingType.Descending );
```

Code Example – VB

```
Dim X As Double = 5.342
Dim Rank As Integer = StatsFunctions.Rank(MyData, X,
  SortingType.Descending)
```

`Percentile()` calculates the value at the nth percentile of the elements in a data set, where $0 \le n \le 1$. For example, to find the value at the 95th percentile:

Code Example – C#

```
double x = StatsFunctions.Percentile( data, 0.95 );
```

Code Example – VB

```
Dim X As Double = StatsFunctions.Percentile(MyData, 0.95)
```

`PercentileRank()` performs the inverse calculation, returning the percentile a given value would have if it were part of the data set:

Code Example – C#

```
double x = 23.653;
double percentile = StatsFunctions.Percentile( data, x );
```

Code Example – VB

```
Dim X As Double = 23.653
Dim Percentile As Double = StatsFunctions.Percentile(MyData, X)
```

The returned percentile value is between `0` and `1`.

Similarly, `Decile()` calculates a given decile, specified as an integer between `0` and `10`, of the elements in a data set. `Quartile()` calculates a given quartile, specified as an integer between `0` and `4`. For example, this code finds the third quartile value:

Code Example – C#

```
double x = StatsFunctions.Quartile( data, 3 );
```

Code Example – VB

```
Dim X As Double = StatsFunctions.Quartile(MyData, 3)
```

38.6 Central Tendency

Measures of *central tendency* are measures of the location of the middle or the center of a data set. For example, the static `Mean()` method on class **StatsFunctions** computes the arithmetic mean (average) of the elements in a data set:

Code Example – C#

```
double mean = StatsFunctions.Mean( data );
```

Code Example – VB

```
Dim Mean As Double = StatsFunctions.Mean(MyData)
```

`Median()` calculates the median of the elements in a data set:

Code Example – C#

```
double median = StatsFunctions.Median( data );
```

Code Example – VB

```
Dim Median As Double = StatsFunctions.Median(MyData)
```

The median is the middle of the set—half the values are above the median and half are below the median. If there are an even number of elements, `Median()` returns the average of the middle two elements.

`Mode()` determines the most frequently occurring value in a data set:

Code Example – C#

```
double mode = StatsFunctions.Mode( data );
```

Code Example – VB

```
Dim Mode As Double = StatsFunctions.Mode(MyData)
```

`GeometricMean()` calculates the geometric mean.

$$\frac{n}{\sqrt[n]{x_1 \cdot x_1 \ldots x_n}}$$

`HarmonicMean()` calculates the harmonic mean.

$$\frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}}$$

`TrimmedMean()` calculates the mean of a data set after the specified trimming. A trimmed mean is calculated by discarding a certain percentage of the lowest and the highest values and then computing the mean of the remaining values. For example, a mean trimmed 50% is computed by discarding the lower and higher 25% of the values and taking the mean of the remaining values. `TrimmedMean()` takes a trimming parameter, which is a value between `0.0` and `1.0`. For example, this code computes the mean trimmed 50%:

Code Example – C#

```
double trimMean = StatsFunctions.TrimmedMean( data, 0.50 );
```

Code Example – VB

```
Dim TrimMean As Double = StatsFunctions.TrimmedMean(MyData, 0.5)
```

The median is the mean trimmed `1.0`, and the arithmetic mean is the mean trimmed `0.0`.

`WeightedMean()` calculates the weighted average of all the elements in a data set using a given set of corresponding weights. The weighted mean is calculated as

$$\frac{w_1 x_1 + w_2 x_2 + \ldots + w_n x_n}{w_1 + w_2 + \ldots + w_n}$$

For instance:

Code Example – C#

```
var v = new DoubleVector( "-0.3 -0.03 4 2.8 -12.3 -5 3 10" );
var weights = new DoubleVector( "1 2 3 4 2 1 3 4" );
double weightedMean = StatsFunctions.WeightedMean( v, weights ));
```

Code Example – VB

```
Dim V As New DoubleVector("-0.3 -0.03 4 2.8 -12.3 -5 3 10")
Dim Weights As New DoubleVector("1 2 3 4 2 1 3 4")
Dim WeightedMean As Double = StatsFunctions.WeightedMean(V,
Weights)
```

A **MismatchedSizeException** is raised if the number of weights does not equal the number of elements in the data set. Note that if all the weights are equal, the weighted mean is the same as the arithmetic mean.

Lastly, `RMS()` calculates the root mean square of the elements in a data set. RMS, sometimes called the quadratic mean, is the square root of the mean squared value.

38.7 Spread

Measures of *spread* are measures of the degree values in the data set differ from each other. For example, the static `SumOfSquaredErrors()` method on class **StatsFunctions** calculates the sum of squared errors (SSE) of the elements in the data set. SSE is the sum of the squared differences between each element and the mean.

`StandardDeviation()` computes the biased standard deviation of the elements in a data set.

$$\sqrt{\frac{\mathrm{SSE}}{n}}$$

For instance:

Code Example – C#

```
double stdev = StatsFunctions.StandardDeviation( data );
```

Code Example – VB

```
Dim StdDev As Double = StatsFunctions.StandardDeviation(MyData)
```

Alternatively, you can specify the unbiased standard deviation

$$\sqrt{\frac{\mathrm{SSE}}{n-1}}$$

using a value from the **BiasType** enumeration:

Code Example – C#

```
double stdev =
  StatsFunctions.StandardDeviation( data, BiasType.Unbiased );
```

Code Example – VB

```
Dim StdDev As Double = StatsFunctions.StandardDeviation(MyData,
BiasType.Unbiased)
```

NOTE—StatsSettings.Bias specifies the default BiasType.

`Variance()` calculates the variance of the elements in a data set. Variance is the square of the standard deviation. Again, you can specify a biased or unbiased estimator using values from the **BiasType** enumeration.

`MeanDeviation()` calculates the mean deviation of the elements in a data set. The mean deviation is the mean of the absolute deviations about the mean. The mean deviation is defined by

$$\frac{1}{n}\sum_{i=1}^{n}\left|x_i - \bar{x}\right|$$

Similarly, `MedianDeviationFromMean()` calculates the median of the absolute deviations from the mean. `MedianDeviationFromMedian()` calculates the median of the absolute deviations from the median.

Lastly, `InterquartileRange()` returns the difference between the median of the highest half and the median of the lowest half of the elements in a data set:

Code Example – C#

```
double iqr = StatsFunctions.InterQuartileRange( data );
```

Code Example – VB

```
Dim IQR As Double = StatsFunctions.InterquartileRange(MyData)
```

38.8 Shape

The static `Skewness()` method on class **StatsFunctions** computes the skewness of the elements in a data set. Skewness is the degree of asymmetry of a distribution. A distribution is skewed if one of its tails is longer than the other. Thus:

Code Example – C#

```
double skewness = StatsFunctions.Skewness( data );
```

Code Example – VB

```
Dim Skewness As Double = StatsFunctions.Skewness(MyData)
```

By default, `Skewness()` uses a biased estimator of the standard deviation (Section 38.7). Alternatively, you can specify the unbiased standard deviation using a value from the **BiasType** enumeration:

Code Example – C#

```
double skewness =
  StatsFunctions.Skewness( data, BiasType.Unbiased );
```

Code Example – VB

```
Dim Skewness As Double = StatsFunctions.Skewness(MyData,
  BiasType.Unbiased)
```

NOTE—StatsSettings.Bias specifies the default BiasType.

`Kurtosis()` calculates the kurtosis of the elements in a data set. Kurtosis is a measure of the degree of peakedness of a distribution. Again, a biased estimator of the standard deviation is used by default—you can specify the unbiased standard deviation using a value from the **BiasType** enumeration.

Finally, `CentralMoment()` returns the moment about the mean of a data set specified by a positive integer *order*. The first central moment is equal to zero. The second central moment is the variance. The third central moment is the skewness. The fourth central moment is the kurtosis.

38.9 Covariance, Correlation, and Autocorrelation

The static `Covariance()` method on class **StatsFunctions** computes the covariance of two data sets. Covariance is a measure of the tendency of two data sets to vary together, and is defined by

$$\mathrm{cov}_{\mathrm{x,\,y}} = \frac{\sum (x_i - \mu_x)(y_i - \mu_y)}{n}$$

Each deviation score in the first data set is multiplied by the corresponding deviation score in the second data set. For example:

Code Example – C#

```
double cov = StatsFunctions.Covariance( data1, data2 );
```

Code Example – VB

```
Dim Cov As Double = StatsFunctions.Covariance(MyData1, MyData2)
```

You can also specify a biased or unbiased estimator using values from the **BiasType** enumeration.

`CovarianceMatrix()` creates a square, symmetric matrix containing the variances and covariances of the columns in a given data matrix. The diagonal elements represent the variances for the columns; the off-diagonal elements represent the covariances of each pair of columns.

`Correlation()` calculates the correlation between two data sets. Correlation is covariance standardized by dividing by the standard deviation of each data set:

$$\mathrm{cor}_{x,\,y} = \frac{\mathrm{cov}_{x,\,y}}{S_x S_y}$$

The resultant value is the Pearson product-moment correlation coefficient, more commonly known simply as the correlation.

`Spearmans()` calculates the Spearman rank correlation coefficient, commonly known as *Spearman's rho*. Spearman's rho differs from Pearson's correlation only in that the computation is done after the values in the data set are converted to ranks (Section 38.5).

`Fisher()` calculates the Fisher transformation at a given value, which can be used to perform hypothesis testing on the correlation coefficient. `FisherInv()` calculates the inverse Fisher transformation.

`Cronbach()` calculates the standardized Cronbach's alpha test for reliability.

Autocorrelation is the correlation between members of a time series of observations. Class **StatsFunctions** provides two static methods for computing first-order autocorrelation:

- `DurbinWatson()` calculates the Durbin-Watson statistic for the elements in a data set.
- `VonNeumannRatio()` calculates the Von Neumann ratio for the elements in a data set.

For instance:

Code Example – C#

```
double dw = StatsFunctions.DurbinWatson( data );
double vnr = StatsFunctions.VonNeumannRatio( data );
```

Code Example – VB

```
Dim DW As Double = StatsFunctions.DurbinWatson(MyData)
Dim VNR As Double = StatsFunctions.VonNeumannRatio(MyData)
```

38.10 Sorting

The static `Sort()` method on class **StatsFunctions** sorts the elements of a data set in ascending or descending order using the quicksort algorithm and returns a new data set containing the result. The sort order is specified using a value from the **SortingType** enumeration.

For example:

Code Example – C#

```
var v = new DoubleVector( "5 7 1 3 9 4 5 2 1 0 11 3" );
v = StatsFunctions.Sort( v, SortingType.Descending );
```

Code Example – VB

```
Dim V As New DoubleVector("5 7 1 3 9 4 5 2 1 0 11 3")
V = StatsFunctions.Sort( V, SortingType.Descending )
```

NOTE—StatsSettings.Sorting specifies the default SortingType.

38.11 Logical Functions

The static `If()` method on class **StatsFunctions** creates an array of boolean values determined by applying a given logical function to the elements in a data set.

For example, suppose `OnInterval01()` is a method that returns `true` if a given numeric value is between `0` and `1`:

Code Example – C#

```
public bool OnInterval01( double x )
{
  return ( ( x >= 0 ) && ( x <= 1 ) );
}
```

Code Example – VB

```
Public Function OnInterval01(X As Double) As Boolean
  Return ((X >= 0) & (X <= 1))
End Function
```

This code creates an array of boolean values by applying the criterion to a data set:

Code Example – C#

```
bool[] bArray = StatsFunctions.If( data, new
  new Func<double, bool>( OnInterval01 ) );
```

Code Example – VB

```
Dim BArray() As Boolean = StatsFunctions.If(MyData,
  New Func(Of Double, Boolean)(AddressOf OnInterval01))
```

As described in Section 37.7, the resultant boolean array could be used to create a **Subset** containing the indices of all `true` elements in the array. The subset could then be used to create a sub-frame from a **DataFrame** containing the rows or columns than meet the criterion.

An overload of `If()` creates a new data set by applying a logical function to the elements of another data set. Elements in the original data set that return `true` are set to a given true value in the new data set; elements that return `false` are not changed.

For instance, suppose `GreaterThan100()` is a method that returns `true` if a given numeric value is greater than `100`. This code creates a new data in which all values in **DoubleVector** `data` that are greater than `100` are set to NaN:

Code Example – C#

```
DoubleVector data2 = StatsFunctions.If( data,
  new Func<double, bool>( GreaterThan100 ),
  Double.NaN );
```

Code Example – VB

```
Dim MyData2 As DoubleVector = StatsFunctions.If(MyData,
  New Func(Of Double, Boolean)(AddressOf GreaterThan100),
  Double.NaN)
```

You can also supply a false value, in which case elements in the original data set that return `false` are set to that value.

Static `CountIf()` and `SumIf()` methods are also provided on class **StatsFunctions**. See Section 38.3 for more information.

CHAPTER 39. SPECIAL FUNCTIONS

In addition to the descriptive statistics described in Chapter 38, class **StatsFunctions** also provides several special functions useful for statistical computation, including combinatorial functions, the beta function, and the gamma function.

39.1 Combinatorial Functions

The static `Factorial()` method on class **StatsFunctions** returns `n!`, the number of ways that *n* objects can be permuted. A lookup table is used for $n < 24$ for faster access. For example:

Code Example – C# factorial

```
int i = StatsFunctions.Factorial( 20 );
// i = 2,432,902,008,176,640,000
```

`FactorialLn()` returns the natural log factorial of *n*, $\ln(n!)$.

The static `Binomial()` method returns the binomial coefficient. The binomial coefficient $_{n}C_{m}$ ("*n* choose *m*") is the number of ways of picking *m* unordered outcomes from *n* possibilities:

$$_{n}C_{m} = \frac{n!}{(n-m)!m!}$$

For instance:

Code Example – C# binomial

```
int nCm = StatsFunctions.Binomial( 6, 4 );
```

`BinomialLn()` returns the natural log of the binomial coefficient.

39.2 Gamma Function

The static `GammaLn()` method on class **StatsFunctions** evaluates the log of the gamma function $\Gamma(x)$ at a value x. The gamma function is an extension of the factorial function to complex and real number arguments.

The “complete” gamma function $\Gamma(x)$ can be generalized to the incomplete gamma function $\Gamma(a, x)$, such that $\Gamma(a) = \Gamma(a, 0)$. The “lower” incomplete gamma function is given by:

$$P(x, a) = \frac{1}{\Gamma(a)}\int_0^x t^{a-1} e^{-t} dt$$

`IncompleteGamma()` returns the value of the lower regularized incomplete gamma function.

39.3 Beta Function

The static `Beta()` method on class **StatsFunctions** method evaluates the beta function $B(n, m)$, which is related to the gamma function $\Gamma(x)$ as follows:

$$B(n, m) = \frac{\Gamma(n)\Gamma(n)}{\Gamma(n+m)} = \frac{(n-1)!(m-1)!}{(n+m-1)!}$$

The incomplete beta function $B_z(n, m)$ is a generalization of the beta function:

$$B_z(a, x) = \int_0^z u^{a-1}(1-u)^{b-1} du$$

`IncompleteBeta()` returns the value of the incomplete beta function.

CHAPTER 40.
PROBABILITY DISTRIBUTIONS

NMath Stats 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 beta, binomial, chi-square (χ^2), exponential, *F*, gamma, geometric, Johnson, logistic, log-normal, negative binomial, normal (Gaussian), Poisson, Student's *t*, triangular, uniform, and Weibull distributions. The distribution classes share a common interface, so once you learn how to use one distribution class, it's easy to use any of the others.

This chapter describes the distribution classes and how to use them. This chapter also describes how to create correlated sets of random variables drawn from different distributions.

40.1 Distribution Classes

The **NMath Stats** probability distribution classes are listed in Table 27.

Table 27 – Probability Distribution Classes

Class	Distribution
BetaDistribution	Beta distribution
BinomialDistribution	Binomial distribution
ChiSquareDistribution	Chi-Square (χ^2) distribution
ExponentialDistribution	Exponential distribution
FDistribution	*F* distribution
GammaDistribution	Gamma distribution
GeometricDistribution	Geometric distribution
JohnsonDistribution	Johnson distribution
LogisticDistribution	Logistic distribution
LognormalDistribution	Log-normal distribution

Table 27 – Probability Distribution Classes

Class	Distribution
NegativeBinomialDistribution	Negative Binomial distribution
NormalDistribution	Normal (Gaussian) distribution
PoissonDistribution	Poisson distribution
TDistribution	Student's *t* distribution
TriangularDistribution	Triangular distribution
UniformDistribution	Uniform distribution
WeibullDistribution	Weibull distribution

All distribution classes share a common interface. Class **ProbabilityDistribution** is the abstract base class for the distribution classes, and provides the following abstract methods implemented by the derived classes:

- `PDF()` computes the probability density function at a given *x*.
- `CDF()` computes the cumulative distribution function at a given *x*.
- `InverseCDF()` computes the inverse cumulative distribution function for a given probability *p*—that is, it returns *x* such that `CDF( x ) = p`.

In addition, all **NMath Stats** distribution classes implement the **IRandomVariableMoments** interface, which provides the following read-only properties:

- `Mean` gets the mean of the distribution.
- `Variance` gets the variance of the distribution.
- `Kurtosis` gets the kurtosis of the distribution.
- `Skewness` gets the skewness of the distribution.

Variance is the square of the standard deviation. *Kurtosis* is a measure of the degree of peakednesss of a distribution; *skewness* is a measure of the degree of asymmetry.

Once you have constructed a derived distribution type, you can query it for the PDF, CDF, inverse CDF, and random variable moments. For example, this code constructs a **NormalDistribution** with mean `0` and variance `1`, then queries it:

Code Example – C# normal distribution

```
var dist = new NormalDistribution( 0, 1 );
double pdf = dist.PDF( 0 );
double cdf = dist.CDF( 0 );
double invCdf = dist.InverseCDF( .5 );
double mean = dist.Mean;
double var = dist.Variance;
double kurt = dist.Kurtosis;
double skew = dist.Skewness;
```

Code Example – VB normal distribution

```
Dim Dist As New NormalDistribution(0, 1)
Dim PDF As Double = Dist.PDF(0)
Dim CDF As Double = Dist.CDF(0)
Dim InvCDF As Double = Dist.InverseCDF(0.5)
Dim Mean As Double = Dist.Mean
Dim Var As Double = Dist.Variance
Dim Kurt As Double = Dist.Kurtosis
Dim Skew As Double = Dist.Skewness
```

Beta Distribution

Class **BetaDistribution** represents the beta probability distribution. The beta distribution is a family of curves with two free parameters, usually labelled α and β. Beta distributions are nonzero only on the interval `(0 1)`.

The distribution function for the beta distribution is:

$$f(x|\alpha, \beta) = \frac{x^{\alpha - 1}(1 - x)^{\beta - 1}}{B(\alpha,\beta)}$$

where $B(x,y)$ is the beta function. The beta CDF is the same as the incomplete beta function.

For example, this code constructs a **BetaDistribution**:

Code Example – C# beta distribution

```
double alpha = 3;
double beta = 7;
var dist = new BetaDistribution( alpha, beta );
```

Code Example – VB beta distribution

```
Dim Alpha As Double = 3
```

```
Dim Beta As Double = 7
Dim Dist As New BetaDistribution(Alpha, Beta)
```

The default constructor creates a **BetaDistribution** with α and β equal to `1`:

Code Example – C# beta distribution

```
var dist = new BetaDistribution();
```

Code Example – VB beta distribution

```
Dim Dist As New BetaDistribution()
```

The provided `Alpha` and `Beta` properties can be used to get and set the shape parameters after construction:

Code Example – C# beta distribution

```
dist.Alpha = 4;
dist.Beta = 10;
```

Code Example – VB beta distribution

```
Dist.Alpha = 4
Dist.Beta = 10
```

Once you have constructed a **BetaDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Binomial Distribution

Class **BinomialDistribution** represents the discrete probability distribution of obtaining exactly *n* successes in *N* trials where the probability of success on each trial is *p*. For example, this code constructs an **BinomialDistribution**:

Code Example – C# binomial distribution

```
int n = 20;
double p = 0.25;
var bin = new BinomialDistribution( n, p );
```

Code Example – VB binomial distribution

```
Dim N As Integer = 20
Dim P As Double = 0.25
Dim Bin As New BinomialDistribution(N, P)
```

The default constructor creates an **BinomialDistribution** with $n = 2$ and $p = 0.5$:

Code Example – C# binomial distribution

```
var bin = new BinomialDistribution();
```

Code Example – VB binomial distribution

```
Dim Bin As New BinomialDistribution()
```

The provided `N` and `P` properties can be used to get and set the number of trials and the probability of success on each trial after construction:

Code Example – C# binomial distribution

```
bin.N = 75;
bin.P = 0.02;
```

Code Example – VB binomial distribution

```
Bin.N = 75
Bin.P = 0.02
```

Once you have constructed an **BinomialDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Chi-Square Distribution

Class **ChiSquareDistribution** represents the chi-square (χ^2) probability distribution. The chi-square distribution is a special case of the gamma distribution with $\alpha = df/2$ and $\beta = 2$, where *df* is the degrees of freedom.

For example, this code constructs a **ChiSquareDistribution**:

Code Example – C# chi-square distribution

```
double df = 16;
var chiSq = new ChiSquareDistribution( df );
```

Code Example – VB chi-square distribution

```
Dim DF As Double = 16
Dim ChiSq As New ChiSquareDistribution(DF)
```

The default constructor creates a **ChiSquareDistribution** with `1` degree of freedom:

Code Example – C# chi-square distribution

```
var chiSq = new ChiSquareDistribution();
```

Code Example – VB chi-square distribution

```
Dim ChiSq As New ChiSquareDistribution()
```

The provided `DegreesOfFreedom` property can be used to get and set the degrees of freedom of the distribution after construction:

Code Example – C# chi-square distribution

```
chiSq.DegreesOfFreedom = 10;
```

Code Example – VB chi-square distribution

```
ChiSq.DegreesOfFreedom = 10
```

Once you have constructed a **ChiSquareDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Exponential Distribution

Class **ExponentialDistribution** represents the exponential distribution. A random variable *w* is said to have an exponential distribution if it has a probability density function

$$g(w) = \lambda e^{-\lambda w}$$

where $\lambda > 0$ is often called the *rate* parameter. The mean of an exponential distribution is $1/\lambda$, and the variance is $1/\lambda^2$. For example, this code constructs an **ExponentialDistribution**:

Code Example – C# exponential distribution

```
double lambda = 22;
var exp = new ExponentialDistribution( lambda );
```

Code Example – VB exponential distribution

```
Dim Lambda As Double = 22
Dim Exp As New ExponentialDistribution(Lambda)
```

The provided `Lambda` property can be used to get and set the rate after construction:

Code Example – C# exponential distribution

```
exp.Lambda = 15;
```

Code Example – VB exponential distribution

```
Exp.Lambda = 15
```

Once you have constructed an **ExponentialDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

F Distribution

Class **FDistribution** represents the *F* probability distribution. The *F* distribution is the ratio of two chi-square distributions with degrees of freedom `df1` and `df2`, respectively, where each chi-square has first been divided by its degrees of freedom. For example, this code constructs an **FDistribution**:

Code Example – C# F distribution

```
double df1 = 11;
double df2 = 19;
var f = new FDistribution( df1, df2 );
```

Code Example – VB F distribution

```
Dim DF1 As Double = 11
Dim DF2 As Double = 19
Dim F As New FDistribution(DF1, DF2)
```

The default constructor creates an **FDistribution** with both degrees of freedom equal to `1`:

Code Example – C# F distribution

```
var f = new FDistribution();
```

Code Example – VB F distribution

```
Dim F As New FDistribution()
```

The provided `DegreesOfFreedom1` and `DegreesOfFreedom2` properties can be used to get and set the degrees of freedom after construction:

Code Example – C# F distribution

```
f.DegreesOfFreedom1 = 15;
f.DegreesOfFreedom2 = 23;
```

Code Example – VB F distribution

```
F.DegreesOfFreedom1 = 15
F.DegreesOfFreedom2 = 23
```

Once you have constructed an **FDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Gamma Distribution

Class **GammaDistribution** represents the gamma probability distribution. The gamma distribution is a family of curves with two free parameters, usually labelled α and β. The mean of the distribution is $\alpha\beta$; the variance is $\alpha\beta^2$. When α is large, the gamma distribution closely approximates a normal distribution.

The distribution function for the gamma distribution is:

$$f(x|\alpha, \beta) = \frac{x^{\alpha - 1} e^{\frac{-x}{\beta}}}{\beta^{\alpha}\Gamma(\alpha)}$$

where $\Gamma(x)$ is the Gamma function.

For example, this code constructs a **GammaDistribution**:

Code Example – C# gamma distribution

```
double alpha = 7;
double beta = 12;
var gamma = new GammaDistribution( alpha, beta );
```

Code Example – VB gamma distribution

```
Dim Alpha As Double = 7
Dim Beta As Double = 12
Dim Gamma As New GammaDistribution(Alpha, Beta)
```

The default constructor creates a **GammaDistribution** with α and β equal to `1`:

Code Example – C# gamma distribution

```
var gamma = new GammaDistribution();
```

Code Example – VB gamma distribution

```
Dim Gamma As New GammaDistribution()
```

The provided `Alpha` and `Beta` properties can be used to get and set the shape parameters after construction:

Code Example – C# gamma distribution

```
gamma.Alpha = 10;
gamma.Beta = 15;
```

Code Example – VB gamma distribution

```
Gamma.Alpha = 10
Gamma.Beta = 15
```

Once you have constructed a **GammaDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Geometric Distribution

Class **GeometricDistribution** represents the geometric distribution. The geometric distribution is the probability distribution of the number of failures before the first success. It is supported on the set $\{0, 1, 2, 3, \ldots\}$.

A **GeometricDistribution** is constructed from a given probability of success *p*, where $0 < p \leq 1$. For example:

Code Example – C# geometric distribution

```
double p = .25;
var geo = new GeometricDistribution( p );
```

Code Example – VB geometric distribution

```
Dim P As Double = 0.25
Dim Geo As New GeometricDistribution(P)
```

Class **GeometricDistribution** provides property `P` that gets and sets the probability for success for the distribution.

Code Example – C# geometric distribution

```
geo.P = .5;
```

Code Example – VB geometric distribution

```
Geo.P = 0.5
```

Once you have constructed a **GeometricDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Johnson Distribution

Class **JohnsonDistribution** represents the Johnson system of distributions. The Johnson system is based on three possible transformations of a normal random variable—exponential, logistic, and hyperbolic sine—plus the identity transformation:

$$z = \gamma + \delta \ln(f(u)) \text{ where } u = \left(\frac{x-\xi}{\lambda}\right)$$

where the transformation *f()* has four possible forms based on the distribution type:

- Normal (SN): f(u) = exp(u)
- Log Normal (SL): f(u) = u
- Unbounded (SU):f(u) = u + sqrt(1+u^2)
- Bounded (SB):f(u) = u/(1-u)

A **JohnsonDistribution** instance is constructed from a set of distribution parameter values, and a **JohnsonTransformationType** enumerated value specifying the transformation type. For instance:

Code Example – C# Johnson distribution

```
double gamma = -0.18;
double delta = 2.55;
double xi = -0.14;
double lambda = 2.35;
JohnsonTransformationType type = JohnsonTransformationType.Normal;

var dist =
  new JohnsonDistribution( gamma, delta, xi, lambda, type );
```

Code Example – VB Johnson distribution

```
Dim Gamma As Double = -0.18
Dim Delta As Double = 2.55
Dim Xi As Double = -0.14
Dim Lambda As Double = 2.35
Dim Type As JohnsonTransformationType =
  JohnsonTransformationType.Normal

Dim Dist As New JohnsonDistribution(Gamma, Delta, Xi, Lambda, Type)
```

Once you have constructed a **JohnsonDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Class **JohnsonDistribution** also provides a static `Fit()` method for fitting a Johnson distribution to a data set. Estimation of the Johnson parameters is done from quantiles that correspond to the cumulative probabilities `[0.05, 0.206, 0.5, 0.794, 0.95]` using the method of Wheeler (1980).[1] For example:

Code Example – C# Johnson distribution

```
var data = new DoubleVector(-0.09736927, 0.21615254,
  0.88246516, 0.20559750, -0.61643584, -0.73479925, -0.13180279,
  0.31001699, -1.03968035, -0.18430887, 0.96726726, -0.10828009, -
  0.69842067, -0.27594517, 1.11464855, 0.55004396, 1.23667580,
  0.13909786, 0.41027510, -0.55845691);
```

[1]Wheeler, R.E. (1980). Quantile estimators of Johnson curve parameters. Biometrika. 67-3 725-728.

```
var dist = JohnsonDistribution.Fit(data);
```

Code Example – VB Johnson distribution

```
Dim Data As New DoubleVector(-0.09736927, 0.21615254,
  0.88246516, 0.2055975, -0.61643584, -0.73479925, -0.13180279,
  0.31001699, -1.03968035, -0.18430887, 0.96726726, -0.10828009,
  -0.69842067, -0.27594517, 1.11464855, 0.55004396, 1.2366758,
  0.13909786, 0.4102751, -0.55845691)
Dim Dist As JohnsonDistribution = JohnsonDistribution.Fit(Data)
```

The `Transform()` method transforms data using a **JohnsonDistribution** object.

Logistic Distribution

Class **LogisticDistribution** represents the logistic probability distribution with a specified location (mean) and scale. The logistic distribution with location *m* and scale *b* has distribution function:

$$f(x) = \frac{1}{1 + e^{-(x-m)/b}}$$

and density:

$$f(x) = \frac{e^{-(x-m)/b}}{b[1 + e^{-(x-m)/b}]^2}$$

For example, this code constructs a **LogisticDistribution**:

Code Example – C# logistic distribution

```
double loc = 2.0;
double scale = 1.5;
var logistic = new LogisticDistribution( loc, scale );
```

Code Example – VB logistic distribution

```
Dim Loc As Double = 2.0
Dim Scale As Double = 1.5
Dim Logistic As New LogisticDistribution(Loc, Scale)
```

The provided `Location` and `Scale` properties can be used to get and set distribution parameters after construction:

Code Example – C# logistic distribution

```
logistic.Location = 7.123;
logistic.Scale = 4.5;
```

Code Example – VB logistic distribution

```
Logistic.Location = 7.123
Logistic.Scale = 4.5
```

Once you have constructed a **LogisticDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Log-Normal Distribution

Class **LognormalDistribution** represents the log-normal distribution. A log-normal distribution has a normal distribution as its logarithm:

$$f(x) = e^{\mathrm{normal}(\mu, \sigma)}$$

For example, this code constructs an **LognormalDistribution** whose associated normal distribution has the specified mean and standard deviation:

Code Example – C# log-normal distribution

```
double mu = -99;
double sigma = 6;
var ln = new LognormalDistribution( mu, sigma );
```

Code Example – VB log-normal distribution

```
Dim Mu As Double = -99
Dim Sigma As Double = 6
Dim LN As New LognormalDistribution(Mu, Sigma)
```

The default constructor creates a **LognormalDistribution** whose associated normal distribution has mean `0` and standard deviation `1`:

Code Example – C# log-normal distribution

```
var ln = new LognormalDistribution();
```

Code Example – VB log-normal distribution

```
Dim LN As New LognormalDistribution()
```

The `Mu` and `Sigma` properties can be used to get and set the mean and standard deviation after construction:

Code Example – C# log-normal distribution

```
ln.Mu = 2.25;
ln.Sigma = .75;
```

Code Example – VB log-normal distribution

```
LN.Mu = 2.25
```

```
LN.Sigma = 0.75
```

Once you have constructed a **LognormalDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Negative Binomial Distribution

Class **NegativeBinomialDistribution** represents the discrete probability distribution of obtaining *N* successes in a series of *x* trials, where the probability of success on each trial is *P*.

For example, this code constructs an **NegativeBinomialDistribution**:

Code Example – C# negative binomial distribution

```
int n = 5;
double p = 0.25;
var negBin = new NegativeBinomialDistribution( n, p );
```

Code Example – VB negative binomial distribution

```
Dim N As Integer = 5
Dim P As Double = 0.25
Dim NegBin As New NegativeBinomialDistribution(N, P)
```

The default constructor creates an **NegativeBinomialDistribution** with $n = 2$ and $p = 0.5$:

Code Example – C# negative binomial distribution

```
var negBin = new NegativeBinomialDistribution();
```

Code Example – VB negative binomial distribution

```
Dim NegBin As New NegativeBinomialDistribution()
```

The provided `N` and `P` properties can be used to get and set the number of successes and the probability of success on each trial after construction:

Code Example – C# negative binomial distribution

```
negBin.N = 75;
negBin.P = 0.02;
```

Code Example – VB negative binomial distribution

```
Bin.N = 75
Bin.P = 0.02
```

Once you have constructed an **NegativeBinomialDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Normal Distribution

Class **NormalDistribution** represents the normal (Gaussian) probability distribution. with a specified mean and variance. For example, this code creates a normal distribution with a mean of `1` and variance of `2.5`:

Code Example – C# normal distribution

```
var norm = new NormalDistribution( 1, 2.5 );
```

Code Example – VB normal distribution

```
Dim Norm As New NormalDistribution(1, 2.5)
```

The default constructor creates a **NormalDistribution** with mean `0` and variance `1`:

Code Example – C# normal distribution

```
var norm = new NormalDistribution();
```

Code Example – VB normal distribution

```
Dim Norm As New NormalDistribution()
```

The `Mean` and `Variance` properties inherited from **IRandomVariableMoments** can be used to get and set the mean and variance after construction:

Code Example – C# normal distribution

```
norm.Mean = 2.25;
norm.Variance = .75;
```

Code Example – VB normal distribution

```
Norm.Mean = 2.25
Norm.Variance = 0.75
```

Once you have constructed a **NormalDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Poisson Distribution

Class **PoissonDistribution** represents a poisson distribution with a specified λ parameter, which is both the mean and the variance of the distribution. The poisson distribution is the probability of obtaining exactly n successes in N trials. It is often used as a model for the number of events in a specific time period. Poisson (1837) showed that the Poisson distribution is the limiting case of a binomial distribution where N approaches infinity and p goes to zero while $Np = \lambda$. The distribution function for the Poisson distribution is:

$$f(x|\lambda) = \frac{e^{-\lambda}\lambda^{x}}{x!}$$

For example, this code constructs a **PoissonDistribution**:

Code Example – C# poisson distribution

```
double lambda = 150;
var poisson = new PoissonDistribution( lambda );
```

Code Example – VB poisson distribution

```
Dim Lambda As Double = 150
Dim Poisson As New PoissonDistribution(Lambda)
```

The `Mean` and `Variance` properties inherited from **IRandomVariableMoments** can also be used to get and set λ after construction:

Code Example – C# poisson distribution

```
poisson.Mean = 3;
```

Code Example – VB poisson distribution

```
Poisson.Mean = 3
```

Once you have constructed a **PoissonDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Student's t Distribution

Class **TDistribution** represents Student's *t* distribution with specified degrees of freedom. As the number of degrees of freedom grows, the *t* distribution approaches the normal distribution with mean `0` and variance `1`.

For example, this code constructs a **TDistribution**:

Code Example – C# t distribution

```
double df = 53;
var t = new TDistribution( df );
```

Code Example – VB t distribution

```
Dim DF As Double = 53
Dim T As New TDistribution(DF)
```

The default constructor creates a **TDistribution** with `1` degree of freedom:

Code Example – C# t distribution

```
var t = new TDistribution();
```

Code Example – VB t distribution

```
Dim T As New TDistribution()
```

The provided `DegreesOfFreedom` property can be used to get and set the degrees of freedom of the distribution after construction:

Code Example – C# t distribution

```
t.DegreesOfFreedom = 54;
```

Code Example – VB t distribution

```
T.DegreesOfFreedom = 54
```

Once you have constructed a **TDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Triangular Distribution

Class **TriangularDistribution** represents the triangular distribution. The triangular distribution is defined by three parameters, a lower limit *a*, an upper limit *b*, and number *c*, between *a* and *b*, called the *mode*. The probability density function has the shape of a triangle in the X/Y plane with vertices *(a, 0)*, *(b, 0)*, and *(c, y)*, where *y* is chosen so that the area of the triangle is `1`.

For example, this code constructs an **TriangularDistribution** with the given parameters:

Code Example – C# triangular distribution

```
double lower = 3;
double upper = 10;
double mode = 8;
var td = new TriangularDistribution( lower, upper, mode );
```

Code Example – VB triangular distribution

```
Dim Lower As Double = 3
Dim Upper As Double = 10
Dim Mode As Double = 8
Dim TD As New TriangularDistribution(Lower, Upper, Mode)
```

If you don't specify the mode, the midpoint of the lower and upper limits is used.

The default constructor creates a **TriangularDistribution** with lower limit `0`, upper limit `1`, and mode `0.5`:

Code Example – C# triangular distribution

```
var td = new TriangularDistribution();
```

Code Example – VB triangular distribution

```
Dim TD As New TriangularDistribution()
```

The `LowerLimit`, `UpperLimit`, and `Mode` properties can be used to get and set the distribution parameters after construction:

Code Example – C# triangular distribution

```
td.LowerLimit = 1.5;
td.UpperLimit = 3.5;
td.Mode = 2.75;
```

Code Example – VB triangular distribution

```
TD.LowerLimit = 1.5
TD.UpperLimit = 3.5
TD.Mode = 2.75
```

Once you have constructed a **TriangularDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Uniform Distribution

Class **UniformDistribution** represents the uniform distribution. For example, this code constructs an **UniformDistribution** with the specified lower and upper limits:

Code Example – C# uniform distribution

```
double lower = -.77;
double upper = 1.22;
var uni = new UniformDistribution( lower, upper );
```

Code Example – VB uniform distribution

```
Dim Lower As Double = -0.77
Dim Upper As Double = 1.22
Dim Uni As New UniformDistribution(Lower, Upper)
```

The default constructor creates a **UniformDistribution** with lower limit `0` and upper limit `1`:

Code Example – C# uniform distribution

```
var uni = new UniformDistribution();
```

Code Example – VB uniform distribution

```
Dim Uni As New UniformDistribution()
```

The `LowerLimit` and `UpperLimit` properties can be used to get and set the lower and upper limits after construction:

Code Example – C# uniform distribution

```
uni.LowerLimit = 0;
uni.UpperLimit = 2.0;
```

Code Example – VB uniform distribution

```
Uni.LowerLimit = 0
Uni.UpperLimit = 2.0
```

Once you have constructed a **UniformDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

Weibull Distribution

Class **WeibullDistribution** represents the Weibull distribution. The probability density function of the Weibull distribution is given by:

$$f(x|k, \lambda) = \frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k}$$

where $k > 0$ is the *shape* parameter and $\lambda > 0$ is the *scale* parameter of the distribution.

For example, this code constructs an **WeibullDistribution** with the specified distribution parameters:

Code Example – C# Weibull distribution

```
double scale = 1.5;
double shape = 3;
var wb = new WeibullDistribution( scale, shape );
```

Code Example – VB Weibull distribution

```
Dim Scale As Double = 1.5
Dim Shape As Double = 3
Dim WB As New WeibullDistribution(Scale, Shape)
```

The `Scale` and `Shape` properties can be used to get and set the distribution parameters after construction:

Code Example – C# Weibull distribution

```
wb.Scale = .5;
wb.Shape = 2;
```

Code Example – VB Weibull distribution

```
WB.Scale = 0.5
WB.Shape = 2
```

Once you have constructed a **WeibullDistribution** object, you can query it for the PDF, CDF, inverse CDF, and random variable moments, as described in Section 40.1.

40.2 Correlated Random Inputs

NMath Stats provides classes **InputVariableCorrelator** and **ReducedVarianceInputCorrelator** to induce a desired rank correlation among a set of random input variables. The correlated inputs retain the same marginal distributions as the original inputs but have a Spearman's rank correlation matrix approximately equal to that specified by the user. The method used is that of Iman and Conover (1982).[2]

ReducedVarianceInputCorrelator performs the same function as **InputVariableCorrelator** class, but uses an algorithm that produces more accurate results, at some cost in performance.

Constructing Correlator Instances

Instances of **InputVariableCorrelator** and **ReducedVarianceInputCorrelator** are constructed from the number of samples and the desired correlation matrix. This code assume `500` samples of `6` input variables:

Code Example – C# correlated random inputs

```
int numSamples = 500;
string str = "6x6 [1 0 0 0 0 0 " +
                  "0 1 0 0 0 0 " +
                  "0 0 1 0 0 0 " +
                  "0 0 0 1 .75 -.70 " +
                  "0 0 0 .75 1 -.95 " +
                  "0 0 0 -.7 -.95 1]";
var desiredCorrelations = new DoubleMatrix( str );

var correlator = new
  InputVariableCorrelator( numSamples, desiredCorrelations );
```

Code Example – VB correlated random inputs

```
Dim NumSamples As Integer = 500
Dim Str As String = "6x6 [1 0 0 0 0 0 " &
                         "0 1 0 0 0 0 " &
```

[2]Iman, Ronald L. and W. J. Conover, "A Distribution-Free Approach to Inducing Rank Correlation Amoung Input Variables", Commun. Statist.-Simula. Computation 11(3), pp. 311-334 (1982)

```
                         "0 0 1 0 0 0 " &
                         "0 0 0 1 .75 -.70 " &
                         "0 0 0 .75 1 -.95 " &
                         "0 0 0 -.7 -.95 1]"
Dim DesiredCorrelations As New DoubleMatrix(Str)

Dim Correlator As New
  InputVariableCorrelator(NumSamples, DesiredCorrelations)
```

Most of the work done by the correlation algorithm involves setting up a *score* matrix which has been transformed so that it's Spearman's rank correlation matrix is equal to the desired correlation matrix. The computation of this score matrix requires only the number of samples and the desired correlation matrix, and is performed at construction time. Once you have constructed an **InputVariableCorrelator** or **ReducedVarianceInputCorrelator** instance, you can correlate batches of random inputs relatively quickly.

Correlating Random Inputs

The `GetCorrelatedInputs()` method on **InputVariableCorrelator** and **ReducedVarianceInputCorrelator** returns a matrix containing a given set of input variables values re-ordered so as to have the desired correlations.

For instance, this code creates a set of samples drawn from `4` different distributions (each row of the `inputs` matrix is a random sample of the `6` input variables), and induces the desired correlation:

Code Example – C# correlated random inputs

```
var betaRng = new RandGenBeta();
var uniformRng = new RandGenUniform();
var poissonRng = new RandGenPoisson();
var normalRng = new RandGenNormal();

var inputs = new DoubleMatrix( numSamples, 6 );
betaRng.Fill( inputs.Col( 0 ).DataBlock.Data );
uniformRng.Fill( inputs.Col( 1 ).DataBlock.Data );
poissonRng.Fill( inputs.Col( 2 ).DataBlock.Data );
normalRng.Fill( inputs.Col( 3 ).DataBlock.Data );
betaRng.Fill( inputs.Col( 4 ).DataBlock.Data );
uniformRng.Fill( inputs.Col( 5 ).DataBlock.Data );

DoubleMatrix correlatedInputs =
  correlator.GetCorrelatedInputs( inputs );
```

Code Example – VB correlated random inputs

```
Dim BetaRng As New RandGenBeta()
Dim UniformRng As New RandGenUniform()
Dim PoissonRng As New RandGenPoisson()
```

```
Dim NormalRng As New RandGenNormal()

Dim Inputs As New DoubleMatrix(NumSamples, 6)
BetaRng.Fill(Inputs.Col(0).DataBlock.Data)
UniformRng.Fill(Inputs.Col(1).DataBlock.Data)
PoissonRng.Fill(Inputs.Col(2).DataBlock.Data)
NormalRng.Fill(Inputs.Col(3).DataBlock.Data)
BetaRng.Fill(Inputs.Col(4).DataBlock.Data)
UniformRng.Fill(Inputs.Col(5).DataBlock.Data)

Dim CorrelatedInputs As DoubleMatrix =
  Correlator.GetCorrelatedInputs(Inputs)
```

You can compare the actual Spearman's rank correlation matrix with the desired correlation matrix, like so:

Code Example – C# correlated random inputs

```
DoubleMatrix actualCorrelations =
  StatsFunctions.Spearmans( correlatedInputs );

Console.WriteLine( "Desired: " + desiredCorrelations );
Console.WriteLine( "Actual: " + actualCorrelations );
```

Code Example – VB correlated random inputs

```
Dim ActualCorrelations As DoubleMatrix =
  StatsFunctions.Spearmans(CorrelatedInputs)

Console.WriteLine("Desired: " & DesiredCorrelations)
Console.WriteLine("Actual: " & ActualCorrelations)
```

Correlator Properties

InputVariableCorrelator and **ReducedVarianceInputCorrelator** provide the following read-only properties:

- `Rstar` gets the permuted score matrix which has been transformed to have the desired correlation matrix.
- `NumInputVariables` gets the number of input variables.
- `SampleSize` gets the sample size of the input variables.

Convenience Method

The static `CorrelatedRandomInputs()` convenience method is provided on class **StatsFunctions** for cases where you need only one set of correlated inputs. For example:

Code Example – C# correlated random inputs

```
DoubleMatrix correlatedInputs =
  StatsFunctions.CorrelatedRandomInputs( inputs,
    desiredCorrelations );
```

Code Example – VB correlated random inputs

```
Dim CorrelatedInputs As DoubleMatrix =
  StatsFunctions.CorrelatedRandomInputs(Inputs,
  DesiredCorrelations)
```

In the special case of two input variables, an additional overload obviates the need for setting up the original input sample matrix. For instance, this code creates two sequences of `100` normally distributed random numbers which have, approximately, the specified rank correlation coefficient `0.8`:

Code Example – C# correlated random inputs

```
double mean1 = 43.2;
double var1 = 1.2;
var normalRng1 = new RandGenNormal( mean1, var1 );

double mean2 = 102.45;
double var2 = 8.098;
var normalRng2 = new RandGenNormal( mean2, var2 );

double desiredRankCorrelation = .8;

int numSamples = 100;

DoubleMatrix correlatedInputs =
  StatsFunctions.CorrelatedRandomInputs( numSamples,
    desiredRankCorrelation, normalRng1, normalRng2 );
```

Code Example – VB correlated random inputs

```
Dim Mean1 As Double = 43.2
Dim Var1 As Double = 1.2
Dim NormalRng1 As New RandGenNormal(Mean1, Var1)

Dim Mean2 As Double = 102.45
Dim Var2 As Double = 8.098
Dim NormalRng2 As New RandGenNormal(Mean2, Var2)

Dim DesiredRankCorrelation As Double = 0.8

Dim NumSamples As Integer = 100

Dim CorrelatedInputs As DoubleMatrix =
  StatsFunctions.CorrelatedRandomInputs(NumSamples,
  DesiredRankCorrelation, NormalRng1, NormalRng2)
```

40.3 Box-Cox Power Transformations

Box-Cox power transformations compute a rank-preserving transformation of data to stabilize variance and make the data more normal. The power transformation is defined as a continuously varying function, with respect to the power parameter λ,

$$y(\lambda) = \frac{y^{\lambda} - 1}{\lambda}$$

In **NMath Stats**, class **BoxCox** compute the Box-Cox power tranformations for a set of data points and parameter value λ. In addition, methods are provided for computing the corresponding log-likelihood function and the value of λ which maximizes it.

For example:

Code Example – C# Box-Cox transformations

```
var data = new DoubleVector( "[.15 .09 .18 .10 .05 .12 .08 .05 .08
.10 .07 .02 .01 .10 .10 .10 .02 .10 .01 .40 .10 .05 .03 .05 .15 .10
.15 .09 .08 .18 .10 .20 .11 .30 .02 .20 .20 .30 .30 .40 .30 .05]"
);

var interval = new Interval( -5, 5, Interval.Type.Closed );

var bc = new BoxCox( data, interval );

Console.WriteLine( bc.Lambda );
Console.WriteLine( bc.TransformedData );
```

Code Example – VB Box-Cox transformations

```
Dim Data As New DoubleVector("[.15 .09 .18 .10 .05 .12 .08 .05 .08
.10 .07 .02 .01 .10 .10 .10 .02 .10 .01 .40 .10 .05 .03 .05 .15 .10
.15 .09 .08 .18 .10 .20 .11 .30 .02 .20 .20 .30 .30 .40 .30 .05]"
)_

Dim Interval As New Interval(-5, 5, Interval.Type.Closed)

Dim BC As New BoxCox(Data, Interval)

Console.WriteLine(BC.Lambda)
Console.WriteLine(BC.TransformedData)
```

BoxCox searches from `-5` to `5` until the best value of λ is found (the value which maximizes the log-likelihood function).

CHAPTER 41.
HYPOTHESIS TESTS

Hypothesis tests use statistics to determine the probability that a given hypothesis is true. For example, could the differences between two sample means be explained away as sampling error? **NMath Stats** provides classes for many common hypothesis tests.

This chapter describes the hypothesis test classes. For non-parametric tests, see Chapter 45.

41.1 Common Interface

All hypothesis test classes share substantially the same interface. Once you learn how to use one test, it's easy to use any of the others.

Static Properties

All hypothesis test classes have static `DefaultAlpha` properties that get and set the default alpha level associated with tests of that type. The default value is `0.01`. For instance:

Code Example – C# hypothesis tests

```
var test1 = new OneSampleTTest();
// test1.Alpha == 0.01
OneSampleTTest.DefaultAlpha = 0.05;
var test2 = new OneSampleTTest();
// test2.Alpha == 0.05
```

Code Example – VB hypothesis tests

```
Dim Test1 As New OneSampleTTest()
'' test1.Alpha == 0.01
OneSampleTTest.DefaultAlpha = 0.05
Dim Test2 As New OneSampleTTest()
'' test2.Alpha == 0.05
```

Similarly, all hypothesis test classes have static `DefaultType` properties that get and set the default form of the alternative hypothesis. The form is specified using the **HypothesisType** enumeration, with the following enumerated values:

- `Left` indicates a one-sided form to the left, $\mu < \mu_0$.

- `Right` indicates a one-sided form to the right, $\mu > \mu_0$.

- `TwoSided` indicates a two-sided form, $\mu \neq \mu_0$.

The default value for all test classes is `HypothesisType.TwoSided`. For example:

Code Example – C# hypothesis tests

```
var test1 = new OneSampleTTest();
// test1.Type == HypothesisType.TwoSided
OneSampleTTest.DefaultType = HypothesisType.Left;
var test2 = new OneSampleTTest();
// test2.Type == HypothesisType.Left
```

Code Example – VB hypothesis tests

```
Dim Test1 As New OneSampleTTest()
'' test1.Type == HypothesisType.TwoSided
OneSampleTTest.DefaultType = HypothesisType.Left
Dim Test2 As New OneSampleTTest()
'' test2.Type == HypothesisType.Left
```

Creating Hypothesis Test Objects

All hypothesis test classes provide two paths for constructing instances of that type:

- A *parameter-based* method, in which all necessary sample and population parameters are explicitly specified.

- A *data-based* method, in which sample parameters are computed from supplied sample data.

NOTE—In the data-based method, once sample parameters have been computed from the given data, the data is discarded, and cannot be recovered from the test object.

For example, a one-sample z-test compares a single sample mean to an expected mean from a normal distribution with known standard deviation. This code constructs a **OneSampleZTest** object by explicitly specifying a sample mean, sample size, population mean, and population standard deviation:

Code Example – C# hypothesis tests

```
double xbar = 112.8;
int n = 9;
double mu0 = 100;
double sigma = 15;
var test = new OneSampleZTest( xbar, n, mu0, sigma );
```

Code Example – VB hypothesis tests

```
Dim XBar As Double = 112.8
```

```
Dim N As Integer = 9
Dim Mu0 As Double = 100
Dim Sigma As Double = 15
Dim Test As New OneSampleZTest(XBar, N, Mu0, Sigma)
```

This code constructs a **OneSampleZTest** object by supplying a vector of sample data, and the necessary population parameters:

Code Example – C# hypothesis tests

```
var data =
  new DoubleVector( "[ 116 110 111 113 112 113 111 109 121 ]" );
double mu0 = 100;
double sigma = 15;
var test = new OneSampleZTest( data, mu0, sigma );
```

Code Example – VB hypothesis tests

```
Dim MyData As New DoubleVector("[ 116 110 111 113 112 113 111 109
121 ]")
Dim Mu0 As Double = 100
Dim Sigma As Double = 15
Dim Test As New OneSampleZTest(MyData, Mu0, Sigma)
```

In this case, the sample mean and sample size are calculated from the given data. The data-based method supports sample data in vectors, arrays, and data frame columns.

In both the parameter-based method and the data-based method, the alpha level for the hypothesis test is set to the current value specified by the static `DefaultAlpha` property, and the form of the hypothesis test is set to the current `DefaultType`, as described above.

Constructors are also provided for all test classes that enable you to set the alpha level and hypothesis type to non-default values. For example:

Code Example – C# hypothesis tests

```
var test = new OneSampleZTest( data, mu0, sigma, 0.05,
  HypothesisType.Left );
```

Code Example – VB hypothesis tests

```
Dim Test As New OneSampleZTest(MyData, Mu0, Sigma, 0.05,
  HypothesisType.Left)
```

Properties of Hypothesis Test Objects

All hypothesis test classes provide the following read-only properties:

- `Distribution` gets the distribution of the test statistic associated with the hypothesis test.

- `Statistic` gets the value of the test statistic associated with this hypothesis test.
- `P` gets the *p*-value associated with the test statistic.
- `Reject` tests whether the null hypothesis can be rejected, using the current hypothesis type and alpha level.
- `LeftCriticalValue` gets the one-sided to the left critical value based on the current probability distribution and alpha level.
- `RightCriticalValue` gets the one-sided to the right critical value based on the current probability distribution and alpha level.
- `LeftProbability` gets the area under the probability distribution to the left of the test statistic.
- `RightProbability` gets the area under the probability distribution to the right of the test statistic.
- `LowerConfidenceLimit` gets the $1 - \alpha$ lower confidence limit for the true mean.
- `UpperConfidenceLimit` gets the $1 - \alpha$ upper confidence limit for the true mean.
- `SEM` gets the standard error of the mean.

The following read-write properties are also provided:

- `Alpha` gets and sets the alpha level associated with the hypothesis test.
- `Type` gets and sets the form of the alternative hypothesis associated with the hypothesis test.

Additionally, each hypothesis test provides properties for accessing the specific sample and population parameters that define the test. For example, a **OneSampleZTest** has additional properties for accessing the sample mean, `Xbar`, the sample size, `N`, the population mean, `Mu0`, and the population standard deviation, `Sigma`.

Modifying Hypothesis Test Objects

All hypothesis test classes provide `Update()` methods for modifying a test with new sample parameters or sample data, and new population parameters. For example, if `test` is a **TwoSampleFTest** instance, this code updates the test with two new samples, taken from two columns in a data frame `df`:

Code Example – C# hypothesis tests

```
test.Update( df[3], df[7]  );
```

Code Example – VB hypothesis tests

```
Test.Update(DF(3), DF(7))
```

Printing Results

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

Code Example – C# hypothesis tests

```
var data1 = new DoubleVector( "9.21 11.51 12.79 11.85 9.97
  8.79 9.69 9.68 9.19" );
var data2 = new DoubleVector( "7.53 7.48 8.08 8.09 10.15
  8.40 10.88 6.13 7.90 7.05 7.48 7.58 8.11" );
var test = new TwoSampleFTest( data1, data2, 0.05,
  HypothesisType.TwoSided );
Console.WriteLine( test.ToString() );
```

Code Example – VB hypothesis tests

```
Dim MyData1 As New DoubleVector("9.21 11.51 12.79 11.85 9.97 8.79
9.69 9.68 9.19")
Dim MyData2 As New DoubleVector("7.53 7.48 8.08 8.09 10.15 8.40
10.88 6.13 7.90 7.05 7.48 7.58 8.11")
Dim Test As New TwoSampleFTest(MyData1, MyData2, 0.05,
HypothesisType.TwoSided)
Console.WriteLine(Test.ToString())
```

The output is:

```
Two Sample F Test
-----------------

Sample Sizes = 9 and 13
Standard Deviations = 1.39787139767736 and 1.23808008936914
Variances = 1.95404444444444 and 1.53284230769231
Ratio of Variances = 1.27478504125206
Computed F statistic: 1.27478504125206, num df = 8, denom df = 12

Hypothesis type: two-sided
Null hypothesis: true ratio of variances = 1
Alt hypothesis: true ratio of variances != 1
P-value: 0.679745985376403
RETAIN the null hypothesis for alpha = 0.05
0.95 confidence interval: 0.363002872041806 5.3536732579205
```

41.2 One Sample Z-Test

Class **OneSampleZTest** determines whether a sample from a normal distribution with known standard deviation could have a given mean. For example, suppose we wish to determine whether the IQs of children from a particular school are above average, given that Wechsler IQ scores are normally distributed with a mean of `100` and standard deviation of `15`. Sample scores from `9` students are `116 110 111 113 112 113 111 109 121`, with a mean of `112.8`.

As described Section 41.1, all hypothesis test classes provide two paths for constructing instances of that type: a parameter-based method and a data-based method. Thus, you can construct a **OneSampleZTest** object by explicitly specifying a sample mean ($\bar{x}$), sample size (n), population mean (μ_0), and population standard deviation (σ), like so:

Code Example – C# z-test

```
double xbar = 112.8;
int n = 9;
double mu0 = 100;
double sigma = 15;
var test = new OneSampleZTest( xbar, n, mu0, sigma );
```

Code Example – VB z-test

```
Dim XBar As Double = 112.8
Dim N As Integer = 9
Dim Mu0 As Double = 100
Dim Sigma As Double = 15
Dim Test As New OneSampleZTest(XBar, N, Mu0, Sigma)
```

Or by supplying a set of sample data, and the necessary population parameters:

Code Example – C# z-test

```
var data =
  new DoubleVector( "[ 116 110 111 113 112 113 111 109 121 ]" );
double mu0 = 100;
double sigma = 15;
var test = new OneSampleZTest( data, mu0, sigma );
```

Code Example – VB z-test

```
Dim MYData As New DoubleVector("[ 116 110 111 113 112 113 111 109
121 ]")
Dim Mu0 As Double = 100
Dim Sigma As Double = 15
Dim Test As New OneSampleZTest(MyData, Mu0, Sigma)
```

In this case, the sample mean and sample size are calculated from the given data.

In addition to the properties common to all hypothesis test objects (Section 41.1), a **OneSampleZTest** object provides the following read-only properties:

- `Xbar` gets the sample mean.
- `N` gets the sample size.
- `Mu0` gets the population mean.
- `Sigma` gets the population standard deviation.

By default, a **OneSampleZTest** object performs a two-sided hypothesis test ($H_1: \mu \neq \mu_0$) with $\alpha = 0.01$. In this example, we wish to test the one-sided form to the right ($H_1: \mu > \mu_0$; that is, we wish to test whether the children in our sample have a *higher* than average IQ. Suppose also that we wish to set the alpha level to `0.05`. Non-default test parameters can be specified at the time of construction using constructor overloads, or after construction using the provided `Alpha` and `Type` properties, like so:

Code Example – C# z-test

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

Code Example – VB z-test

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

Once you've constructed and configured a **OneSampleZTest** object, you can access the test results using the provided properties, as described in Section 41.1:

Code Example – C# z-test

```
Console.WriteLine( "z-statistic = " + test.Statistic );
Console.WriteLine( "p-value = " + test.P );
Console.WriteLine( "reject the null hypothesis? " + test.Reject);
```

Code Example – VB z-test

```
Console.WriteLine("z-statistic = " & Test.Statistic)
Console.WriteLine("p-value = " & Test.P)
Console.WriteLine("reject the null hypothesis? " & Test.Reject)
```

The output is:

```
z-statistic = 2.56
p-value = 0.00523360816355578
reject the null hypothesis? true
```

This indicates that we can reject the null hypotheses ($H_0: \mu = \mu_0$). We can conclude that the children have IQs significantly above average.

Finally, remember that the `ToString()` method returns a formatted string representation of the complete 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
```

41.3 One Sample T-Test

Class **OneSampleTTest** determines whether a sample from a normal distribution with unknown standard deviation could have a given mean. For example, suppose we wish to determine whether the self-esteem of children from a particular school differ from average, given a known population value of `3.9` on the Rosenberg Self-Esteem Scale. `113` children are tested, with a mean score of `4.0408` and a standard deviation of `.6542`.

As described Section 41.1, all hypothesis test classes provide two paths for constructing instances of that type: a parameter-based method and a data-based method. Thus, you can construct a **OneSampleTTest** object by explicitly specifying a sample mean ($\bar{x}$), sample standard deviation (s), sample size (n), and population mean (μ_0), like so:

Code Example – C# t-test

```
double xbar = 4.0408;
double s = .6542;
int n = 113;
double mu0 = 3.9;
var test = new OneSampleTTest( xbar, s, n, mu0 );
```

Code Example – VB t-test

```
Dim XBar As Double = 4.0408
Dim S As Double = 0.6542
Dim N As Integer = 113
Dim Mu0 As Double = 3.9
```

```
Dim Test As New OneSampleTTest(XBar, S, N, Mu0)
```

Or by supplying a set of sample data, and the necessary population parameters. For instance, if the sample data is in column `3` of **DataFrame** `df`:

Code Example – C# t-test

```
double mu0 = 3.9;
var test = new OneSampleTTest( df[3], mu0 );
```

Code Example – VB t-test

```
Dim Mu0 As Double = 3.9
Dim Test As New OneSampleTTest(DF(3), Mu0)
```

In this case, the sample mean, standard deviation, and size are calculated from the given data.

In addition to the properties common to all hypothesis test objects (Section 41.1), a **OneSampleTTest** object provides the following read-only properties:

- `Xbar` gets the sample mean.
- `S` gets the sample standard deviation.
- `N` gets the sample size.
- `Mu0` gets the population mean.
- `DegreesOfFreedom` gets the degrees of freedom.

By default, a **OneSampleTTest** 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 `Alpha` and `Type` properties, like so:

Code Example – C# t-test

```
test.Alpha = 0.05;
```

Code Example – VB t-test

```
Test.Alpha = 0.05
```

Once you've constructed and configured a **OneSampleTTest** object, you can access the various test results using the provided properties, as described in Section 41.1:

Code Example – C# t-test

```
Console.WriteLine( "t-statistic = " + test.Statistic );
Console.WriteLine( "deg of freedom = " + test.DegreesOfFreedom );
Console.WriteLine( "p-value = " + test.P );
Console.WriteLine( "reject the null hypothesis? " + test.Reject);
```

Code Example – VB t-test

```
Console.WriteLine("t-statistic = " & Test.Statistic)
Console.WriteLine("deg of freedom = " & Test.DegreesOfFreedom)
Console.WriteLine("p-value = " & Test.P)
Console.WriteLine("reject the null hypothesis? " & Test.Reject)
```

The output is:

```
t-statistic = 2.28786996397591
deg of freedom = 112
p-value = 0.0240223660991041
reject the null hypothesis? True
```

This indicates that we can reject the null hypotheses ($H_0 : \mu = \mu_0$). We can conclude that the children have self-esteem scores significantly different than average.

Finally, remember that the `ToString()` method returns a formatted string representation of the complete test results:

```
One Sample t Test
-----------------

Sample mean = 4.0408
Sample standard deviation = 0.6542
Sample size = 113
Population mean = 3.9
Computed t statistic: 2.28786996397591, df = 112

Hypothesis type: two-sided
Null hypothesis: sample mean = population mean
Alt hypothesis: sample mean != population mean
P-value: 0.0240223660991041
REJECT the null hypothesis for alpha = 0.05
0.95 confidence interval: 3.91886249658971 4.16273750341029
```

41.4 Two Sample Paired T-Test

Class **TwoSamplePairedTTest** tests the null hypothesis that the population mean of the *paired* differences of two samples is zero. Pairing involves matching up individuals in two samples so as to minimize their dissimilarity except in the factor under study. Paired samples often occur in pre-test/post-test studies in which subjects are measured before and after an intervention. They also occur in matched-pairs (for example, matching on age and sex), cross-over trials, and sequential observational samples. Paired samples are also called *matched* samples and *dependent* samples.

NOTE—TwoSamplePairedTTest is equivalent to performing a OneSampleTTest on the paired differences (see Section 41.3).

For example, suppose we measure the thickness of plaque (mm) in the carotid artery of `10` randomly selected patients with mild atherosclerotic disease. Two measurements are taken: before treatment with Vitamin E (baseline), and after two years of taking Vitamin E daily. The mean difference between paired measurements is `0.045` with a standard deviation of `0.0264`.

As described Section 41.1, all hypothesis test classes provide two paths for constructing instances of that type: a parameter-based method and a data-based method. Thus, you can construct a **TwoSamplePairedTTest** object by explicitly specifying the mean difference between paired observations ($\bar{x}$), the standard deviation of the differences (s), and the sample size (n), like so:

Code Example – C# paired t-test

```
double xbar = 0.045;
double s = 0.0264;
int n = 10;
var test = new TwoSamplePairedTTest( xbar, s, n );
```

Code Example – VB paired t-test

```
Dim XBar As Double = 0.045
Dim S As Double = 0.0264
Dim N As Integer = 10
Dim Test As New TwoSamplePairedTTest(XBar, S, N)
```

Alternatively, you can supply two sets of sample data. For instance, this code adds data to a **DataFrame** (Chapter 37):

Code Example – C# paired t-test

```
var df = new DataFrame();
df.AddColumn( new DFNumericColumn( "Baseline" ) );
df.AddColumn( new DFNumericColumn( "Vit E" ) );
df.AddRow( 1, 0.66, 0.60 );
df.AddRow( 2, 0.72, 0.65 );
df.AddRow( 3, 0.85, 0.79 );
df.AddRow( 4, 0.62, 0.63 );
df.AddRow( 5, 0.59, 0.54 );
df.AddRow( 6, 0.63, 0.55 );
df.AddRow( 7, 0.64, 0.62 );
df.AddRow( 8, 0.70, 0.67 );
df.AddRow( 9, 0.73, 0.68 );
df.AddRow( 10, 0.68, 0.64 );
```

Code Example – VB paired t-test

```
Dim DF As New DataFrame()
DF.AddColumn(New DFNumericColumn("Baseline"))
DF.AddColumn(New DFNumericColumn("Vit E"))
```

```
DF.AddRow(1, 0.66, 0.6)
DF.AddRow(2, 0.72, 0.65)
DF.AddRow(3, 0.85, 0.79)
DF.AddRow(4, 0.62, 0.63)
DF.AddRow(5, 0.59, 0.54)
DF.AddRow(6, 0.63, 0.55)
DF.AddRow(7, 0.64, 0.62)
DF.AddRow(8, 0.7, 0.67)
DF.AddRow(9, 0.73, 0.68)
DF.AddRow(10, 0.68, 0.64)
```

And this code constructs a **TwoSamplePairedTTest** from the two columns of data:

Code Example – C# paired t-test

```
var test =
   new TwoSamplePairedTTest( df[ "Baseline" ], df[ "Vit E" ] );
```

Code Example – VB paired t-test

```
Dim Test As New TwoSamplePairedTTest(DF("Baseline"), DF("Vit E"))
```

The mean difference between paired measurements, the standard deviation, and the sample size are calculated from the given data.

In addition to the properties common to all hypothesis test objects (Section 41.1), a **TwoSamplePairedTTest** object provides the following read-only properties:

- `Xbar` gets the mean of the differences between paired observations.
- `S` gets the standard deviation of the differences between paired observations.
- `N` gets the number of pairs.
- `DegreesOfFreedom` gets the degrees of freedom.

By default, a **TwoSamplePairedTTest** object performs a two-sided hypothesis test ($H_1: \mu_d \neq 0$) 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.

Once you've constructed and configured a **TwoSamplePairedTTest** object, you can access the various test results using the provided properties, as described in Section 41.1:

Code Example – C# paired t-test

```
Console.WriteLine( "t-statistic = " + test.Statistic );
Console.WriteLine( "deg of freedom = " + test.DegreesOfFreedom );
Console.WriteLine( "p-value = " + test.P );
Console.WriteLine( "reject the null hypothesis? " + test.Reject);
```

Code Example – VB paired t-test

```
Console.WriteLine("t-statistic = " & Test.Statistic)
Console.WriteLine("deg of freedom = " & Test.DegreesOfFreedom)
Console.WriteLine("p-value = " & Test.P)
Console.WriteLine("reject the null hypothesis? " & Test.Reject)
```

The output is:

```
t-statistic = 5.4
deg of freedom = 9
p-value = 0.000433006432003502
reject the null hypothesis? True
```

This indicates that we can reject the null hypotheses ($H_0 : \mu_d = 0$). We can conclude that the true mean thickness of plaque after two years treatment with Vitamin E is significantly different than before treatment.

Finally, remember that the `ToString()` method returns a formatted string representation of the complete test results:

```
Two Sample t Test (Paired)
--------------------------

Mean of differences between pairs = 0.045
Standard deviation of differences between pairs =
0.0263523138347365
Sample size (number of pairs) = 10
Computed t statistic: 5.4, df = 9

Hypothesis type: two-sided
Null hypothesis: true mean of differences between pairs = 0
Alt hypothesis: true mean of differences between pairs != 0
P-value: 0.000433006432003502
REJECT the null hypothesis for alpha = 0.01
0.99 confidence interval: 0.0179180371533991 0.0720819628466008
```

41.5 Two Sample Unpaired T-Test

Class **TwoSampleUnpairedTTest** tests whether two samples from a normal distribution could have the same mean when the standard deviations are unknown but assumed to be equal, allowing for a pooled estimate of the variance.

Class **TwoSampleUnpairedUnequalTTest** assumes that the samples may come from populations with unequal variances, and the Welch-Satterthwaite approximation to the degrees of freedom is used. Unlike **TwoSampleUnpairedTTest**, a pooled estimate of the variance is not used.

For example, suppose we work for a company that makes plastic widgets and we want to compare plastic samples from two suppliers for strength. We record the breaking strength in psi (pounds per square inch) for random samples from each supplier and obtain the following data: `11` samples from the first supplier having a mean strength of `4.2` psi and a standard deviation of `4.68`; `8` samples from the second supplier have a mean strength of `5.6` and a standard deviation of `3.92`.

As described Section 41.1, all hypothesis test classes provide two paths for constructing instances of that type: a parameter-based method and a data-based method. Thus, you can construct a **TwoSampleUnpairedTTest** object by explicitly specifying the mean ($\bar{x}$), standard deviation (s), and size (n) of each sample, like so:

Code Example – C# unpaired t-test

```
double xbar1 = 4.2;
double s1 = 4.68;
int n1 = 11;

double xbar2 = 5.6;
double s2 = 3.92;
int n2 = 8;

var test = new TwoSampleUnpairedTTest( xbar1, s1, n1, xbar2, s2, n2
);
```

Code Example – VB unpaired t-test

```
Dim XBar1 As Double = 4.2
Dim S1 As Double = 4.68
Dim N1 As Integer = 11

Dim XBar2 As Double = 5.6
Dim S2 As Double = 3.92
Dim N2 As Integer = 8

Dim Test As New
  TwoSampleUnpairedTTest(XBar1, S1, N1, XBar2, S2, N2)
```

Or by supplying two sets of sample data. For instance, if the sample data is in two vectors `supplier1` and `supplier2`:

Code Example – C# unpaired t-test

```
var test =
  new TwoSampleUnpairedTTest( supplier1, supplier2 );
```

Code Example – VB unpaired t-test

```
Dim Test As New TwoSampleUnpairedTTest(Supplier1, Supplier2)
```

The sample means, standard deviations, and sizes are calculated from the given data.

In addition to the properties common to all hypothesis test objects (Section 41.1), a **TwoSampleUnpairedTTest** object provides the following read-only properties:

- `Xbar1` and `Xbar2` get the means of the samples.
- `S1` and `S2` get the standard deviations of the samples.
- `SPooled` gets the pooled estimate of the standard deviation.
- `N1` and `N2` get the sizes of the samples.
- `DegreesOfFreedom` gets the degrees of freedom.

By default, a **TwoSampleUnpairedTTest** object performs a two-sided hypothesis test ($H_1: \mu_1 - \mu_2 \neq 0$) 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.

Once you've constructed and configured a **TwoSampleUnpairedTTest** object, you can access the various test results using the provided properties, as described in Section 41.1:

Code Example – C# unpaired t-test

```
Console.WriteLine( "t-statistic = " + test.Statistic );
Console.WriteLine( "pooled standard deviation = " + test.SPooled );
Console.WriteLine( "deg of freedom = " + test.DegreesOfFreedom );
Console.WriteLine( "p-value = " + test.P );
Console.WriteLine( "reject the null hypothesis? " + test.Reject);
```

Code Example – VB unpaired t-test

```
Console.WriteLine("t-statistic = " & Test.Statistic)
Console.WriteLine("pooled standard deviation = " & Test.SPooled)
Console.WriteLine("deg of freedom = " & Test.DegreesOfFreedom)
Console.WriteLine("p-value = " & Test.P)
Console.WriteLine("reject the null hypothesis? " & Test.Reject)
```

The output is:

```
t-statistic = -0.687410859118054
pooled standard deviation = 4.38304755647859
degrees of freedom = 17
p-value = 0.501095386120306
reject the null hypothesis? False
```

This indicates that we cannot reject the null hypotheses ($H_0: \mu_1 - \mu_2 = 0$).

Finally, remember that the `ToString()` method returns a formatted string representation of the complete test results:

```
Two Sample t Test (Unpaired)
----------------------------

Sample means = 4.2 and 5.6
Sample standard deviations = 4.68 and 3.92
Sample sizes = 11 and 8
Difference in means = -1.4
Pooled standard deviation = 4.38304755647859
Computed t statistic: -0.687410859118054, df = 17

Hypothesis type: two-sided
Null hypothesis: true difference in means = 0
Alt hypothesis: true difference in means != 0
P-value: 0.501095386120306
Decision: RETAIN the null hypothesis for alpha = 0.05
0.95 confidence interval: -5.69690885703539 2.8969088570354
```

41.6 Two Sample F-Test

Class **TwoSampleFTest** tests whether the variances of two populations are equal. For example, suppose random samples from two normal populations are taken. The first sample consists of `10` observations with a standard deviation of `5.203`; the second sample consists of `25` observations with a standard deviation of `2.623`. At the `0.10` significance level, is there sufficient evidence to suggest that the populations from which these samples were drawn have equal variances?

As described Section 41.1, all hypothesis test classes provide two paths for constructing instances of that type: a parameter-based method and a data-based method. Thus, you can construct a **TwoSampleFTest** object by explicitly specifying the standard deviation (s),and size (n) of each sample, like so:

Code Example – C# F-test

```
double s1 = 5.203;
int n1 = 10;

double s2 = 2.623;
int n2 = 25;

var test = new TwoSampleFTest( s1, n1, s2, n2 );
```

Code Example – VB F-test

```
Dim S1 As Double = 5.203
Dim N1 As Integer = 10

Dim S2 As Double = 2.623
```

```
Dim N2 As Integer = 25

Dim Test As New TwoSampleFTest(S1, N1, S2, N2)
```

Or by supplying two sets of sample data. For instance, if the sample data is in two vectors `v1` and `v2`:

Code Example – C# F-test

```
var test = new TwoSampleFTest( v1, v2 );
```

Code Example – VB F-test

```
Dim Test As New TwoSampleFTest(V1, V2)
```

The sample standard deviations and sizes are calculated from the given data.

In addition to the properties common to all hypothesis test objects (Section 41.1), a **TwoSampleFTest** object provides the following read-only properties:

- `S1` and `S2` get the standard deviations of the samples.
- `N1` and `N2` get the sizes of the samples.
- `DegreesOfFreedom1` gets the numerator degrees of freedom.
- `DegreesOfFreedom2` gets the denominator degrees of freedom.

By default, a **TwoSampleFTest** object performs a two-sided hypothesis test ($H_1:s_1^2/s_2^2 \neq 1$) 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.

Once you've constructed and configured a **TwoSampleFTest** object, you can access the various test results using the provided properties, as described in Section 41.1:

Code Example – C# F-test

```
Console.WriteLine( "t-statistic = " + test.Statistic );
Console.WriteLine( "numerator df = " + test.DegreesOfFreedom1 );
Console.WriteLine( "denomenator df = " + test.DegreesOfFreedom2 );
Console.WriteLine( "p-value = " + test.P );
Console.WriteLine( "reject the null hypothesis? " + test.Reject);
```

Code Example – VB F-test

```
Console.WriteLine("t-statistic = " & Test.Statistic)
Console.WriteLine("numerator df = " & Test.DegreesOfFreedom1)
Console.WriteLine("denomenator df = " & Test.DegreesOfFreedom2)
Console.WriteLine("p-value = " & Test.P)
Console.WriteLine("reject the null hypothesis? " & Test.Reject)
```

The output is:

```
F-statistic = 3.93469497446923
numerator df = 9
denominator df = 24
p-value = 0.00693561186501657
reject the null hypothesis? True
```

This indicates that we cannot reject the null hypotheses ($H_0:s_1^2/s_2^2 = 1$).

Finally, remember that the `ToString()` method returns a formatted string representation of the complete test results:

```
Two Sample F Test
-----------------

Sample Sizes = 10 and 25
Standard Deviations = 5.203 and 2.623
Variances = 27.071209 and 6.880129
Computed F statistic: 3.93469497446923, num df = 9, denom df = 24

Hypothesis type: two-sided
Null hypothesis: true ratio of variances = 1
Alt hypothesis: true ratio of variances != 1
P-value: 0.00693561186501657
REJECT the null hypothesis for alpha = 0.01
0.99 confidence interval: 1.06490202325594 22.5425454339445
```

41.7 Pearson's Chi-Square Test

NMath Stats provides class **PearsonsChiSquareTest** for performing Pearson's chi-square test. Pearson's chi-square test is the most well-known of the chi-square tests, which are statistical procedures whose results are evaluated by reference to the chi-square distribution. It tests the null hypothesis that the frequency distribution of experimental outcomes are consistent with a particular theoretical distribution. The event outcomes considered must be mutually exclusive and have a total probability of `1`.

Instances of **PearsonsChiSquareTest** are constructed either from raw data or tables of counts. For example, this code constructs a **PearsonsChiSquareTest** using outcomes from a series of experiment runs, along with the expected frequencies:

Code Example – C# chi-square test

```
int[] outcomes = { 59, 20, 11, 10 };
var probs = new DoubleVector( 0.5625, 0.1875, 0.1875, 0.0625 );
var test = new PearsonsChiSquareTest( outcomes, probs );
```

Code Example – VB chi-square test

```
Dim Outcomes() As Integer = {59, 20, 11, 10}
Dim Probs As New DoubleVector(0.5625, 0.1875, 0.1875, 0.0625)
Dim Test As New PearsonsChiSquareTest(Outcomes, Probs)
```

This code uses a *contingency table* (or *cross tabulation*) to store the relation between two or more categorical variables:

Code Example – C# chi-square test

```
var data = new int[2, 2];
data[0, 0] = 4298;
data[0, 1] = 767;
data[1, 0] = 7136;
data[1, 1] = 643;
bool yatesCorrect = true;
var test = new PearsonsChiSquareTest( data, yatesCorrect );
```

Code Example – VB chi-square test

```
Dim Data(2, 2) As Integer
Data(0, 0) = 4298
Data(0, 1) = 767
Data(1, 0) = 7136
Data(1, 1) = 643
Dim YatesCorrect As Boolean = True
Dim Test As New PearsonsChiSquareTest(Data, YatesCorrect)
```

The Yates' correction for continuity may optionally be applied.

Once you've constructed and configured a **PearsonsChiSquareTest** object, you can access the various test results using the provided properties, as described in Section 41.1:

Code Example – C# chi-square test

```
Console.WriteLine( "chi-square statistic = " +
    test.ChiSquareStatistic );
Console.WriteLine( "numerator df = " + test.DegreesOfFreedom );
Console.WriteLine( "p-value = " + test.P );
Console.WriteLine( "reject the null hypothesis? " + test.Reject );
```

Code Example – VB chi-square test

```
Console.WriteLine("chi-square statistic = " &
Test.ChiSquareStatistic)
Console.WriteLine("numerator df = " & Test.DegreesOfFreedom)
Console.WriteLine("p-value = " & Test.P)
Console.WriteLine("reject the null hypothesis? " & Test.Reject)
```

The output is:

```
chi-square statistic = 147.761248704421
```

```
numerator df = 1
p-value = 0
reject the null hypothesis? True
```

Again, the `ToString()` method returns a formatted string representation of the complete test results:

```
Pearson chi-square test
-----------------

Sample size = 12844
Yates corrected = True
Computed chi-square statistic: 147.761248704421, df = 1

P-value: 0
REJECT the null hypothesis for alpha = 0.01
```

41.8 Fisher's Exact Test

StatsFunctions provides the `FisherEactTest()` method for performing a Fisher's Exact Test for a specified `2 x 2` contingency table. Fisher's Exact Test is a useful alternative to the chi-square test in cases where sample sizes are small.

Fisher's Exact Test is so-called because the significance of the deviation from a null hypothesis can be calculated exactly, rather than relying on an approximation. The usual rule of thumb for deciding whether the chi-squared approximation is good enough is whether the expected values in all cells of the contingency table is greater than or equal to `5`.

You can perform a Fisher's Exact Test by providing the cell values directly, plus an **HypothesisType** specifying the form of the alternative hypothesis:

Code Example – C# Fisher's exact test

```
int a = 12, b = 17, c = 4, d = 25;
double pvalue = StatsFunctions.FishersExactTest( a, b, c, d,
  HypothesisType.TwoSided );
```

Code Example – VB Fisher's exact test

```
Dim A As Integer = 12
Dim B As Integer = 17
Dim C As Integer = 4
Dim D As Integer = 25
Dim PValue As Double = StatsFunctions.FishersExactTest(A, B, C, D,
HypothesisType.TwoSided)
```

Values `a`, `b`, `c` and `d` are cell counts for contingency table:

```
a   b
c   d
```

If no hypothesis type is specified, `FisherExactTest()` returns the lesser of the right and left tail *p*-value.

Overloads are also provided for data in an `int[,]` array or **DataFrame** containg two **DFIntColumn**.

CHAPTER 42.
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*).

In a linear model, a quantity y depends on one or more independent variables a_1, $a_2, \ldots, a_n$ such that $y = x_0 + x_1 a_1 + \ldots + x_n a_n$. (Parameter x_0 is called the *intercept parameter*.) Several observations of the independent values a_i are recorded, along with the corresponding values of the dependent variable y. If m observations are performed, and for the *i*th observation we denote the values of the independent variables $a_{i1}, a_{i2}, \ldots, a_{in}$ and the corresponding dependent value of y as y_i, then we form the linear system $Ax = y$, where matrix $A = (a_{ij})$ and vector $y = (y_i)$. The regression solution is the value of x that minimizes $||Ax - y||$.

This chapter describes how to use the **LinearRegression** class, and related supporting classes.

42.1 Creating Linear Regressions

A **LinearRegression** instance is constructed from a predictor matrix and observation vector, like so:

Code Example – C# linear regression

```
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( predictors, obs );
```

Code Example – VB linear regression

```
Dim Predictors As 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 ]")
Dim Obs As New DoubleVector("[ 67 79 61 75 59 90 52 87 ]")
Dim LR As New LinearRegression(Predictors, Obs)
```

A **MismatchedSizeException** is raised if the number of rows in the matrix `A` is not equal to the length of the vector `obs`.

You can also construct a **LinearRegression** instance from data in a **DataFrame**, by indicating which column contains the observations. Non-numeric columns are ignored. For instance, if column `8` contains the dependent variable, this code constructs a regression from the data:

Code Example – C# linear regression

```
var lr = new LinearRegression( df, 8 );
```

Code Example – VB linear regression

```
Dim LR As New LinearRegression(DF, 8)
```

Parameter Calculation by Least Squares Minimization

By default, class **LinearRegression** computes the model parameter values 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.

IRegressionCalculation is the interface for classes used by **LinearRegression** to calculate regression parameters. **NMath Stats** includes three regression calculator classes:

- Class **QRRegressionCalculation** (the default) solves the regression problem using a QR decomposition.
- Class **SVDRegressionCalculation** solves the regression problem using a singular value decomposition.
- Class **CORegressionCalculation** solves least squares problems using a complete orthogonal decomposition.

You can specify a non-default regression calculation object in the constructor. For example:

Code Example – C# linear regression

```
var calcObj = new CORegressionCalculation();
calcObj.Tolerance = 1e-8;
var lr = new LinearRegression( predictors, obs, calcObj );
```

Code Example – VB linear regression

```
Dim CalcObj As New CORegressionCalculation()
CalcObj.Tolerance = 0.00000001
Dim LR As New LinearRegression(Predictors, Obs, CalcObj)
```

The `Tolerance` property is used for computing numerical rank. Values with less than the specified tolerance are considered zero when computing the effective rank.

After construction, the regression calculator used by a **LinearRegression** instance can be changed using the `RegressionCalculator` property.

Intercept Parameters

If the linear model `Ax = y` contains a non-zero intercept parameter, then the first column of matrix `A` must be all ones. Some of the **LinearRegression** constructors allow you to specify whether a column of ones should be prepended to the data in the input regression matrix, or whether the regression matrix should be used as it is given. Thus, this code prepends a column of ones:

Code Example – C# linear regression

```
var lr = new LinearRegression( predictors, obs, true );
```

Code Example – VB linear regression

```
Dim LR As New LinearRegression(Predictors, Obs, True)
```

This code does not:

Code Example – C# linear regression

```
var lr = new LinearRegression( predictors, obs, false );
```

Code Example – VB linear regression

```
Dim LR As New LinearRegression(Predictors, Obs, False)
```

42.2 Regression Results

Class **LinearRegression** provides the following properties for accessing the regression results:

- `IsGood` gets a boolean value indicating whether or not the model parameters were successfully computed.
- `ParameterCalculationErrorMessage` gets any error message produced by the regression calculation object.
- `Parameters` gets the vector of computed model parameters.
- `ParameterEstimates` gets an array of **LinearRegressionParameter** objects suitable for performing hypothesis testing on individual parameters (see Section 42.5).
- `Residuals` gets the vector of residuals. This is the difference between the vector of observed values and the values predicted by the model.
- `Variance` gets an estimate of the variance. This is the residual sum of squares divided by the degrees of freedom for the model. The degrees of freedom for the model is equal to the difference between the number of observations and the number of parameters.
- `CovarianceMatrix` gets the covariance matrix (sometimes called the *dispersion matrix* or *variance-covariance matrix*).

`GetStandardizedResiduals()` gets the standardized residuals (also known as the internally studentized residuals). The residuals are renormalized to have unit variance using an overall measure of error variance.

`GetStudentizedResiduals()` gets the (externally) studentized residuals, which renormalizes the residuals to have unit variance using a leave-one-out measure of error variance—that is, a vector of estimates of the residual variance obtained when the *i*-th case is dropped from the regression.

For more information about a linear regression fit, you can perform hypothesis tests on individual parameters (Section 42.5) or the overall model (Section 42.6).

You can also modify the model and recalculate the parameters, as described in Section 42.4.

Variance Inflation Factor

The variance inflation factor (VIF) quantifies the severity of multicollinearity in a least squares regression analysis—that is, how much the variance of a coefficient is increased because of collinearity. Class **LinearRegression** provides methods `VarianceInflationFactor()` and `VarianceInflationFactors()` for this purpose. For instance:

Code Example – C# linear regression

```
DoubleVector vif = lr.VarianceInflationFactors();
```

Code Example – VB linear regression

```
Dim VIF As DoubleVector = LR.VarianceInflationFactors()
```

42.3 Predictions

You can use a **LinearRegression** object to generate predictions. The `PredictedObservation()` method returns the response predicted by the model for a given set of predictor variable values. For example:

Code Example – C# linear regression

```
var predictors =
  new DoubleVector( 150.0, 33.5, 0.66, 80.0 );
double predicted = lr.PredictedObservation( predictors );
```

Code Example – VB linear regression

```
Dim Predictors As New DoubleVector(150.0, 33.5, 0.66, 80.0)
Dim Predicted As Double = LR.PredictedObservation(Predictors)
```

A **MismatchedSizeException** is raised if the length of the given vector is not equal to the number of parameters in the model.

Similarly, the `PredictedObservations()` method returns the responses predicted by the model for a given collection of predictors:

Code Example – C# linear regression

```
var predictors =
  new DoubleMatrix( "3x4  [ 150.0 33.5 0.66 80.0
                           160.0 24.5 0.88 70.0
                           170.0 22.6 0.56 60.0 ]" );
DoubleVector predicted = lr.PredictedObservations( predictors );
```

Code Example – VB linear regression

```
Dim Predictors As New DoubleMatrix("3x4 [ 150.0 33.5 0.66 80.0
                                          160.0 24.5 0.88 70.0
                                          170.0 22.6 0.56 60.0 ]")
Dim Predicted As DoubleVector =
LR.PredictedObservations(Predictors)
```

In the returned vector of predicted observations, the *i*th element is the predicted response for the set of predictor variable values in the *i*th row of the given matrix.

42.4 Accessing and Modifying the Model

Class **LinearRegression** provides a variety of properties and member functions for accessing and modifying the predictors in the model, the observations, and the intercept option.

Accessing and Modifying Predictors

Class **LinearRegression** provides the following properties for accessing the predictors in the model:

- `RegressionMatrix` gets the regression matrix.
- `PredictorMatrix` gets the predictor matrix. If the model contains an intercept parameter, then the predictor matrix is obtained from the regression matrix by removing the leading column of ones. If the model does not have an intercept parameter then the predictor matrix is the same as the regression matrix.
- `NumberOfParameters` gets the number of parameters in the model.
- `NumberOfPredictors` gets the number of predictors in the model. If the model contains an intercept parameter then the number of predictors is equal to the number of parameters minus one. If the model does not contain an intercept parameter, then the number of predictors is equal to the number of parameters.

If you modify the data in the regression or predictor matrix using the reference returned by `RegressionMatrix` or `PredictorMatrix`, respectively, invoke method `RecalculateParameters()` to recalculate the regression parameters. For instance:

Code Example – C# linear regression

```
lr.PredictorMatrix[2,13] = 15.4;
lr.RecalculateParameters();
```

Code Example – VB linear regression

```
LR.PredictorMatrix(2, 13) = 15.4
LR.RecalculateParameters()
```

Member functions are also provided for adding and removing one or more predictors. The `AddPredictor()` method appends a given column of predictor values to the predictor matrix, and recalculates the parameters:

Code Example – C# linear regression

```
var predictors = new DoubleVector( "[ 1.43 5.5 0.43 14.2 9.0 ]" );

lr.AddPredictor( predictors );
```

Code Example – VB linear regression

```
Dim Predictors As New DoubleVector("[ 1.43 5.5 0.43 14.2 9.0 ]")

LR.AddPredictor(Predictors)
```

A **MismatchedSizeException** is thrown if the number of predictor values is not equal to the number of rows in the regression matrix (also equal to the length of the observation vector).

Similarly, `AddPredictors()` adds a matrix of predictors. Each column of the input matrix is a set of observed predictor values. This, this code adds three predictors:

Code Example – C# linear regression

```
var predictors =
  new DoubleMatrix( " 8x3 [ 1450 .50 70
                            1600 .50 70
                            1450 .70 70
                            1600 .70 70
                            1450 .50 120
                            1600 .50 120
                            1450 .70 120
                            1600 .70 120 ]" );
lr.AddPredictor( predictors );
```

Code Example – VB linear regression

```
Dim Predictors As New DoubleMatrix(" 8x3 [ 1450 .50 70
                                           1600 .50 70
                                           1450 .70 70
                                           1600 .70 70
                                           1450 .50 120
                                           1600 .50 120
                                           1450 .70 120
                                           1600 .70 120 ]")
LR.AddPredictor(predictors)
```

The `RemovePredictor()` method removes the *i*th predictor from the model and recalculates the parameters. This code removes the predictor at (zero-based) index `4`:

Code Example – C# linear regression

```
lr.RemovePredictor( 4 );
```

Code Example – VB linear regression

```
LR.RemovePredictor(4)
```

If the model has an intercept parameter, removing the *0*th predictor will *not* remove the intercept parameter. Use the `RemoveInterceptParameter()` method to remove the intercept parameter (see below).

RemovePredictors() removes the specified number of columns from the predictor matrix beginning with the specified column. Thus, this code removes the second, third, and fourth predictors:

Code Example – C# linear regression

```
lr.RemovePredictors( 1, 3 );
```

Code Example – VB linear regression

```
LR.RemovePredictors(1, 3)
```

Accessing and Modifying Observations

The Observations property gets the vector of observations. If you use the returned reference to modify the observation vector, invoke method RecalculateParameters() to recalculate the regression parameters. For instance:

Code Example – C# linear regression

```
lr.Observations[5] = 0.965;
lr.RecalculateParameters();
```

Code Example – VB linear regression

```
LR.Observations(5) = 0.965
LR.RecalculateParameters()
```

The NumberOfObservations property gets the number of observations, which is simply the length of the observation vector, and also the number of rows in the regression matrix.

Member functions are also provided for adding and removing one or more observations. The AddObservation() method appends a given row of predictor values to the predictor matrix and a given observation to the observation vector, and recalculates the parameters:

Code Example – C# linear regression

```
var predictors =
  new DoubleVector( "[ 1.43 5.5 0.43 14.2 9.0 ]" );
double obs = 2.5;

lr.AddObservation( predictors, obs );
```

Code Example – VB linear regression

```
Dim Predictors As New DoubleVector("[ 1.43 5.5 0.43 14.2 9.0 ]")
Dim Obs As Double = 2.5

LR.AddObservation(Predictors, Obs)
```

NOTE—If the model has an intercept parameter, do not include the leading one in the predictors vector. It will be accounted for in the model.

A **MismatchedSizeException** is thrown if the length of the predictors vector is not equal to the number of predictors in the model.

Similarly, `AddObservations()` adds a collection of observations:

Code Example – C# linear regression

```
var predictors =
  new DoubleMatrix( "3x4  [ 150.0 33.5 0.66 80.0
                            160.0 24.5 0.88 70.0
                            170.0 22.6 0.56 60.0 ]" );
var obs = new DoubleVector( "14.2, 15.5, 10.3" );

lr.AddObservation( predictors, obs );
```

Code Example – VB linear regression

```
Dim Predictors As New DoubleMatrix("3x4 [ 150.0 33.5 0.66 80.0
                                          160.0 24.5 0.88 70.0
                                          170.0 22.6 0.56 60.0 ]")
Dim obs As New DoubleVector("14.2, 15.5, 10.3")

LR.AddObservation(Predictors, Obs)
```

`RemoveObservation()` removes the row at the indicated index from the predictor matrix and the corresponding element from the observation vector. This code removes the observation at (zero-based) index `3`:

Code Example – C# linear regression

```
lr.RemoveObservation( 3 );
```

Code Example – VB linear regression

```
LR.RemoveObservation(3)
```

`RemoveObservations()` removes the specified number of rows from the predictor matrix beginning with the specified row. Thus, this code removes the third, fourth, fifth, and sixth observations:

Code Example – C# linear regression

```
lr.RemoveObservations( 2, 4 );
```

Code Example – VB linear regression

```
LR.RemoveObservations(2, 4)
```

Accessing and Modifying the Intercept Option

The `HasInterceptParameter` property gets a boolean value indicating whether or not the model already has an intercept parameter.

The `AddInterceptParameter()` method adds an intercept parameter to the model and recalculates the parameters. Thus, this code prepends a column of one to the regression matrix:

Code Example – C# linear regression

```
lr.AddInterceptParameter()
```

Code Example – VB linear regression

```
LR.AddInterceptParameter()
```

NOTE—If the model already has an intercept parameter AddInterceptParameter() has no effect.

The `RemoveInterceptParameter()` method removes the intercept parameter.

Updating the Entire Model

Method `SetRegressionData()` updates the entire model by setting the regression matrix, the observation vector, and the intercept option to the specified values, and recalculating the model parameters. For instance:

Code Example – C# linear regression

```
var A = 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 ]" );

lr.SetRegressionData( A, obs, true );
```

Code Example – VB linear regression

```
Dim A As 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 ]")
Dim Obs As New DoubleVector("[ 67 79 61 75 59 90 52 87 ]")

LR.SetRegressionData(A, Obs, True)
```

42.5 Significance of Parameters

Instances of class **LinearRegressionParameter** test statistical hypothesis about individual parameters in a **LinearRegression**.

Creating Linear Regression Parameter Objects

You can construct a **LinearRegressionParameter** from a **LinearRegression** object and the index of the parameter you wish to test. For instance, this code creates a test object for the third parameter:

Code Example – C# linear regression

```
var param = new LinearRegressionParameter( lr, 2 );
```

Code Example – VB linear regression

```
Dim Param As New LinearRegressionParameter(LR, 2)
```

Alternatively, you can get an array of test objects for all parameters in a linear regression using the `ParameterEstimates` property on **LinearRegression**:

Code Example – C# linear regression

```
LinearRegressionParameter[] params = lr.ParameterEstimates;
```

Code Example – VB linear regression

```
Dim Params() As LinearRegressionParameter = LR.ParameterEstimates
```

Properties Linear Regression Parameters

Class **LinearRegressionParameter** provides the following properties:

- `Value` gets the value of the parameter.
- `StandardError` gets the standard error of the parameter.
- `ParameterIndex` gets the index of the parameter in the linear regresssion.

Hypothesis Tests

Class **LinearRegressionParameter** provides the following methods for testing statistical hypotheses regarding parameter values:

- `TStatisticPValue()` returns the *p*-value for a two-sided *t* test with the null hypothesis that the parameter is equal to a given test value, versus the alternative hypothesis that it is not.
- `TStatistic()` returns the value of the *t* statistic for the null hypothesis that the parameter value is equal to a given test value.
- `TStatisticCriticalValue()` gets the critical value for the t-statistic for a given alpha level.
- `ConfidenceInterval()` returns a $1 - \alpha$ confidence interval for the parameter for a given alpha level.

For example, this code tests whether the fifth parameter in a model is significantly different than zero:

Code Example – C# linear regression

```
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 );
```

Code Example – VB linear regression

```
Dim Param As New LinearRegressionParameter(LR, 4)
Dim TStat As Double = Param.TStatistic(0.0)
Dim PValue As Double = Param.TStatisticPValue(0.0)
Dim CriticalValue As Double = Param.TStatisticCriticalValue(0.05)
Dim ConfidenceInterval As Interval = Param.ConfidenceInterval(0.05)
```

Updating Linear Regression Parameters

The `SetRegression()` method updates the regression and parameter index in a parameter test object:

Code Example – C# linear regression

```
param.SetRegression( lr, 6 );
```

Code Example – VB linear regression

```
Param.SetRegression(LR, 6)
```

42.6 Significance of the Overall Model

Class **LinearRegressionAnova** tests the overall model significance for linear regressions. Simply construct a **LinearRegressionAnova** from a **LinearRegression** object:

Code Example – C# linear regression

```
var lrAnova = new LinearRegressionAnova( lr );
```

Code Example – VB linear regression

```
Dim LRAnova As New LinearRegressionAnova(LR)
```

A variety of properties are provided for assessing the significance of the overall model:

- `RegressionSumOfSquares` gets the regression sum of squares. This quantity indicates the amount of variability explained by the model. It is the sum of the squares of the difference between the values predicted by the model and the mean.
- `ResidualSumOfSquares` gets the residual sum of squares. This is the sum of the squares of the differences between the predicted and actual observations.
- `ModelDegreesOfFreedom` gets the number of degrees of freedom for the model, which is equal to the number of predictors in the model.
- `ErrorDegreesOfFreedom` gets the number of degress of freedom for the model error, which is equal to the number of observations minus the number of model paramters.
- `RSquared` gets the coefficient of determination.
- `AdjustedRsquared` gets the adjusted coefficient of determination.
- `MeanSquaredResidual` gets the mean squared residual. This quantity is the equal to `ResidualSumOfSquares / ErrorDegreesOfFreedom` (equals the number of observations minus the number of model parameters).
- `MeanSquaredRegression` gets the mean squared for the regression. This is equal to `RegressionSumOfSquares / ModelDegreesOfFreedom` (equals the number of predictors in the model).
- `FStatistic` gets the overall F statistic for the model. This is equal to the ratio of `MeanSquaredRegression / MeanSquaredResidual`. This is the statistic for the hypothesis test where the null hypothesis, H_0 is that all the parameters are equal to `0` and the alternative hypothesis is that at least one paramter is nonzero.

- `FStatisticPValue` gets the *p*-value for the *F* statistic.

For example:

Code Example – C# linear regression

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

Code Example – VB linear regression

```
Dim LRAnova As New LinearRegressionAnova(LR)
Dim SSE As Double = LRAnova.ResidualSumOfSquares
Dim R2 As Double = LRAnova.RSquared
Dim FStat As Double = LRAnova.FStatistic
Dim FStatPVal As Double = LRAnova.FStatisticPValue
```

Lastly, the `FStatisticCriticalValue()` function computes the critical value for the *F* statistic at a given significance level:

Code Example – C# linear regression

```
double critVal = lrAnova.FStatisticCriticalValue(.05);
```

Code Example – VB linear regression

```
Dim CritVal As Double = LRAnova.FStatisticCriticalValue(0.05)
```

CHAPTER 43.
LOGISTIC REGRESSION

Class **LogisticRegression** performs a binomial logistic regression.

Logistic regression is used to model the relationship between a binary response variable and one or more predictor variables, which may be either discrete or continuous. Binary outcome data is common in medical applications. For example, the binary response variable might indicate whether or not a patient is alive five years after treatment for cancer or whether the patient has an adverse reaction to a new drug. As in multiple linear regression (Chapter 42), we are interested in finding an appropriate combination of predictor variables to help explain the binary outcome.

This chapter describes how to use the **LogisticRegression** class, and related supporting classes.

43.1 Regression Calculators

Class **LogisticRegression** is templatized on the **ILogisticRegressionCalc** calculator to use to calculate the parameters of the logistic regression model. Two implementations are provided:

- **NewtonRaphsonParameterCalc** computes the parameters to maximize the log likelihood function for the model using the Newton Raphson algorithm to compute the zeros of the first order partial derivatives of the log likelihood function. This algorithm is equivalent to, and sometimes referred to, as *iteratively reweighted least squares*. Each iteration involves solving a linear system of the form `X'WX = b`, where `X` is the regression matrix, `X'` is its transpose, and `W` is a diagonal matrix of weights.

  The matrix `X'WX` will be singular if the matrix `X` does not have full rank. **NewtonRaphsonParameterCalc** has property `FailIfNotFullRank` which, if `true`, fails in this case. If `FailIfNotFullRank` is `false`, the linear system is solved using a pseudo-inverse, and the calculation will not fail.

- **TrustRegionParameterCalc** computes the parameters to maximize the log likelihood function for the model, using a trust region optimization algorithm to compute the zeros of the first order partial derivative of the log likelihood function. This approach is more robust than Newton Raphson with design matrices of less than full rank.

  The minimization is performed by an instance of **TrustRegionMinimizer**, and **TrustRegionParameterCalc** instances may be constructed with a given minimizer with the desired algorithm properties.

43.2 Creating Logistic Regressions

A **LogisticRegression** object is constructed from data in the following format: a matrix whose rows contain the predictor variable values, and an **IList<bool>** for the observed values.

Code Example – C# logistic regression

```
DoubleMatrix A = ...
bool[] obs = ...
var lr = new LogisticRegression<NewtonRaphsonParameterCalc>(
  A, obs );
```

Code Example – VB logistic regression

```
Dim A As DoubleMatrix = ...
Dim Obs() As Boolean = ...
Dim LR As New LogisticRegression(Of NewtonRaphsonParameterCalc)(A,
  Obs)
```

A **MismatchedSizeException** is raised if the number of rows in the matrix `A` is not equal to the length of the vector `obs`.

If you want the model to have an intercept parameter, you can specify that as well:

Code Example – C# logistic regression

```
bool addIntercept = true;
var lr = new LogisticRegression<NewtonRaphsonParameterCalc>(
  A, obs, addIntercept );
```

Code Example – VB logistic regression

```
Dim AddIntercept As Boolean = True
Dim LR As New LogisticRegression(Of NewtonRaphsonParameterCalc)(A,
  Obs, AddIntercept)
```

If `true`, a column of ones is prepended onto the data in the regression matrix `A`, thus adding an intercept to the model. If `false`, the data in the regression matrix is used as given.

You can also provide a regression calculator instance to use. For example, if you want regression to fail consistently when the regression matrix is rank deficient, you can construct a **NewtonRaphsonParameterCalc** object with the `FailIfNotFullRank` property set to `true` (see Section 43.1), then construct a **LogisticRegression** object with the resulting parameter calculation object:

Code Example – C# logistic regression

```
var parameterCalc = new NewtonRaphsonParameterCalc() {
  FailIfNotFullRank = true };
var lr = new LogisticRegression<NewtonRaphsonParameterCalc>(
  A, obs, addIntercept, parameterCalc );
```

Code Example – VB logistic regression

```
Dim ParameterCalc As New NewtonRaphsonParameterCalc()
ParameterCalc.FailIfNotFullRank = True
Dim LR As New LogisticRegression(Of NewtonRaphsonParameterCalc)(A,
  Obs, AddIntercept, ParameterCalc)
```

Additional **LogisticRegression** constructors provide flexibility in how the observation values are specified. For example, you can provide a vector of floating point observation values, which is converted to dichotomous values using a supplied **Predictate<double>** function. This code uses a lambda expression to specify the predicate:

Code Example – C# logistic regression

```
DoubleVector v = ...
var lr = new LogisticRegression<NewtonRaphsonParameterCalc>(
  A, v, x => x >= 110.0, addIntercept);
```

Code Example – VB logistic regression

```
Dim V As DoubleVector = ...
Dim LR As New LogisticRegression(Of NewtonRaphsonParameterCalc)(A,
  V, X = X >= 110.0, AddIntercept)
```

Similarly, you can provide the observation values as one of the columns of the regression matrix:

Code Example – C# logistic regression

```
int observationColIndex = 0;
var lr = new LogisticRegression<NewtonRaphsonParameterCalc>(
  A, observationColIndex, x => x != 0, addIntercept);
```

Code Example – VB logistic regression

```
Dim ObservationColIndex As Integer = 0
```

```
Dim LR As New LogisticRegression(Of NewtonRaphsonParameterCalc)(A,
  ObservationColIndex, X = X <> 0, AddIntercept)
```

Design Variables

LogisticRegression provides static convenience method `DesignVariables()` for producing design, or dummy, variables using *reference cell coding*. If the categorical variable has *k* levels, there will be *k - 1* design variables created. Reference cell coding involves setting all the design variable values to `0` for the reference group, and then setting a single design variable equal to `1` for each of the other groups.

For example, suppose we have a **DataFrame** `df` with a column of race values, which has three levels.

Code Example – C# logistic regression

```
int raceColIndex = df.IndexOfColumn( "Race" );
DataFrame raceDesignVars =
  LogisticRegression<NewtonRaphsonParameterCalc>.DesignVariables(
    df[raceColIndex] );
```

Code Example – VB logistic regression

```
Dim RaceColIndex As Integer = DF.IndexOfColumn("Race")
Dim RaceDesignVars As DataFrame = LogisticRegression(Of
  NewtonRaphsonParameterCalc).DesignVariables(DF(RaceColIndex))
```

Since the race variable has three levels there will be two design variables. By default they will be named `Race_0` and `Race_1`.

We then replace the original race column with the two design variable columns, and convert the data frame to a matrix of floating point values.

Code Example – C# logistic regression

```
df.RemoveColumn( raceColIndex );
for ( int c = 0; c < raceDesignVars.Cols; c++ )
{
  df.InsertColumn( raceColIndex + c, raceDesignVars[c] );
}
DoubleMatrix matrixDat = data.ToDoubleMatrix();
```

Code Example – VB logistic regression

```
DF.RemoveColumn(RaceColIndex)
Dim C As Integer
For C = 0 To RaceDesignVars.Cols - 1
  DF.InsertColumn(RaceColIndex + C, RaceDesignVars(C))
Next
Dim MatrixDat As DoubleMatrix = DF.ToDoubleMatrix()
```

43.3 Checking for Convergence

After constructing a **LogisticRegression** object, first check that the parameter calculation was successful. For example, this code checks the `IsGood` property, and if the calculation failed, prints out some diagnostic information using the `ParameterCalculationErrorMessage` property.

Code Example – C# logistic regression

```
if ( !lr.IsGood )
{
    Console.WriteLine(
      "Logistic regression parameter calculation failed:" );
    Console.WriteLine( lr.ParameterCalculationErrorMessage );

    var parameterCalc = lr.ParameterCalculator;
    Console.WriteLine( "Maximum iterations: " +
      parameterCalc.MaxIterations );
    Console.WriteLine( "Number of iterations: " +
      parameterCalc.Iterations );
    Console.WriteLine( "Converged? " + parameterCalc.Converged );
}
```

Code Example – VB logistic regression

```
If Not LR.IsGood Then
  Console.WriteLine("Logistic regression parameter calculation
    failed:")
  Console.WriteLine(LR.ParameterCalculationErrorMessage)

  Dim ParameterCalc As ParameterCalc = LR.ParameterCalculator
  Console.WriteLine("Maximum iterations: " &
    ParameterCalc.MaxIterations)
  Console.WriteLine("Number of iterations: " &
    ParameterCalc.Iterations)
  Console.WriteLine("Converged? " & ParameterCalc.Converged)
End If
```

43.4 Goodness of Fit

Class **LogisticRegressionFitAnalysis** calculates *goodness of fit* statistics for a logistic regression model.

Code Example – C# logistic regression

```
var fit = new
  LogisticRegressionFitAnalysis<NewtonRaphsonParameterCalc>( lr );
```

Code Example – VB logistic regression

```
Dim Fit As New LogisticRegressionFitAnalysis(Of
  NewtonRaphsonParameterCalc)(LR)
```

Provided properties access the model statistics:

- `GStatistic` gets the G statistic for the model. The G statistic is

  ```
  G = -2*ln[(likelihood without the variables)/
            (likelihood with the variables)]
  ```

- `GStatisticPValue` gets the *p*-value for the G statistic.
- `LogLikelihood` gets the log likelihood for the model.

For instance:

Code Example – C# logistic regression

```
Console.WriteLine( "Log likelihood: " + fit.LogLikelihood );
Console.WriteLine( "G-statistic: " + fit.GStatistic );
Console.WriteLine( "G-statistic P-value: " +
  fit.GStatisticPValue );
```

Code Example – VB logistic regression

```
Console.WriteLine("Log likelihood: " & Fit.LogLikelihood)
Console.WriteLine("G-statistic: " & Fit.GStatistic)
Console.WriteLine("G-statistic P-value: " & Fit.GStatisticPValue)
```

Two methods on **LogisticRegressionFitAnalysis** provide access to additional statistics:

- `PearsonStatistic()` computes the Pearson chi-square statistic, and related quantities from the Pearson residuals, to determine if two observations share the same covariate pattern.
- `HLStatistic()` calculates the Hosmer Lemeshow statistic for the model. This test assesses whether or not the observed event rates match expected event rates in subgroups of the model population.

For instance, this code calculates the Hosmer Lemeshow statistic using 10 groups.

Code Example – C# logistic regression

```
var hosmerLemeshowStat = fit.HLStatistic(10);
Console.WriteLine(hosmerLemeshowStat);
```

Code Example – VB logistic regression

```
Dim HosmerLemeshowStat = Fit.HLStatistic(10)
Console.WriteLine(HosmerLemeshowStat)
```

43.5 Parameter Estimates

The `ParameterEstimates` property on **LogisticRegression** gets an array of **LogisticRegressionParameter** estimate objects. This class tests statistical hypotheses about estimated parameters in logistic regressions:

- `Value` gets the value of the parameter.
- `StandardError` gets the standard error of the parameter.
- `ParameterIndex` gets the index of the parameter in the linear regresssion.
- `Beta` gets the standardized beta coefficient. Beta coefficients are weighted by the ratio of the standard deviation of the independent variable over the standard deviation of the dependent variable.
- `ConfidenceInterval()` returns the `1 - alpha` confidence interval for the parameter.
- `TStatistic()` returns the *t*-statistic for the null hypothesis that the parameter is equal to a given test value.
- `TStatisticPValue()` returns the *p*-value for a *t*-test with the null hypothesis that the parameter is equal to a given test value versus the alternative hypothesis that it is not.
- `TStatisticCriticalValue()` gets the critical value of the *t*-statistic for the specified alpha level.

For instance, this code prints out the model parameter estimates and standard error.

Code Example – C# logistic regression

```
var parameterEstimates = lr.ParameterEstimates;
for ( int i = 0; i < parameterEstimates.Length; i++ )
{
  var estimate = parameterEstimates[i];
  if ( i == 0 )
  {
    Console.WriteLine( "Constant term = {0}, SE = {1}",
      estimate.Value, estimate.StandardError);
  }
  else
  {
    Console.WriteLine( "Coefficient for {0} = {1}, SE = {2}",
      df[i].Name, estimate.Value, estimate.StandardError);
  }
}
```

Code Example – VB logistic regression

```
Dim ParameterEstimates = LR.ParameterEstimates
For I As Integer = 0 To ParameterEstimates.Length - 1
  Dim Estimate = ParameterEstimates(I)
  If (I = 0) Then
    Console.WriteLine("Constant term = {0}, SE = {1}",
      Estimate.Value, Estimate.StandardError)
  Else
    Console.WriteLine("Coefficient for {0} = {1}, SE = {2}",
      DF(I).Name, Estimate.Value, Estimate.StandardError)
  End If
Next
```

43.6 Predicted Probabilities

You can use a **LogisticRegression** object to generate predictions. The `PredictedProbability()` method returns the probability of a positive outcome predicted by the model for a given set of predictor values. For example:

Code Example – C# logistic regression

```
var predictors =
  new DoubleVector( 150.0, 33.5, 0.66, 80.0 );
double predicted = lr.PredictedProbability( predictors );
```

Code Example – VB logistic regression

```
Dim Predictors As New DoubleVector(150.0, 33.5, 0.66, 80.0)
Dim Predicted As Double = LR.PredictedProbability(Predictors)
```

A **MismatchedSizeException** is raised if the length of the given vector is not equal to the number of parameters in the model.

Similarly, the `PredictedProbabilities()` method returns a vector of predicted probabilities of a positive outcome for the predictor variable values contained in the rows of an input matrix.

Code Example – C# logistic regression

```
var predictors =
  new DoubleMatrix( "3x4 [ 150.0 33.5 0.66 80.0
                           160.0 24.5 0.88 70.0
                           170.0 22.6 0.56 60.0 ]" );
DoubleVector predicted = lr.PredictedProbabilities( predictors );
```

Code Example – VB logistic regression

```
Dim Predictors As New DoubleMatrix("3x4 [ 150.0 33.5 0.66 80.0
                                          160.0 24.5 0.88 70.0
                                          170.0 22.6 0.56 60.0 ]")
Dim Predicted As DoubleVector =
  LR.PredictedProbabilities(Predictors)
```

In the returned vector of predicted observations, the *i*th element is the predicted response for the set of predictor variable values in the *i*th row of the given matrix.

43.7 Auxiliary Statistics

Class **LogisticRegressionAuxiliaryStats** computes auxiliary statistics for logistic regressions, such as pseudo R-squared metrics and odds ratios for the computed coefficients.

Code Example – C# logistic regression auxiliary statistics

```
var auxStats = new
LogisticRegressionAuxiliaryStats<NewtonRaphsonParameterCalc>( lr );
```

Code Example – VB logistic regression auxiliary statistics

```
Dim AuxStats As New LogisticRegressionAuxiliaryStats(Of
  NewtonRaphsonParameterCalc)(LR)
```

Provided properties access the model statistics:

- `CoxSnell` gets the Cox and Snell pseudo R-squared statistic for the model.
- `Nagelkerke` gets the Nagelkerke pseudo R-squared statistic for the model.
- `LogLikelihoodFullModel` gets the log of the value of the likelihood function for the full model (estimated coefficients).
- `LogLikelihoodInterceptOnly` gets the log of the value of the likelihood function for the intercept-only model.
- `OddsRatios` gets the odds ratio values for the computed coefficients. The odds ratio for the intercept parameter, if there is one, is not computed.

Finally, property `LikelihoodRatioStat` gets the likelihood ratio statistic and related values for the logistic regression. The result is returns as an instance of **LikelihoodRatioStatistic**.

Code Example – C# logistic regression auxiliary statistics

```
var lrs = auxStats.LikelihoodRatioStat;
Console.WriteLine( lrs.ChiSquareStatistic );
Console.WriteLine( lrs.RightTailProbability );
```

Code Example – VB logistic regression auxiliary statistics

```
Dim LRS = AuxStats.LikelihoodRatioStat
Console.WriteLine(LRS.ChiSquareStatistic)
Console.WriteLine(LRS.RightTailProbability)
```

CHAPTER 44.
ANALYSIS OF VARIANCE

Analysis of variance (ANOVA) is the multigroup generalization of the *t* test (Chapter 41). Like the *t* test, ANOVA assumes that samples are randomly drawn from normally distributed populations with the same standard deviations. If differences between the observed means of the samples are larger than one would expect from the underlying population variability, estimated by the standard deviations within the samples, you can conclude that at least one of the samples has a different mean than the others.

NMath Stats provides classes for both *one-way* (or *one-factor*) and *two-way* (or *two-factor*) ANOVAs, for both balanced and unbalanced designs, and with or without repeated measures (RANOVA).

This chapter describes the analysis of variance classes.

44.1 One-Way ANOVA

Class **OneWayAnova** computes and summarizes a traditional one-way (single factor) analysis of variance.

Creating One-Way ANOVA Objects

A **OneWayAnova** instance is constructed from numeric data organized into different groups. The groups need not contain the same number of observations. For example, this code constructs a **OneWayAnova** from an array of **DoubleVector** objects. Each vector in the array contains data for a single group:

Code Example – C# ANOVA

```
var data = new DoubleVector[5];

data[0] = new DoubleVector( "[24 15 21 27 33 23]" );
data[1] = new DoubleVector( "[14 7 12 17 14 16]" );
data[2] = new DoubleVector( "[11 9 7 13 12 18]" );
data[3] = new DoubleVector( "[7 7 4 7 12 18]" );
data[4] = new DoubleVector( "[19 24 19 15 10 20]" );

var anova = new OneWayAnova( data );
```

Code Example – VB ANOVA

```
Dim Data As New DoubleVector(5)

Data(0) = New DoubleVector("[24 15 21 27 33 23]")
Data(1) = New DoubleVector("[14 7 12 17 14 16]")
Data(2) = New DoubleVector("[11 9 7 13 12 18]")
Data(3) = New DoubleVector("[7 7 4 7 12 18]")
Data(4) = New DoubleVector("[19 24 19 15 10 20]")

Dim Anova As New OneWayAnova(Data)
```

This code constructs a **OneWayAnova** from a data frame `df`:

Code Example – C# ANOVA

```
var anova = new OneWayAnova( df, 1, 3 );
```

Code Example – VB ANOVA

```
Dim Anova As New OneWayAnova(DF, 1, 3)
```

Two column indices are also provided: a *group* column and a *data* column. A **Factor** is constructed from the group column using the **DataFrame** method `GetFactor()`, which creates a sorted array of the unique values. The specified data column must be of type **DFNumericColumn**.

Lastly, you can also construct a **OneWayAnova** from a **DoubleMatrix**:

Code Example – C# ANOVA

```
var data = new DoubleMatrix( "6 x 5 [ 24 14 11 7 19
                                      15 7 9 7 24
                                      21 12 7 7 19
                                      27 17 13 12 15
                                      33 14 12 12 10
                                      23 16 18 18 20 ]" );
var anova = new OneWayAnova( data );
```

Code Example – VB ANOVA

```
Dim Data As New DoubleMatrix("6 x 5 [ 24 14 11 7 19
                                      15 7 9 7 24
                                      21 12 7 7 19
                                      27 17 13 12 15
                                      33 14 12 12 10
                                      23 16 18 18 20 ]")
Dim Anova As New OneWayAnova(Data)
```

Each column in the given matrix contains the data for a group. If your groups have different numbers of observations, you must pad the columns with `Double.NaN` values until they are all the same length, because a **DoubleMatrix** must be rectangular. Alternatively, use one of the other constructors described above.

The One-Way ANOVA Table

Once you've constructed a **OneWayAnova**, you can display the complete ANOVA table:

Code Example – C# ANOVA

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

Code Example – VB ANOVA

```
Console.WriteLine(Anova)
```

For example:

```
Source      Deg of Freedom    Sum Of Sq  Mean Sq     F       P
Between groups     4          803.0000   200.7500  9.0076  0.0001
Within groups      25         557.1667   22.2867   .       .
Total              29         1360.1667  46.9023   .       .
```

Class **OneWayAnovaTable** is provided for summarizing the information in a traditional one-way ANOVA table. Class **OneWayAnovaTable** derives from **DataFrame**. An instance of **OneWayAnovaTable** can be obtained from a **OneWayAnova** object using the `AnovaTable` property. For example:

Code Example – C# ANOVA

```
OneWayAnovaTable myTable = anova.AnovaTable;
```

Code Example – VB ANOVA

```
Dim MyTable As OneWayAnovaTable = Anova.AnovaTable
```

Class **OneWayAnovaTable** provides the following read-only properties for accessing individual elements in the ANOVA table:

- `DegreesOfFreedomBetween` gets the between-groups degrees of freedom.
- `DegreesOfFreedomWithin` gets the within-groups degrees of freedom.
- `DegreesOfFreedomTotal` gets the total degrees of freedom.
- `SumOfSquaresBetween` gets the between-groups sum of squares.
- `SumOfSquaresWithin` gets the within-groups sum of squares.
- `SumOfSquaresTotal` gets the total sum of squares.
- `MeanSquareBetween` gets the between-groups mean square. The between-groups mean square is the between-groups sum of squares divided by the between-groups degrees of freedom.

- `MeanSquareWithin` gets the within-group mean square. The within-groups mean square is the within-group sum of squares divided by the within-group degrees of freedom.
- `MeanSquareTotal` gets the total mean square. The total mean square is the total sum of squares divided by the total degrees of freedom.
- `FStatistic` gets the F statistic.
- `FStatisticPValue` gets the p-value for the F statistic.

Grand Mean, Group Means, and Group Sizes

Class **OneWayAnova** provides properties and methods for retrieving the grand mean, group means, and group sizes:

- `GrandMean` gets the grand mean of the data. The grand mean is the mean of all of the data.
- `GroupMeans` gets a vector of group means.
- `GroupSizes` gets an array of group sizes.
- `GroupNames` gets an array of group names. If the anova was constructed from a data frame using a grouping column, the group names are the sorted, unique **Factor** levels created from the column values. If the anova object was constructed from a matrix or an array of vectors, the group names are simply `Group_0, Group_1...Group_n`.
- `GetGroupMean()` returns the mean for a specified group, identified either by group name or group number (a zero-based index into the `GroupMeans` vector).
- `GetGroupSize()` returns the mean for a specified group, identified either by group name or group number (a zero-based index into the `GroupSizes` array).

For example, if a **OneWayAnova** is constructed from a matrix, this code returns the mean for the group in the third column of the matrix:

Code Example – C# ANOVA

```
double maleMean = anova.GetGroupMean( 2 );
```

Code Example – VB ANOVA

```
Dim MaleMean As Double = Anova.GetGroupMean(2)
```

If a **OneWayAnova** is constructed from a data frame using a grouping column with values `male` and `female`, this code returns the mean for the `male` group:

Code Example – C# ANOVA

```
double maleMean = anova.GetGroupMean( "male" );
```

Code Example – VB ANOVA

```
Dim MaleMean As Double = Anova.GetGroupMean("male")
```

Critical Value of the F Statistic

Class **OneWayAnova** provides the convenience function `FStatisticCriticalValue()` which computes the critical value for the ANOVA F statistic at a given significance level. Thus:

Code Example – C# ANOVA

```
double alpha = 0.05;
double critVal = anova.FStatisticCriticalValue( alpha );
```

Code Example – VB ANOVA

```
Dim Alpha As Double = 0.05
Dim CritVal As Double = Anova.FStatisticCriticalValue(Alpha)
```

Updating One-Way ANOVA Objects

Method `SetData()` updates an entire analysis of variance object with new data. As with the class constructors (see above), you can supply data as an array of group vectors, a matrix, or as a data frame. For instance, this code updates an ANOVA with data from **DataFrame** `df`, using column `2` as the group column and column `5` as the data column:

Code Example – C# ANOVA

```
anova.SetData( df, 2, 5 );
```

Code Example – VB ANOVA

```
Anova.SetData(DF, 2, 5)
```

44.2 One-Way Repeated Measures ANOVA

Class **OneWayRanova** calculates and summarizes the information of a one-way repeated measures analysis of variance (RANOVA).

Creating One-Way RANOVA Objects

A **OneWayRanova** instance is constructed from numeric data for multiple treatments applied to each experimental subject. For example, this code constructs a **OneWayRanova** from a **DoubleMatrix**:

Code Example – C# RANOVA

```
var data = new DoubleMatrix( "8x4 [ 180 200 160 200
                                    230 250 200 220
                                    280 310 260 270
                                    180 200 160 200
                                    190 210 170 210
                                    140 160 120 110
                                    270 300 250 260
                                    110 130 100 100 ]" );
var ranova = new OneWayRanova( data );
```

Code Example – VB RANOVA

```
Dim Data As New DoubleMatrix("8x4 [ 180 200 160 200
                                    230 250 200 220
                                    280 310 260 270
                                    180 200 160 200
                                    190 210 170 210
                                    140 160 120 110
                                    270 300 250 260
                                    110 130 100 100 ]")
Dim Ranova As New OneWayRanova(Data)
```

Each row of the matrix contains the data for an individual subject. There should be one column for each treatment. The example above shows `4` different measurements for each of `8` subjects.

NOTE—Data rows containing missing values (NaNs) are ignored by class OneWayRanova.

Similarly, you can also construct a **OneWayRanova** from a **DataFrame**:

Code Example – C# RANOVA

```
var ranova = new OneWayRanova( df );
```

Code Example – VB RANOVA

```
Dim Ranova As New OneWayRanova(DF)
```

Each row in the **DataFrame** contains the data for an individual subject. There should be one column for each treatment.

Note that all numeric columns in the given **DataFrame** are interpreted as treatments; only non-numeric columns are ignored. If you have numeric columns in the data frame that you also wish to ignore, apply the appropriate **Subset** first. For instance:

Code Example – C# RANOVA

```
var colIndices = new Subset( new int[] { 3, 14, 5, 8, 4 } );
var ranova = new OneWayRanova( df.GetColumns( colIndices ) );
```

Code Example – VB RANOVA

```
Dim ColIndices As New Subset(New Integer() {3, 14, 5, 8, 4})
Dim Ranova As New OneWayRanova(DF.GetColumns(ColIndices))
```

The One-Way RANOVA Table

Once you've constructed a **OneWayRanova**, you can display the complete RANOVA table:

Code Example – C# RANOVA

```
Console.WriteLine( ranova );
```

Code Example – VB RANOVA

```
Console.WriteLine(Ranova)
```

For example:

```
Source    Deg of Freedom  Sum Of Sq    Mean Square     F        P
Subjects       9         102822.5000  11424.7222     .        .
Treatment      3         9247.5000    3082.5000     31.6755  0.0000
Error         27         2627.5000    97.3148        .        .
Total         39         114697.5000  2940.9615      .        .
```

Class **OneWayRanovaTable** is provided for summarizing the information in a traditional one-way RANOVA table. Class **OneWayRanovaTable** derives from **DataFrame**. An instance of **OneWayRanovaTable** can be obtained from a **OneWayRanova** object using the `RanovaTable` property. For example:

Code Example – C# RANOVA

```
OneWayRanovaTable myTable = ranova.RanovaTable;
```

Code Example – VB RANOVA

```
Dim MyTable As OneWayRanovaTable = Ranova.RanovaTable
```

Class **OneWayRanovaTable** provides the following read-only properties for accessing individual elements in the RANOVA table:

- `DegreesOfFreedomTreatment` gets the treatment degrees of freedom.

- `DegreesOfFreedomWithinSubject` gets the within-subject degrees of freedom.
- `DegreesOfFreedomError` gets the error degrees of freedom.
- `DegreesOfFreedomTotal` gets the total degrees of freedom.
- `SumOfSquaresTreatment` gets the treatment sum of squares.
- `SumOfSquaresWithinSubject` gets the within-subject sum of squares.
- `SumOfSquaresTotal` gets the total sum of squares.
- `SumOfSquaresError gets` the error sum of squares.
- `MeanSquareTreatment` gets the treatment mean square.
- `MeanSquareWithinSubject` gets the within-subject mean square.
- `MeanSquareError` gets the error mean square.
- `MeanSquareTotal` gets the total mean square.
- `FStatistic` gets the F statistic for the RANOVA.
- `FStatisticPValue` gets the p-value for the F statistic.

Grand Mean, Subject Means, and Treatment Means

Class **OneWayRanova** provides properties for retrieving the grand mean, subject means, and treatment means:

- `GrandMean` gets the grand mean of the data. The grand mean is the mean of all of the data.
- `SubjectMeans` gets a vector of means for each subject.
- `TreatmentMeans` gets a vector of means for each treatment.

Critical Value of the F Statistic

Class **OneWayRanova** provides the convenience function `FStatisticCriticalValue()` which computes the critical value for the RANOVA F statistic at a given significance level. Thus:

Code Example – C# RANOVA

```
double alpha = 0.01;
double critVal = ranova.FStatisticCriticalValue( alpha );
```

Code Example – VB RANOVA

```
Dim Alpha As Double = 0.01
Dim CritVal As Double = Ranova.FStatisticCriticalValue(Alpha)
```

Updating One-Way RANOVA Objects

Method `SetData()` updates an entire repeated measures analysis of variance object with new data. As with the class constructors (see above), you can supply data as a matrix or as a data frame. For instance, this code updates a RANOVA with data from matrix `A`:

Code Example – C# RANOVA

```
ranova.SetData( A );
```

Code Example – VB RANOVA

```
Ranova.SetData(A)
```

44.3 Two-Way Balanced ANOVA

Class **TwoWayAnova** performs a balanced two-way analysis of variance. Two-way analysis of variance is a direct extension of one-way analysis of variance (Section 44.1). In this case, data are grouped according to two factors—for example, *sex* and *age group*—rather than a single factor. The total variability is partitioned into components associated with each of the two factors, their interaction, and the residual (or error).

Creating Two-Way ANOVA Objects

A **TwoWayAnova** instance is constructed from data in a data frame. Three column indices are specified in the data frame: the column containing the first factor, the column containing the second factor, and the column containing the numeric data. For example, this code groups the numeric data in column `3` of **DataFrame** `df` by factors constructed from columns `0` and `4`:

Code Example – C# ANOVA

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

Code Example – VB ANOVA

```
Dim Anova As New TwoWayAnova(DF, 0, 4, 3)
```

Factor objects are constructed from the factor columns using the **DataFrame** method `GetFactor()`, which creates a sorted array of the unique values

(Section 37.10). The indicated data column must be of type **DFNumericColumn**.

NOTE—Class TwoWayAnova throws an InvalidArgumentException if the data contains missing values (NaNs).

The Two-Way ANOVA Table

Once you've constructed a **TwoWayAnova**, you can display the complete ANOVA table:

Code Example – C# ANOVA

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

Code Example – VB ANOVA

```
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  .             .         .
```

Class **TwoWayAnovaTable** is provided for summarizing the information in a traditional two-way ANOVA table. Class **TwoWayAnovaTable** derives from **DataFrame**. An instance of **TwoWayAnovaTable** can be obtained from a **TwoWayAnova** object using the `AnovaTable` property. For example:

Code Example – C# ANOVA

```
TwoWayAnovaTable myTable = anova.AnovaTable;
```

Code Example – VB ANOVA

```
Dim MyTable As TwoWayAnovaTable = Anova.AnovaTable
```

Class **TwoWayAnovaTable** provides the following member functions and read-only properties for accessing individual elements in the ANOVA table:

- `DegreesOfFreedom()` gets the degrees of freedom for a specified factor.
- `ErrorDegreesOfFreedom` gets the number of degrees of freedom for the error.
- `InteractionDegreesOfFreedom` gets the number of degrees of freedom for the interactions.
- `TotalDegreesOfFreedom` gets the total number of degrees of freedom.

- `SumOfSquares()` gets the sum of squares for a specified factor.
- `InteractionSumOfSquares` gets the sum of squares for the interaction.
- `ErrorSumOfSquares` gets the sum of squares for the error.
- `TotalSumOfSquares` gets the total sum of squares.
- `MeanSquare()` gets the mean square for a specified factor.
- `InteractionMeanSquare` gets the mean square for the interaction.
- `ErrorMeanSquare` gets the mean square for the error.
- `Fstatistic()` gets the *F* statistic for a specified factor.
- `InteractionFstatistic` gets the *F* statistic for the interaction.
- `FstatisticPvalue()` gets the *p*-value for the *F* statistic for a specified factor.
- `InteractionFstatisticPvalue` gets the *p*-value for the *F* statistic for the interaction.

Factors are identified to accessor methods by name, which corresponds to the name of the column in the original data frame that was used to create the **Factor**. For instance, if one factor in the ANOVA is named `Dosage`, this code gets the *F* statistic and *p*-value for that factor:

Code Example – C# ANOVA

```
double Fstatistic = anova.AnovaTable.Fstatistic( "Dosage" );
double Pvalue = anova.AnovaTable.FstatisticPvalue( "Dosage" );
```

Code Example – VB ANOVA

```
Dim FStatistic As Double = Anova.AnovaTable.FStatistic("Dosage")
Dim PValue As Double = Anova.AnovaTable.FStatisticPValue("Dosage")
```

Cell Data

Class **TwoWayAnova** provides the `GetCellData()` method for accessing the data in a cell, as defined by a specified level of each of the factors in the ANOVA. For example, if `anova` has factor `Sex` with levels `Male` and `Female`, and factor `AgeGroup` with levels `Child`, `Adult`, and `Senior`, this code gets the data for adult females:

Code Example – C# ANOVA

```
DFNumericColumn data =
  anova.GetCellData( "Sex", "Female", "AgeGroup", "Adult" );
```

Code Example – VB ANOVA

```
Dim Data As DFNumericColumn =
  Anova.GetCellData("Sex", "Female", "AgeGroup", "Adult")
```

A copy of the data is returned as a **DFNumericColumn** object.

Grand Mean, Cell Means, and Group Means

Class **TwoWayAnova** provides the following properties and member functions for accessing the grand mean, cell means, and group means:

- `GrandMean` gets the grand mean. The grand mean is the mean of all the data.
- `GetMeanForCell()` returns the mean for a specified cell.
- `GetMeanForFactorLevel()` returns the mean for a specified factor level.

Again, factors and factor levels are identified to accessor methods by name. For example, if `anova` has factor `Sex` with levels `Male` and `Female`, and factor `AgeGroup` with levels `Child`, `Adult`, and `Senior`, this code gets the mean for all males:

Code Example – C# ANOVA

```
double meanM = anova.GetMeanForFactorLevel( “Sex”, “Male” );
```

Code Example – VB ANOVA

```
Dim MeanM As Double = Anova.GetMeanForFactorLevel("Sex", "Male")
```

This code gets the mean for male children:

Code Example – C# ANOVA

```
double meanMChild =
  anova.GetMeanForCell( “Sex”, “Male”, “AgeGroup”, “Child” );
```

Code Example – VB ANOVA

```
Dim MeanMChild As Double =
  Anova.GetMeanForCell("Sex", "Male", "AgeGroup", "Child")
```

ANOVA Regression Parameters

NMath Stats 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.

To solve the two-way ANOVA problem using multiple linear regression, **NMath Stats** creates a series of *dummy variables* to encode the different levels of each of the two factors. The specific encoding used, known as *effects encoding*, encodes dummy variables so that the coefficients of the dummy variables in the regression model quantify deviations of each group from the grand mean.[1]

In the effects encoding, $k-1$ dummy variables are defined to encode the k levels of a factor, like so:

$$E_1 = \begin{cases} 1 \text{ if group } 1 \\ -1 \text{ if group k} \\ 0 \text{ othewise} \end{cases}$$

$$E_2 = \begin{cases} 1 \text{ if group } 2 \\ -1 \text{ if group k} \\ 0 \text{ othewise} \end{cases}$$

and so on, up to E_{k-1} for group $k-1$.

For example, suppose we have an experimental design with two factors: `FactorA` and `FactorB`. `FactorA` has two levels, labelled `A1` and `A1`. Effects encoding defines *one* dummy variable for `FactorA`:

$$A = \begin{cases} 1 \text{ if group A1} \\ -1 \text{ if group A2} \end{cases}$$

`FactorB` has three levels, labelled `B1`, `B2`, and `B3`. Effects encoding defines *two* dummy variable for `FactorB`:

$$B_1 = \begin{cases} 1 \text{ if group B1} \\ 0 \text{ if group B2} \\ -1 \text{ if group B3} \end{cases}$$

$$B_2 = \begin{cases} 0 \text{ if group B1} \\ 1 \text{ if group B2} \\ -1 \text{ if group B3} \end{cases}$$

[1] S. A. Glantz and B. K. Slinker, *Primer of Applied Regression & Analysis of Variance* (2nd ed.), NewYork, McGraw-Hill, 2001, pp. 357-358.

Combined, these three dummy variables completely identify all the combinations of `FactorA` and `FactorB`. The multiple regression model is then:

$$\hat{A} = b_0 + b_A A + b_{B_1} B_1 + b_{B_2} B_2 + b_{AB_1} AB_1 + b_{AB_2} AB_2$$

where

- the intercept b_0 is an estimate of the grand mean
- b_A estimates the difference between the grand mean and the mean of `A1`
- $-b_A$ is the difference between the grand mean and the mean of `A2`
- $b_{B_{11}}$ estimates the difference between the grand mean and the mean of `B1`
- $b_{B_{21}}$ estimates the difference between the grand mean and the mean of `B2`
- $-(b_{B_1} + b_{B_2})$ estimates the difference between the grand mean and the mean of `B3`

NMath Stats includes several classes that derive from **LinearRegressionParameter**, and provide access to the dummy variable regression parameters in an ANOVA analysis of variance:

- Class **AnovaRegressionParameter** provides a `SumOfSquares` property that gets the sum of squares due to a parameter.
- Class **AnovaRegressionFactorParam** derives from **AnovaRegressionParameter** and provides the additional properties `FactorName`, which gets the name of the ANOVA factor encoded by a dummy variable, `FactorLevel`, which gets the level of the ANOVA factor encoded by a dummy variable, and `Encoding`, which gets the actual encoding. The encoding is the value the dummy variable assumes when an ANOVA observation is made with the factor at that level.
- Class **AnovaRegressionInteractionParam** also derives from **AnovaRegressionParameter** and provides the additional properties `FactorAName` and `FactorALevel`, which get the name and level of the first factor in the interaction, and `FactorBName` and `FactorBLevel`, which get the name and level of the second factor in the interaction.

Of course, these classes also inherit from **LinearRegressionParameter** methods such as `TStatisticPValue()`, `TStatistic()`, `TStatisticCriticalValue()`, and `ConfidenceInterval()` for testing statistical hypotheses regarding parameter values in a linear regression (Section 42.5).

Instances of these classes cannot be constructed independently. Instead, they are returned by properties and member functions on class **TwoWayAnova**:

- `RegressionInterceptParameter` gets the intercept parameter in the linear regression as an **AnovaRegressionParameter**.

- `GetRegressionFactorParameter()` returns the **AnovaRegressionFactorParam** associated with a specified factor level.
- `RegressionFactorParameters` gets a complete array of **AnovaRegressionFactorParam** estimates for the different factor levels.
- `GetRegressionInteractionParameter()` returns the **AnovaRegressionInteractionParam** associated with the specified interaction.
- `RegressionInteractionParameters` gets a complete array of **AnovaRegressionInteractionParam** estimates for the interactions.

For example, this code gets the regression parameter for `FactorA` at level `A1`:

Code Example – C# ANOVA

```
AnovaRegressionFactorParam param =
  anova.GetRegressionFactorParameter( "FactorA", "A1" );
Console.WriteLine( param );
```

Code Example – VB ANOVA

```
Dim Param As AnovaRegressionFactorParam =
  Anova.GetRegressionFactorParameter("FactorA", "A1")
Console.WriteLine(Param)
```

Example output:

```
Value                            : 4.375
Standard Error                   : 1.63741694728596
t-Statistic for parameter = 0    : 2.67189124141632
p-value for t-Statistic          : 0.0155516784650136
0.05 confidence interval         : [9.3491E-001, 7.8151E+000]
```

Note that method `GetRegressionFactorParameter()` may return `null`. In the effects encoding method, there are $k-1$ dummy variables defined to encode the k levels of a factor. Hence, one level does not have a dummy variable associated with it in the linear regression, and a null reference may be returned even though a valid factor level is specified. Thus:

Code Example – C# ANOVA

```
AnovaRegressionFactorParam param =
  anova.GetRegressionFactorParameter( "FactorA", "A2" );
// param == null
```

Code Example – VB ANOVA

```
Dim Param As AnovaRegressionFactorParam =
  Anova.GetRegressionFactorParameter("FactorA", "A2")
'' param == null
```

Similarly, method `GetRegressionInteractionParameter()` may return `null`. If there are j different levels for the first factor and k different levels for the second factor, there are $(j-1)(k-1)$ dummy variables corresponding to the interactions. Hence, some interactions do not have a dummy variable associated with them in the linear regression, and a null reference may be returned even though valid interactions are specified.

This code prints out the intercept regression parameter, all factor regression parameters, and all interaction regression parameters:

Code Example – C# ANOVA

```
Console.WriteLine( "Intercept" );
Console.WriteLine( anova.RegressionInterceptParameter );
Console.WriteLine();

AnovaRegressionFactorParam[] factorParams =
  anova.RegressionFactorParameters;
for ( int i = 0; i < factorParams.Length; i++ )
{
  Console.WriteLine( factorParams[i].FactorLevel );
  Console.WriteLine( factorParams[i] );
  Console.WriteLine();
}

AnovaRegressionInteractionParam[] interactionParams =
  anova.RegressionInteractionParameters;
for ( int i = 0; i < interactionParams.Length; i++ )
{
  Console.WriteLine( interactionParams[i].FactorALevel +  " x " +
                     interactionParams[i].FactorBLevel );
  Console.WriteLine( interactionParams[i] );
  Console.WriteLine();
}
```

Code Example – VB ANOVA

```
Console.WriteLine("Intercept")
Console.WriteLine(Anova.RegressionInterceptParameter)
Console.WriteLine()

Dim FactorParams As AnovaRegressionFactorParam() =
  Anova.RegressionFactorParameters
For I As Integer = 0 To FactorParams.Length - 1
  Console.WriteLine(FactorParams(I).FactorLevel)
  Console.WriteLine(FactorParams(I))
  Console.WriteLine()
Next

Dim InteractionParams As AnovaRegressionInteractionParam() =
  Anova.RegressionInteractionParameters
```

```
For I As Integer = 0 To InteractionParams.Length - 1
  Console.WriteLine(InteractionParams(I).FactorALevel & " x " &
InteractionParams(I).FactorBLevel)
  Console.WriteLine(InteractionParams(I))
  Console.WriteLine()
Next
```

Example output:

```
Intercept
Value                          : 28.875
Standard Error                 : 1.63741694728596
t-Statistic for parameter = 0: 17.6344821933477
p-value for t-Statistic        : 8.35997937542743E-13
0.05 confidence interval       : [2.5435E+001, 3.2315E+001]

A1
Value                          : 4.375
Standard Error                 : 1.63741694728596
t-Statistic for parameter = 0: 2.67189124141632
p-value for t-Statistic        : 0.0155516784650136
0.05 confidence interval       : [9.3491E-001, 7.8151E+000]

B1
Value                          : 25.5
Standard Error                 : 2.31565725411135
t-Statistic for parameter = 0: 11.0119923640365
p-value for t-Statistic        : 1.98637151171965E-09
0.05 confidence interval       : [2.0635E+001, 3.0365E+001]

B2
Value                          : -7.25
Standard Error                 : 2.31565725411135
t-Statistic for parameter = 0: -3.13086057408882
p-value for t-Statistic        : 0.00577563474636933
0.05 confidence interval       : [-1.2115E+001, -2.3850E+000]

A1 x B1
Value                          : 6
Standard Error                 : 2.31565725411135
t-Statistic for parameter = 0: 2.59105702683213
p-value for t-Statistic        : 0.0184427158909004
0.05 confidence interval       : [1.1350E+000, 1.0865E+001]

A1 x B2
Value                          : -0.999999999999999
Standard Error                 : 2.31565725411135
t-Statistic for parameter = 0: -0.431842837805354
p-value for t-Statistic        : 0.670984111233603
0.05 confidence interval       : [-5.8650E+000, 3.8650E+000]
```

44.4 Two-Way Unbalanced ANOVA

Class **TwoWayAnovaUnbalanced** is the base class for performing a two-way ANOVA when the number of observations in each cell is not the same—an unbalanced design. Three derived classes are provided:

- **TwoWayAnovaTypeI** performs a Type I ANOVA on unbalanced data. Type I, also called *sequential* sum of squares, tests the main effect of factor A, followed by the main effect of factor B after the main effect of A, followed by the interaction effect AB after the main effects.
- **TwoWayAnovaTypeII** performs a Type II ANOVA on unbalanced data. This type tests for each main effect after the other main effect. No significant interaction is assumed.
- **TwoWayAnovaTypeIII** performs a Type III ANOVA on unbalanced data. This type tests for the presence of a main effect after the other main effect and interaction.

Creating UnbalancedTwo-Way ANOVA Objects

Unbalanced two-way ANOVA instances are constructed in the same manner as balanced **TwoWayAnova** objects (Section 44.3). For example, this code groups the numeric data in column `3` of **DataFrame** `df` by factors constructed from columns `0` and `1`:

Code Example – C# Unbalanced ANOVA

```
var type1anova = new TwoWayAnovaTypeI( df, 0, 1, 2 );
```

Code Example – VB Unbalanced ANOVA

```
Dim Type1Anova As New TwoWayAnovaType(DF, 0, 1, 2)
```

Unbalanced Two-Way ANOVA Tables and Regression Parameters

Using an unbalanced two-way ANOVA object is similar to using a balanced **TwoWayAnova** object (Section 44.3). For instance, this code prints the ANOVA table.

Code Example – C# Unbalanced ANOVA

```
Console.WriteLine( type1anova.AnovaTable );
```

Code Example – VB Unbalanced ANOVA

```
Console.WriteLine(Type1Anova.AnovaTable)
```

This code prints the regression parameters.

Code Example – C# Unbalanced ANOVA

```
Console.WriteLine( "FACTOR A ANOVA --------" );
var fa = type1anova.FactorARegressionFactorParameters;
for ( int i = 0; i < fa.Length; i++ )
{
  Console.WriteLine( fa[i] );
  Console.WriteLine();
}

Console.WriteLine( "\nFACTOR B ANOVA --------" );
var fb = type1anova.FactorBRegressionFactorParameters;
for ( int i = 0; i < fb.Length; i++ )
{
  Console.WriteLine( fb[i] );
  Console.WriteLine();
}

Console.WriteLine( "\nINTERACTION FACTOR ANOVA --------" );
var fi = type2anova.InteractionRegressionFactorParameters;
for ( int i = 0; i < fi.Length; i++ )
{
  Console.WriteLine( fi[i] );
  Console.WriteLine();
}
```

Code Example – VB Unbalanced ANOVA

```
Console.WriteLine("FACTOR A ANOVA --------")
Dim FA As AnovaRegressionFactorParam() =
Type1Anova.FactorARegressionFactorParameters
For I As Integer = 0 To FA.Length - 1
  Console.WriteLine(FA(I))
  Console.WriteLine()
Next

Console.WriteLine("\nFACTOR B ANOVA --------")
Dim FB As AnovaRegressionFactorParam() =
Type1Anova.FactorBRegressionFactorParameters
For I As Integer = 0 To FB.Length - 1
  Console.WriteLine(FB(I))
  Console.WriteLine()
Next

Console.WriteLine("\nINTERACTION FACTOR ANOVA --------")
Dim FI As AnovaRegressionInteractionParam() =
Type2Anova.InteractionRegressionFactorParameters
For I As Integer = 0 To FI.Length - 1
  Console.WriteLine(FI(I))
```

```
  Console.WriteLine()
Next
```

44.5 Two-Way Repeated Measures ANOVA

NMath Stats provides two classes for calculating a two-way analysis of variance with repeated measures (RANOVA):

- Class **TwoWayRanova** performs a balanced two-way analysis of variance with repeated measures on one factor.
- Class **TwoWayRanovaTwo** performs a balanced two-way analysis of variance with repeated measures on both factors.

Both classes extend **TwoWayAnova**, and so inherit the methods and properties described in Section 44.3. Like **TwoWayAnova**, both **TwoWayRanova** and **TwoWayRanovaTwo** use multiple linear regression to compute the RANOVA values.

Creating Two-Way RANOVA Objects

Instances of both **TwoWayRanova** and **TwoWayRanovaTwo** are constructed from data in a data frame. Three column indices are specified in the data frame: the column containing the first factor, the column containing the second factor, and the column containing the numeric data. For **TwoWayRanova**, the first factor is the repeated factor; for **TwoWayRanovaTwo**, both factors are repeated.

For example, this code groups the numeric data in column `3` of **DataFrame** `df` by factors constructed from columns `0` and `4`:

Code Example – C# RANOVA

```
var ranova = new TwoWayRanova( df, 0, 4, 3 );
```

Code Example – VB RANOVA

```
Dim Ranova As New TwoWayRanova(DF, 0, 4, 3)
```

The factor constructed from column `0` is the repeated factor. In the following example, both factors are repeated:

Code Example – C# RANOVA

```
var ranova2 = new TwoWayRanovaTwo( df, 0, 4, 3 );
```

Code Example – VB RANOVA

```
Dim Ranova2 As New TwoWayRanovaTwo(DF, 0, 4, 3)
```

NOTE—Both TwoWayRanova and TwoWayRanovaTwo throw an InvalidArgumentException if the data contains missing values (NaNs).

Two-Way RANOVA Tables

Once you've constructed a **TwoWayRanova**, you can display the complete RANOVA table:

Code Example – C# RANOVA

```
var ranova = new TwoWayRanova( df, 0, 4, 3 );
Console.WriteLine( ranova );
```

Code Example – VB RANOVA

```
Dim Ranova As New TwoWayRanova(DF, 0, 4, 3)
Console.WriteLine(Ranova)
```

For instance:

```
Source   Deg of Freedom   SumOfSqu    Mean Square   F         P
FactorA       1           0.2032      0.2032        29.2322   0.0001
Subjects      14          1.7559      0.1254        .         .
FactorB       1           0.0205      0.0205        0.1635    0.6921
Interaction   1           0.0830      0.0830        11.9442   0.0039
Error         14          0.0973      0.0070        .         .
Total         31          2.1599      .             .         .
```

Class **TwoWayRanovaTable** summarizes the information in a traditional two-way RANOVA table with repeated measures on one factor. An instance of **TwoWayRanovaTable** can be obtained from a **TwoWayRanova** object using the `RanovaTable` property. For example:

Code Example – C# RANOVA

```
TwoWayRanovaTable myTable = ranova.RanovaTable;
```

Code Example – VB RANOVA

```
Dim MyTable As TwoWayRanovaTable = Ranova.RanovaTable
```

Class **TwoWayRanovaTable** derives from **TwoWayAnovaTable**, and so inherits the properties described in Section 44.3. In addition, **TwoWayRanovaTable** provides the following properties for accessing the new row in the RANOVA table for repeated measures on one factor:

- `SubjectsDegreesOfFreedom` gets the subjects degrees of freedom.
- `SubjectsSumOfSquares` gets the sum of squares for the subjects.
- `SubjectsMeanSquare` gets the mean square for the subjects.

Similarly, once you've constructed a **TwoWayRanovaTwo**, you can display the RANOVA table:

Code Example – C# RANOVA

```
var ranova2 = new TwoWayRanovaTwo( df, 0, 4, 3 );
Console.WriteLine( ranova2 );
```

Code Example – VB RANOVA

```
Dim Ranova2 As New TwoWayRanovaTwo(DF, 0, 4, 3)
Console.WriteLine(Ranova2)
```

For example:

```
Source   Deg of Freedom   SumOfSq    Mean Square   F         P
FactorA       1            1.4700    1.4700        88.2000   0.0000
FactorB       2           14.5654    7.2827        59.2348   0.0000
Interaction   2            3.3387    1.6694        18.9305   0.0001
A x Subject  14            1.7213    0.1229        .         .
B x Subject   7            0.1167    0.0167        .         .
Error        14            1.2346    0.0882        .         .
Total        47           29.3592    .             .         .
```

An instance of **TwoWayRanovaTwoTable** can be obtained from a **TwoWayRanovaTwo** object using the `RanovaTable` property. For example:

Code Example – C# RANOVA

```
TwoWayRanovaTwoTable myTable = ranova2.RanovaTable;
```

Code Example – VB RANOVA

```
Dim MyTable As TwoWayRanovaTwoTable = Ranova2.RanovaTable
```

Class **TwoWayRanovaTwoTable** also derives from **TwoWayAnovaTable**, and provides the following methods for accessing the additional rows in the RANOVA table with repeated measures on both factors:

- `SubjectInteractionDegreesOfFreedom()` returns the degrees of freedom for the interaction between subjects and the specified factor.
- `SubjectInteractionSumOfSquares()` returns the sum of squares for the interaction between subjects and the specified factor.
- `SubjectInteractionMeanSquare` returns the mean square for the interaction between subjects and the specified factor.

CHAPTER 45.
NON-PARAMETRIC TESTS

Non-parametric (or distribution-free) tests make no assumptions about the probability distributions of the variables being assessed. **NMath Stats** provides classes for several common non-parametric tests:

- Class **OneSampleKSTest** performs a Kolmogorov-Smirnov test of the distribution of one sample.
- Class **TwoSampleKSTest** performs a two-sample Kolmogorov-Smirnov test to compare the distributions of values in two data sets.
- Class **ShapiroWilkTest** tests the null hypothesis that the sample comes from a normally distributed population.
- Class **OneSampleAndersonDarlingTest** performs a Anderson-Darling test of the distribution of one sample.
- Class **KruskalWallisTest** performs a Kruskal-Wallis rank sum test.
- Class **WilcoxonSignedRankTest** performs a Wilcoxon signed-rank test for comparing the means between two paired samples, or repeated measurements on a single sample.

This chapter describes the non-parametric test classes.

See Section 38.9 for Spearman's rank correlation coefficient, commonly known as *Spearman's rho*.

45.1 One Sample Kolmogorov-Smirnov Test

Class **OneSampleKSTest** performs a Kolmogorov-Smirnov test of the distribution of one sample. This class compares the distribution of a given sample to the hypothesized distribution defined by a specified cumulative distribution function (CDF). For each potential value x, the Kolmogorov-Smirnov test compares the proportion of values less than x with the expected number predicted by the specified CDF. The null hypothesis is that the given sample data follow the specified distribution. The alternative hypothesis that the data do not have that distribution.

Sample data can be passed to the constructor as a vector, numeric column in a data frame, or an array of doubles. The hypothesized distribution can be specified either by using an instance of **ProbabilityDistribution** or by supplying a delegate that encapsulates the CDF of the hypothesized distribution. For example, this code creates a **OneSampleKSTest** instance that compares the distribution of `data` to a standard normal distribution:

Code Example – C# Kolmogorov-Smirnov test

```
var norm = new NormalDistribution();
var ks = new OneSampleKSTest( data, norm );
```

Code Example – VB Kolmogorov-Smirnov test

```
Dim Norm As New NormalDistribution()
Dim KS As New OneSampleKSTest(Data, Norm)
```

If `myDist.CDF()` is the CDF for some distribution, this code creates a **OneSampleKSTest** instance that compares the distribution of the data in column `3` of **DataFrame** `df` to the hypothesized distribution:

Code Example – C# Kolmogorov-Smirnov test

```
var ks = new OneSampleKSTest( df[3],
  new Func<double, double>(myDist.CDF) );
```

Code Example – VB Kolmogorov-Smirnov test

```
Dim KS As New OneSampleKSTest(DF(3), New Func(Of Double,
  Double)(AddressOf MyDist.CDF))
```

By default, a **OneSampleKSTest** object performs the Kolmogorov-Smirnov test with $\alpha = 0.01$. A different alpha level can be specified at the time of construction using constructor overloads, or after construction using the provided `Alpha` property.

Once you've constructed and configured a **OneSampleKSTest** object, you can access the various test results using the provided properties:

Code Example – C# Kolmogorov-Smirnov test

```
Console.WriteLine( "statistic = " + test.Statistic );
Console.WriteLine( "p-value = " + test.P );
Console.WriteLine( "alpha = " + test.Alpha );
Console.WriteLine( "reject the null hypothesis? " + test.Reject);
```

Code Example – VB Kolmogorov-Smirnov test

```
Console.WriteLine("statistic = " & Test.Statistic)
Console.WriteLine("p-value = " & Test.P)
Console.WriteLine("alpha = " & Test.Alpha)
Console.WriteLine("reject the null hypothesis? " & Test.Reject)
```

45.2 Two Sample Kolmogorov-Smirnov Test

Class **TwoSampleKSTest** performs a two-sample Kolmogorov-Smirnov test to compare the distributions of values in two data sets. For each potential value x, the Kolmogorov-Smirnov test compares the proportion of values in the first sample less than x with the proportion of values in the second sample less than x. The null hypothesis is that the two samples have the same continuous distribution. The alternative hypothesis is that they have different continuous distributions.

Sample data can be passed to the constructor as vectors, numeric columns in a data frame, or arrays of doubles. Thus:

Code Example – C# Kolmogorov-Smirnov test

```
var ks = new TwoSampleKSTest( data1, data2 );
```

Code Example – VB Kolmogorov-Smirnov test

```
Dim KS As New TwoSampleKSTest(Data1, Data2)
```

By default, a **TwoSampleKSTest** object performs the Kolmogorov-Smirnov test with $\alpha = 0.01$. A different alpha level can be specified at the time of construction using constructor overloads, or after construction using the provided `Alpha` property.

Once you've constructed and configured a **TwoSampleKSTest** object, you can access the various test results using the provided properties:

Code Example – C# Kolmogorov-Smirnov test

```
Console.WriteLine( "statistic = " + test.Statistic );
Console.WriteLine( "p-value = " + test.P );
Console.WriteLine( "alpha = " + test.Alpha );
Console.WriteLine( "reject the null hypothesis? " + test.Reject);
```

Code Example – VB Kolmogorov-Smirnov test

```
Console.WriteLine("statistic = " & Test.Statistic)
Console.WriteLine("p-value = " & Test.P)
Console.WriteLine("alpha = " & Test.Alpha)
Console.WriteLine("reject the null hypothesis? " & Test.Reject)
```

45.3 Shapiro-Wilk Test

Class **ShapiroWilkTest** tests the null hypothesis that a sample comes from a normally distributed population. The sample data provided must be of size between `3` and `5000`. If the size becomes too large, then the test begins to perform poorly.

Code Example – C# Shapiro-Wilk test

```
var data = new DoubleVector(
  "4.6057571 5.0352571 2.5780990 3.8300667 3.9096730 0.3203129 " +
  "0.7165054 9.8681061 3.8967762 9.4639023 6.4092569 2.9835816 " +
  "8.1763496 8.5650066 10.2810477 7.7123572 2.6411587 2.5043797 " +
  "7.5617508 11.2223571" );

double alpha = 0.1;
var test = new ShapiroWilkTest( data, alpha );
```

Code Example – VB Shapiro-Wilk test

```
Dim Data As New DoubleVector(
  "4.6057571 5.0352571 2.5780990 3.8300667 3.9096730 0.3203129 " &
  "0.7165054 9.8681061 3.8967762 9.4639023 6.4092569 2.9835816 " &
  "8.1763496 8.5650066 10.2810477 7.7123572 2.6411587 2.5043797 " &
  "7.5617508 11.2223571")

Dim Alpha As Double = 0.1
Dim Test As New ShapiroWilkTest(Data, Alpha)
```

Once you've constructed and configured a **TwoSampleKSTest** object, you can access the various test results using the provided properties:

Code Example – C# Shapiro-Wilk test

```
Console.WriteLine( "statistic = " + test.Statistic );
Console.WriteLine( "p-value = " + test.P );
Console.WriteLine( “alpha = “ + test.Alpha );
Console.WriteLine( "reject the null hypothesis? " + test.Reject);
```

Code Example – VB Shapiro-Wilk test

```
Console.WriteLine("statistic = " & Test.Statistic)
Console.WriteLine("p-value = " & Test.P)
Console.WriteLine("alpha = " & Test.Alpha)
Console.WriteLine("reject the null hypothesis? " & Test.Reject)
```

45.4 One Sample Anderson-Darling Test

Class **OneSampleAndersonDarlingTest** performs a Anderson-Darling test of the distribution of one sample. An Anderson-Darling test compares the distribution of a given sample to normal distribution function (CDF). The alternative hypothesis that the data do not have a normal distribution.

Code Example – C# Anderson-Darling test

```
int n = 100;
var data = new DoubleVector( n, new RandGenGamma( 23.0 ) );
```

```
var test = new OneSampleAndersonDarlingTest( data );

Console.WriteLine( "statistic = " + test.Statistic );
Console.WriteLine( "p-value = " + test.P );
Console.WriteLine( "alpha = " + test.Alpha );
Console.WriteLine( "reject the null hypothesis? " + test.Reject);
```

Code Example – VB Anderson-Darling test

```
Dim N As Integer = 100
Dim Data As New DoubleVector(N, New RandGenGamma(23.0))
Dim Test As New OneSampleAndersonDarlingTest(Data)

Console.WriteLine("statistic = " & Test.Statistic)
Console.WriteLine("p-value = " & Test.P)
Console.WriteLine("alpha = " & Test.Alpha)
Console.WriteLine("reject the null hypothesis? " & Test.Reject)
```

45.5 Kruskal-Wallis Test

Class **KruskalWallisTest** performs a Kruskal-Wallis rank sum test. The Kruskal-Wallis test is a non-parametric test for equality of population medians among groups. It is a non-parametric version of the classical one-way ANOVA. The interface for **KruskalWallisTest** is nearly identical to **OneWayAnova**.

Creating Kruskal-Wallis Objects

A **KruskalWallisTest** instance is constructed from numeric data organized into different groups. The groups need not contain the same number of observations. For example, this code constructs a **KruskalWallisTest** from an array of **DoubleVector** objects. Each vector in the array contains data for a single group:

Code Example – C# Kruskal-Wallis test

```
var a =
  new DoubleVector(6.4, 6.8, 7.2, 8.3, 8.4, 9.1, 9.4, 9.7);
var b =
  new DoubleVector(2.5, 3.7, 4.9, 5.4, 5.9, 8.1, 8.2);
var c =
  new DoubleVector(1.3, 4.1, 4.9, 5.2, 5.5, 8.2);

var data_ = new DoubleVector[] { a, b, c };

var test = new KruskalWallisTest( data_);
```

Code Example – VB Kruskal-Wallis test

```
Dim A As New DoubleVector(6.4, 6.8, 7.2, 8.3, 8.4, 9.1, 9.4, 9.7)
Dim B As New DoubleVector(2.5, 3.7, 4.9, 5.4, 5.9, 8.1, 8.2)
Dim C As New DoubleVector(1.3, 4.1, 4.9, 5.2, 5.5, 8.2)

Dim Data_() As DoubleVector = {A, B, C}

Dim Test As New KruskalWallisTest(Data )
```

An optional boolean parameter may also be supplied to the constructor. If `true`, a standard correction for ties is applied.

Code Example – C# Kruskal-Wallis test

```
bool correct_for_ties = true;
var test = new KruskalWallisTest( data, correct_for_ties_);
```

Code Example – VB Kruskal-Wallis test

```
Dim CorrectForTies As Boolean = True
Dim Test As New KruskalWallisTest(Data, CorrectForTies)
```

This correction usually makes little difference in the value of the test statistic, unless there are a large number of ties.

This code constructs a **KruskalWallisTest** from a data frame `df`:

Code Example – C# Kruskal-Wallis test

```
var test = new KruskalWallisTest( df, 1, 3 );
```

Code Example – VB Kruskal-Wallis test

```
Dim Test As New KruskalWallisTest(DF, 1, 3)
```

Two column indices are also provided: a *group* column and a *data* column. A **Factor** is constructed from the group column using the **DataFrame** method `GetFactor()`, which creates a sorted array of the unique values. The specified data column must be of type **DFNumericColumn**.

Lastly, you can also construct a **KruskalWallisTest** from a **DoubleMatrix**:

Code Example – C# Kruskal-Wallis test

```
var data = new DoubleMatrix( "6 x 5 [ 24 14 11 7 19
                                      15 7 9 7 24
                                      21 12 7 7 19
                                      27 17 13 12 15
                                      33 14 12 12 10
                                      23 16 18 18 20 ]" );

bool correct_for_ties = true;
var test = new KruskalWallisTest( data, correct_for_ties );
```

Code Example – VB Kruskal-Wallis test

```
Dim Data As New DoubleMatrix("6 x 5 [ 24 14 11 7 19
                                      15 7 9 7 24
                                      21 12 7 7 19
                                      27 17 13 12 15
                                      33 14 12 12 10
                                      23 16 18 18 20 ]")

Dim CorrectForTies As Boolean = True
Dim Test As New KruskalWallisTest(Data, CorrectForTies)
```

Each column in the given matrix contains the data for a group. If your groups have different numbers of observations, you must pad the columns with `Double.NaN` values until they are all the same length, because a **DoubleMatrix** must be rectangular. Alternatively, use one of the other constructors described above.

The Kruskal-Wallis Table

Once you've constructed a **KruskalWallisTest**, you can display the complete results table:

Code Example – C# Kruskal-Wallis test

```
Console.WriteLine( test );
```

Code Example – VB Kruskal-Wallis test

```
Console.WriteLine(Test)
```

For example:

```
Source      Deg of Freedom    Sum Of Sq  Mean Sq    Chi-sq    P
Between groups     2          13.5000   6.7500      0.7714    0.6800
Within groups      11         214       19.4545     .         .
Total              13         227.5000  .           .         .
```

Class **KruskalWallisTable** is provided for summarizing the information in the results table. Class **KruskalWallisTable** derives from **DataFrame**. An instance of **KruskalWallisTable** can be obtained from a **KruskalWallisTest** object using the `Table` property. For example:

Code Example – C# Kruskal-Wallis test

```
KruskalWallisTable table = test.Table;
```

Code Example – VB Kruskal-Wallis test

```
Dim Table As KruskalWallisTable = Test.Table
```

Class **KruskalWallisTable** provides the following read-only properties for accessing individual elements in the results table:

- `DegreesOfFreedomBetween` gets the between-groups degrees of freedom.
- `DegreesOfFreedomWithin` gets the within-groups degrees of freedom.
- `DegreesOfFreedomTotal` gets the total degrees of freedom.
- `SumOfSquaresBetween` gets the between-groups sum of squares.
- `SumOfSquaresWithin` gets the within-groups sum of squares.
- `SumOfSquaresTotal` gets the total sum of squares.
- `MeanSquareBetween` gets the between-groups mean square. The between-groups mean square is the between-groups sum of squares divided by the between-groups degrees of freedom.
- `MeanSquareWithin` gets the within-group mean square. The within-groups mean square is the within-group sum of squares divided by the within-group degrees of freedom.
- `MeanSquareTotal` gets the total mean square. The total mean square is the total sum of squares divided by the total degrees of freedom.
- `Statistic` gets the test statistic.
- `PValue` gets the p-value for the test statistic.

Ranks, Grand Mean Ranks, Group Means Ranks, and Group Sizes

Class **KruskalWallisTest** provides properties and methods for retrieving the ranks, grand mean ranks, group means ranks, and group sizes:

- `Ranks` gets an array of vectors containing the ranks of the data.
- `GrandMeanRank` gets the grand mean rank of the data. The grand mean rank is the mean of all of the data ranks.
- `GroupMeanRanks` gets a vector of group mean ranks.
- `GroupSizes` gets an array of group sizes.
- `GroupNames` gets an array of group names. If the test was constructed from a data frame using a grouping column, the group names are the sorted, unique **Factor** levels created from the column values. If the test object was constructed from a matrix or an array of vectors, the group names are simply `Group_0, Group_1...Group_n`.

- `GetGroupRanks()` returns the ranks for a specified group, identified either by group name or group number (a zero-based index into the `Ranks` array).
- `GetGroupMeanRank()` returns the mean rank for a specified group, identified either by group name or group number (a zero-based index into the `GroupMeanRanks` vector).
- `GetGroupSize()` returns the mean for a specified group, identified either by group name or group number (a zero-based index into the `GroupSizes` array).

For example, if a **KruskalWallisTest** is constructed from a matrix, this code returns the mean rank for the group in the third column of the matrix:

Code Example – C# Kruskal-Wallis test

```
double mean = test.GetGroupMeanRank( 2 );
```

Code Example – VB Kruskal-Wallis test

```
Dim Mean As Double = Test.GetGroupMeanRank(2)
```

If a **KruskalWallisTest** is constructed from a data frame using a grouping column with values `male` and `female`, this code returns the mean rank for the `male` group:

Code Example – C# Kruskal-Wallis test

```
double maleMean = test.GetGroupMeanRank( “male” );
```

Code Example – VB Kruskal-Wallis test

```
Dim MaleMean As Double = Test.GetGroupMeanRank("male")
```

Critical Value of the Test Statistic

Class **KruskalWallisTest** provides the convenience function `StatisticCriticalValue()` which computes the critical value for the test statistic at a given significance level. Thus:

Code Example – C# Kruskal-Wallis test

```
double alpha = 0.05;
double critVal = test.StatisticCriticalValue( alpha );
```

Code Example – VB Kruskal-Wallis test

```
Dim Alpha As Double = 0.05
Dim CritVal As Double = Test.StatisticCriticalValue(Alpha)
```

Updating Kruskal-Wallis Test Objects

Method `SetData()` updates an entire test object with new data. As with the class constructors (see above), you can supply data as an array of group vectors, a matrix, or as a data frame. For instance, this code updates a test with data from **DataFrame** `df`, using column `2` as the group column and column `5` as the data column:

Code Example – C# Kruskal-Wallis test

```
test.SetData( df, 2, 5 );
```

Code Example – VB Kruskal-Wallis test

```
Test.SetData(DF, 2, 5)
```

45.6 Wilcoxon Signed-Rank Test

The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test for comparing the means between two paired samples, or repeated measurements on a single sample. It can be used as an alternative to **TwoSamplePairedTTest** when the population cannot be assumed to be normally distributed.

Class **WilcoxonSignedRankTest** tests if two paired sets of observed values differ from each other in a significant way. The null hypothesis is that the distribution x - y is symmetric about `0`.

Creating Wilcoxon Signed-Rank Objects

A **WilcoxonSignedRankTest** instance is constructed from paired vectors of sample data.

Code Example – C# Wilcoxon signed-rank test

```
var a = new DoubleVector( 78, 24, 64, 45, 64, 52, 30, 50, 64, 50,
  78, 22, 84, 40, 90, 72 );
var b = new DoubleVector( 78, 24, 62, 48, 68, 56, 25, 44, 56, 40,
  68, 36, 68, 20, 58, 32 );

double alpha = 0.05;
var type = HypothesisType.TwoSided;
bool exactPValue = false;
var test =
  new WilcoxonSignedRankTest( a, b, alpha, type, exactPValue  );
```

Code Example – VB Wilcoxon signed-rank test

TODO

Note that paired observations where either value is missing, or where the difference between values is zero, are ignored. In the example above, a normal approximation is used to compute p-value. For $n > 10$, the sampling distribution of the test statistic converges to a normal distribution. For smaller sample sizes, an exact p-value can be calculated by enumerating all possible combinations of the test statistic given n.

Code Example – C# Wilcoxon signed-rank test

```
var x = new DoubleVector( 1.83, 0.50, 1.62, 2.48, 1.68, 1.88, 1.55,
  3.06, 1.30 );
var y = new DoubleVector( 0.878, 0.647, 0.598, 2.050, 1.060, 1.290,
  1.060, 3.140, 1.290 );

alpha = 0.01;
exactPValue = true;
test =
  new WilcoxonSignedRankTest( x, y, alpha, type, exactPValue );
```

Code Example – VB Wilcoxon signed-rank test

TODO

An **InvalidArgumentException** is raised if the given data contains zero valid pairs (valid pairs are non-NaN and unequal), or if an exact p-value is specified for $n > 30$.

CHAPTER 46.
MULTIVARIATE TECHNIQUES

Multivariate statistical analysis techniques are useful when you need a concise understanding of large amounts of data. **NMath Stats** provides classes for dimension reduction using *principal component analysis* or *factor analysis*, and case reduction using *hierarchical cluster analysis* and *k-means clustering*.

This chapter describes the multivariate statistical analysis classes.

46.1 Principal Component Analysis

Principal component analysis (PCA) 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. In **NMath Stats**, classes **DoublePCA** and **FloatPCA** perform principal component analyses.

Creating Principal Component Analyses

A **DoublePCA** or **FloatPCA** instance is constructed from a matrix or a dataframe containing numeric data. Each column represents a variable, and each row represents an observation:

Code Example – C# principal component analysis (PCA)

```
var pca = new DoublePCA( data );
```

Code Example – VB principal component analysis (PCA)

```
Dim PCA As New DoublePCA(Data)
```

The data may optionally be zero-centered and scaled to have unit variance:

Code Example – C# principal component analysis (PCA)

```
bool center = true;
bool scale = true;
var pca = new DoublePCA( data, center, scale );
```

Code Example – VB principal component analysis (PCA)

```
Dim Center As Boolean = True
```

```
Dim Scale As Boolean = True
Dim PCA As New DoublePCA(Data, Center, Scale)
```

By default, variables are centered but not scaled.

After construction, you can retrieve information about the data set using the provided read-only properties:

- `Data` gets the data matrix. If centering or scaling were specified at construction time, the returned matrix may not match the original data.
- `NumberOfObservations` gets the number of observations in the data matrix.
- `NumberOfVariables` gets the number of variables in the data matrix.
- `IsCentered` returns `true` if the data supplied at construction time was shifted to be zero-centered.
- `IsScaled` returns `true` if the data supplied at construction time was scaled to have unit variance.
- `Means` gets the column means of the data matrix. If centering is specified, the column means are substracted from the column values before analysis takes place.
- `Norms` gets the column norms (1-norm). If scaling is specified, column values are scaled to have unit variance before analysis by dividing by the column norm.

Principal Component Analysis Results

The `Loadings` property gets the complete loading matrix. Each column in the loading matrix is a principal component. 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.

Code Example – C# principal component analysis (PCA)

```
Console.WriteLine( "Loading Martrix = " + pca.Loadings );
```

Code Example – VB principal component analysis (PCA)

```
Console.WriteLine("Loading Matrix = " & PCA.Loadings)
```

The provided indexer also gets a specified principal component, referenced by zero-based index. For example:

Code Example – C# principal component analysis (PCA)

```
Console.WriteLine( "First principal component = " + pca[0] );
```

```
Console.WriteLine( "Second principal component = " + pca[1] );
```

Code Example – VB principal component analysis (PCA)

```
Console.WriteLine("First principal component = " & PCA(0))
Console.WriteLine("Second principal component = " & PCA(1))
```

The `VarianceProportions` property gets an ordered vector containing the proportion of the total variance accounted for by each principal component. `CumulativeVarianceProportions` gets the cumulative variance proportions. Thus:

Code Example – C# principal component analysis (PCA)

```
Console.WriteLine( "Variance Proportions = " +
                   pca.VarianceProportions );
Console.WriteLine( "Cumulative Variance Proportions = " +
                   pca.CumulativeVarianceProportions );
```

Code Example – VB principal component analysis (PCA)

```
Console.WriteLine("Variance Proportions = " &
  PCA.VarianceProportions)
Console.WriteLine("Cumulative Variance Proportions = " &
  PCA.CumulativeVarianceProportions)
```

The `Threshold()` method calculates the number of principal components required to account for a given proportion of the total variance:

Code Example – C# principal component analysis (PCA)

```
Console.WriteLine( "PCs that account for 99% of the variance = " +
                   pca.Threshold( .99 ) );
```

Code Example – VB principal component analysis (PCA)

```
Console.WriteLine("PCs that account for 99% of the variance = " &
  PCA.Threshold(0.99))
```

The `StandardDeviations` property gets the standard deviations of the principal components. `Eigenvalues` gets the eigenvalues of the covariance/correlation matrix, though the calculation is actually performed using the singular values of the data matrix. The eigenvalues of the covariance/correlation matrix are equal to the squares of the standard deviations of the principal components.

Lastly, the `Scores` property gets the score matrix. The scores are the data formed by transforming the original data into the space of the principal components:

Code Example – C# principal component analysis (PCA)

```
Console.WriteLine( "Scores = " + pca.Scores );
```

Code Example – VB principal component analysis (PCA)

```
Console.WriteLine("Scores = " & PCA.Scores)
```

This code displays the data in the minimal synthetic dimensions required to account for 99% of the variance:

Code Example – C# principal component analysis (PCA)

```
Slice rowSlice = Slice.All;
var colSlice = new Slice( 0, pca.Threshold( .99 ) );
Console.WriteLine( pca.Scores[ rowSlice, colSlice ] );
```

Code Example – VB principal component analysis (PCA)

```
Dim RowSlice As Slice = Slice.All
Dim ColSlice As New Slice(0, PCA.Threshold(0.99))
Console.WriteLine(PCA.Scores(RowSlice, ColSlice))
```

46.2 Factor Analysis

Factor analysis describes the variability among observed, correlated variables in terms of a potentially lower number of unobserved variables, called *factors*.

In general, factor analysis consists of two steps:

- In the *extraction* step, factors are extracted from the data.

  In **NMath Stats**, **IFactorExtraction** is the interface for factor extraction algorithms. Class **PCFactorExtraction** implements the principle component (PC) algorithm for factor extraction.

- In the *rotation* step, the factors are rotated in order to maximize the relationship between the variables and the factors.

  In **NMath Stats**, **IFactorRotation** is the interface for factor rotation algorithms. Class **VarimaxRotation** computes the varimax rotation of the factors. Factors are rotated to maximize the sum of the variances of the squared loadings. Kaiser normalization is optionally performed. Class **NoRotation** can be used when no rotation is desired.

Creating Factor Analyses

NMath Stats provides three classes for performing factor analysis:

- **FactorAnalysisCorrelation** performs a factor analysis on given case data by forming the correlation matrix for the variables.

- **FactorAnalysisCovariance** performs a factor analysis on given case data using the covariance matrix.

- **DoubleFactorAnalysis** performs a factor analysis on a symmetric matrix of data, assumed to be either a correlation or covariance matrix, if you don't have access to the original case data.

When case data is used, the data should provided in matrix form—the variable values in columns and each row representing a case.

All factor analysis are templatized on the extraction and rotation algorithm to use. For example:

Code Example – C# factor analysis

```
var fa = new FactorAnalysisCorrelation<PCFactorExtraction,
  VarimaxRotation>( data );
```

Code Example – VB factor analysis

```
Dim FA As New FactorAnalysisCorrelation(Of PCFactorExtraction,
  VarimaxRotation)(Data)
```

For greater control, construct the extraction and rotation objects explicitly. For example, a **PCFactorExtraction** instance can be constructed from a delegate for determining the number of factors to extract. The type of this argument is `Func<DoubleVector, DoubleMatrix, int>`. It takes as arguments the vector of eigenvalues and the matrix of eigenvectors, and returns the number of factors to extract. Class **NumberOfFactors** contains static methods for creating functors for several common strategies. This code extracts factors whose eigenvalues are greater than `1.2` times the mean of the eigenvalues:

Code Example – C# factor analysis

```
var factorExtraction = new PCFactorExtraction(
  NumberOfFactors.EigenvaluesGreaterThanMean( 1.2 ) );
```

Code Example – VB factor analysis

```
Dim FactorExtraction As New PCFactorExtraction(
  NumberOfFactors.EigenvaluesGreaterThanMean(1.2))
```

The following code constructs a **VarimaxRotation** instance with a specified tolerance. Iteration stops when the relative change in the sum of the singular values is less than this number. We also specify that we do not want Kaiser normalization to be performed.

Code Example – C# factor analysis

```
var factorRotation = new VarimaxRotation
{
  Tolerance = 1e-6,
  Normalize = false
};
```

Code Example – VB factor analysis

```
Dim FactorRotation As New VarimaxRotation()
FactorRotation.Tolerance = 0.000001
FactorRotation.Normalize = False
```

Once you've constructed your extraction and rotation objects, you can construct the factor analysis instance:

Code Example – C# factor analysis

```
var fa = new FactorAnalysisCovariance<PCFactorExtraction,
  VarimaxRotation>( data, BiasType.Biased, factorExtraction,
    factorRotation );
```

Code Example – VB factor analysis

```
Dim FA As New FactorAnalysisCovariance(Of PCFactorExtraction,
  VarimaxRotation)(Data, BiasType.Biased, FactorExtraction,
  FactorRotation)
```

Factor Analysis Results

Once you've constructed a factor analysis instance, you can access the results using the following properties:

- `NumberOfFactors` get the number of factors extracted.
- `Factors` gets the extracted factors. Each column of the matrix is a factor.
- `RotatedFactors` gets the rotated factors. Each column of the matrix is a factor.
- `VarianceProportions` gets a vector of proportion of variance explained by each factor.
- `CumulativeVarianceProportions` gets the cumulative variance proportions.
- `ExtractedCommunalities` get the proportion of each variable's variance that can be explained by the extracted factors jointly.
- `InitialCommunalities` get the proportion of each variable's variance that can be explained by the factors jointly.
- `SumOfSquaredLoadings` gets the sum of squared loadings for each extracted factor.
- `RotatedSumOfSquaredLoadings` gets the sum of squared loadings for each rotated extracted factor.

For instance:

Code Example – C# factor analysis

```
DoubleVector extractedCommunalities = fa.ExtractedCommunalities;
for ( int i = 0; i < data.Cols; i++ )
{
  Console.WriteLine( "{0}\t{1}", data[i].Name,
    extractedCommunalities[i] );
}
Console.WriteLine();

for ( int i = 0; i < fa.VarianceProportions.Length; i++ )
{
  double varProportion = fa.VarianceProportions[i] * 100.0;
  double cummlativeVarProportion =
    fa.CumulativeVarianceProportions[i] * 100.0;
  double eigenValue = fa.FactorExtraction.Eigenvalues[i];
  Console.WriteLine( "{0}\t\t{1}\t{2}\t\t{3}", i, eigenValue,
    varProportion, cummlativeVarProportion );
}
Console.WriteLine();

double eigenValueSum =
  NMathFunctions.Sum( fa.FactorExtraction.Eigenvalues );
DoubleVector RotatedSSLoadingsVarianceProportions =
  fa.RotatedSumOfSquaredLoadings / eigenValueSum;
Console.WriteLine(
  "\nRotated Extraction Sums of Squared Loadings - " );
Console.WriteLine( "factor\tTotal\t% of Variance\tCummlative %" );
Console.WriteLine(
  "----------------------------------------------------" );
double cummlative = 0;

for ( int i = 0; i < fa.NumberOfFactors; i++ )
{
  double varProportion =
    RotatedSSLoadingsVarianceProportions[i] * 100.0;
  cummlative += RotatedSSLoadingsVarianceProportions[i];
  double cummlativeVarProportion = cummlative * 100.0;
  double sumSquaredLoading = fa.RotatedSumOfSquaredLoadings[i];
  Console.WriteLine( "{0}\t\t{1}\t{2}\t\t{3}", i,
    sumSquaredLoading, varProportion, cummlativeVarProportion );
}
Console.WriteLine();
```

```
DoubleMatrix rotatedComponentMatrix = fa.RotatedFactors;
for ( int i = 0; i < data.Cols; i++ )
{
  var formatString = "{0}\t\t{1}\t{2}\t{3}";
  double comp0 = rotatedComponentMatrix.Row( i )[0];
  double comp1 = rotatedComponentMatrix.Row( i )[1];
  double comp2 = rotatedComponentMatrix.Row( i )[2];
  Console.WriteLine( "{0}\t{1}\t{2}\t{3}", data[i].Name,
    comp0, comp1, comp2 );
}
```

Code Example – VB factor analysis

```
Dim ExtractedCommunalities As DoubleVector =
  FA.ExtractedCommunalities
For I As Integer = 0 To Data.Cols - 1
  Console.WriteLine("{0}\t{1}", Data(I).Name,
    ExtractedCommunalities(I))
Next
Console.WriteLine()

For I As Integer = 0 To FA.VarianceProportions.Length - 1
  Dim VarProportion As Double = FA.VarianceProportions(I) * 100.0
  Dim CumulativeVarProportion = FA.CumulativeVarianceProportions(I)
    * 100.0
  Dim EigenValue As Double = FA.FactorExtraction.Eigenvalues(I)
  Console.WriteLine("{0}\t\t{1}\t{2}\t\t{3}", I, EigenValue,
    VarProportion, CumulativeVarProportion)
Next
Console.WriteLine()

Dim EigenValueSum As Double =
  NMathFunctions.Sum(FA.FactorExtraction.Eigenvalues)
Dim RotatedSSLoadingsVarianceProportions As DoubleVector =

  FA.RotatedSumOfSquaredLoadings / EigenValueSum
Console.WriteLine(
"\nRotated Extraction Sums of Squared Loadings - ")
Console.WriteLine("factor\tTotal\t% of Variance\tCumulative %")
Console.WriteLine(
"-----------------------------------------------------")
Dim Cumulative As Double = 0

For I As Integer = 0 To FA.NumberOfFactors - 1
  Dim VarProportion As Double =
    RotatedSSLoadingsVarianceProportions(I) * 100.0;
  Cumulative += RotatedSSLoadingsVarianceProportions(I)
  Dim CumulativeVarProportion As Double = Cumulative * 100.0
  Dim SumSquaredLoading As Double =
    FA.RotatedSumOfSquaredLoadings(I)
  Console.WriteLine("{0}\t\t{1}\t{2}\t\t{3}", I, SumSquaredLoading,
```

```
    VarProportion, CumulativeVarProportion)
Next

Console.WriteLine()

Dim RotatedComponentMatrix As DoubleMatrix = FA.RotatedFactors
For I As Integer = 0 To Data.Cols - 1
  Dim formatString As String = "{0}\t\t{1}\t{2}\t{3}"
  Dim Comp0 As Double = RotatedComponentMatrix.Row(I)(0)
  Dim Comp1 As Double = RotatedComponentMatrix.Row(I)(1)
  Dim Comp2 As Double = RotatedComponentMatrix.Row(I)(2)
  Console.WriteLine("{0}\t{1}\t{2}\t{3}", Data(I).Name, Comp0,
    Comp1, Comp2)
Next
```

Factor Scores

The case data values for new factor variables are contained in the *factor scores* matrix. The score for a given factor is a linear combination of all of the measures, weighted by the corresponding factor loading.

There are different algorithms for producing the factors scores. The `FactorScores()` method can be passed an object implementing the **IFactorScores** interface, specifying the algorithm to be used. If no argument is passed, the regression algorithm for computing factor scores is used, implemented in class **RegressionFactorScores**.

For example, this code print the factor scores for the first three cases. Data is normalized.

Code Example – C# factor analysis

```
var rowSlice = new Slice( 0, 3 );
Console.WriteLine(
  fa.FactorScores()[rowSlice, Slice.All].ToTabDelimited() );
```

Code Example – VB factor analysis

```
Dim RowSlice As New Slice(0, 3)
Console.WriteLine(FA.FactorScores()(RowSlice,
  Slice.All).ToTabDelimited())
```

Factor scores are a linear combination of the original variable values. The coefficients used for the linear combination are found in the *factor score coefficients matrix*. This matrix may be obtained from the `FactorScoreCoefficients()` method on the factor analysis class. Like factor scores, the algorithm to use may be specified by passing an object implementing the **IFactorScores** interface to this method. By default, the regression algorithm is used.

The factor score coefficients can be used to compute scores for novel case data. For instance:

Code Example – C# factor analysis

```
DoubleMatrix scoreCoefficients = fa.FactorScoreCoefficients();
var newCaseData = new DoubleMatrix(
  "2x10 [0.0 38.9 3.8 196.0 115.4 71.9 177.0 3.972 17.5 27.8  " +
        "1.0 46.0 2.5 220.0 101.6 73.4 168.6 3.75  19.0 20.0]" );
Console.WriteLine(
  NMathFunctions.Product( newCaseData, scoreCoefficients ) );
```

Code Example – VB factor analysis

```
Dim ScoreCoefficients As DoubleMatrix =
  FA.FactorScoreCoefficients()
Dim NewCaseData As New DoubleMatrix(
  "2x10 [0.0 38.9 3.8 196.0 115.4 71.9 177.0 3.972 17.5 27.8  " &
  "1.0 46.0 2.5 220.0 101.6 73.4 168.6 3.75  19.0 20.0]")
Console.WriteLine(NMathFunctions.Product(NewCaseData,
  ScoreCoefficients))
```

46.3 Hierarchical Cluster Analysis

Cluster analysis detects natural groupings in data. In hierarchical cluster analysis, 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. In **NMath Stats**, class **ClusterAnalysis** performs hierarchical cluster analyses.

Distance Functions

During clustering, the distance between individual objects is computed using a distance function. The distance function is encapsulated in a `Distance.Function` delegate, which takes two vectors and returns a measure of the distance (similarity) between them:

Code Example – C# hierarchical cluster analysis

```
public delegate double Function( DoubleVector data1,
                                 DoubleVector data2 );
```

Code Example – VB hierarchical cluster analysis

```
Delegate Function(Data1 As DoubleVector, Data2 As DoubleVector) As
  Double
```

Delegates are provided as static variables on class **Distance** for many common distance functions:

- `Distance.EuclideanFunction` computes the Euclidean distance between two data vectors (2 norm):

  $$d_{xy} = \sqrt{\sum (x_i - y_i)^2}$$

  Euclidean distance is simply the geometric distance in the multidimensional space.

- `Distance.SquaredEuclideanFunction` computes the squared Euclidean distance between two vectors:

  $$d_{xy} = \sum (x_i - y_i)^2$$

  Squaring the simple Euclidean distance places progressively greater weight on objects that are further apart.

- `Distance.CityBlockFunction` computes the city-block (Manhattan) distance between two vectors (1 norm):

  $$d_{xy} = \sum |x_i - y_i|$$

  In most cases, the city-block distance measure yields results similar to the simple Euclidean distance. Note, however, that the effect of outliers is dampened, since they are not squared.

- `Distance.MaximumFunction` computes the maximum (Chebychev) distance between two vectors:

  $$d_{xy} = \text{maximum}|x_i - y_i|$$

  This distance measure may be appropriate in cases when you want to define two objects as different if they differ on any one of the dimensions.

- `Distance.PowerFunction( double p, double r )` computes the power distance between two vectors:

  $$d_{xy} = (\sum |x_i - y_i|^p)^{1/r}$$

  where `p` and `r` are user-defined parameters. Parameter `p` controls the progressive weight that is placed on differences on individual dimensions;

parameter `r` controls the progressive weight that is placed on larger differences between objects. Appropriate selections of `p` and `r` yield Euclidean, squared Euclidean, Minkowski, city-block, and many other distance metrics. For example, if `p` and `r` are equal to `2`, the power distance is equal to the Euclidean distance.

All provided distance functions allow missing values. Pairs of elements are excluded from the distance measure when their comparison returns `NaN`. If all pairs are excluded, `NaN` is returned for the distance measure.

You can also define your own `Distance.Function` delegate and use it to cluster your data. For example, if you have function `MyDistance()` that computes the distance between two vectors:

Code Example – C# hierarchical cluster analysis

```
public double MyDistance( DoubleVector x, DoubleVector y );
```

Code Example – VB hierarchical cluster analysis

```
Public Function MyDistance(X As DoubleVector, Y As DoubleVector) As
  Double
```

You can define a `Distance.Function` delegate like so:

Code Example – C# hierarchical cluster analysis

```
var MyDistanceFunction = new Distance.Function( MyDistance );
```

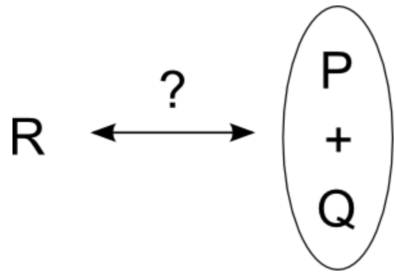
Code Example – VB hierarchical cluster analysis

```
Dim MyDistanceFunction As New Distance.Function(AddressOf
  MyDistance)
```

Linkage Functions

During clustering, the distances between clusters of objects are computed using a linkage function. The linkage function is encapsulated in a `Linkage.Function` delegate. When two groups `P` and `Q` are united, a linkage function computes the distance between the new combined group `P + Q` and another group `R`.

Figure 6 – Computing the distance between clusters using a linkage function



The parameters to the `Linkage.Function`—which may not necessarily all be used to calculate the result—are the distance between `R` and `P`, the distance between `R` and `Q`, the distance between `P` and `Q`, and the sizes (`n`) of all three groups:

Code Example – C# hierarchical cluster analysis

```
public delegate double Function( double Drp, double Drq,
  double Dpq, double Nr, double Np, double Nq );
```

Code Example – VB hierarchical cluster analysis

```
Delegate Function(DRP As Double, DRQ As Double,
  DPQ As Double, NR As Double, NP As Double, NQ As Double) As
  Double
```

Delegates are provided as static variables on class **Linkage** for many common linkage functions:

- `Linkage.SingleFunction` computes the distance between two clusters as the distance of the two closest objects (nearest neighbors) in the clusters. Adopting a friends-of-friends clustering strategy closely related to the minimal spanning tree, the single linkage method tends to result in long chains of clusters.

- `Linkage.CompleteFunction` computes the distance between two clusters as the greatest distance between any two objects in the different clusters (furthest neighbors). The complete linkage method tends to work well in cases where objects form naturally distinct clumps.

- `Linkage.UnweightedAverageFunction` computes the distance between two clusters as the average distance between all pairs of objects in the two different clusters. This method is sometimes referred to as *unweighted pair-group method using arithmetic averages*, and abbreviated UPGMA.

- `Linkage.WeightedAverageFunction` computes the distance between two clusters as the average distance between all pairs of objects in the two different clusters, using the size of each cluster as a weighting factor. This method is sometimes referred to as *weighted pair-group method using arithmetic averages*, and abbreviated WPGMA.

- `Linkage.CentroidFunction` computes the distance between two clusters as the difference between centroids. The centroid of a cluster is the average point in the multidimensional space. The centroid method is sometimes referred to as *unweighted pair-group method using the centroid average*, and abbreviated UPGMC.

- `Linkage.MedianFunction` computes the distance between two clusters as the difference between centroids, using the size of each cluster as a weighting factor. This is sometimes referred to as *weighted pair-group method using the centroid average*, and abbreviated WPGMC.

- `Linkage.WardFunction` computes the distance between two clusters using Ward's method. Ward's method uses an analysis of variance approach to evaluate the distances between clusters. The smaller the increase in the total within-group sum of squares as a result of joining two clusters, the closer they are. The within-group sum of squares of a cluster is defined as the sum of the squares of the distance between all objects in the cluster and the centroid of the cluster. Ward's method tends to produce compact groups of well-distributed size.

You can also define your own `Linkage.Function` delegate and use it to cluster your data. For example, if you have function `MyLinkage()` that computes the distance between two clusters:

Code Example – C# hierarchical cluster analysis

```
public double MyLinkage( double Drp, double Drq, double Dpq,
                         double Nr, double Np, double Nq );
```

Code Example – VB hierarchical cluster analysis

```
Public Function MyLinkage(DRP As Double, DRQ As Double, DPQ As
  Double, NR As Double, NP As Double, NQ As Double) As Double
```

You can define a `Linkage.Function` delegate like so:

Code Example – C# hierarchical cluster analysis

```
var MyLinkageFunction = new Linkage.Function( MyLinkage );
```

Code Example – VB hierarchical cluster analysis

```
Dim MyLinkageFunction As New Linkage.Function(AddressOf MyLinkage)
```

Creating Cluster Analyses

A **ClusterAnalysis** instance is constructed from a matrix or a dataframe containing numeric data. Each row in the data set represents an object to be clustered.

Code Example – C# hierarchical cluster analysis

```
var ca = new ClusterAnalysis( data );
```

Code Example – VB hierarchical cluster analysis

```
Dim CA As New ClusterAnalysis(Data)
```

The current default distance and linkage delegates are used. The default distance and linkage delegates are `Distance.EuclideanFunction` and `Linkage.SingleFunction`, unless the defaults have been changed using the static `DefaultDistanceFunction` and `DefaultLinkageFunction` properties. For example:

Code Example – C# hierarchical cluster analysis

```
ClusterAnalysis.DefaultDistanceFunction = Distance.MaximumFunction;
ClusterAnalysis.DefaultLinkageFunction = Linkage.CentroidFunction;
```

Code Example – VB hierarchical cluster analysis

```
ClusterAnalysis.DefaultDistanceFunction = Distance.MaximumFunction
ClusterAnalysis.DefaultLinkageFunction = Linkage.CentroidFunction
```

This changes the default distance and linkage functions for all subsequently constructed **ClusterAnalysis** objects.

You can also specify non-default distance and linkage functions in the constructor:

Code Example – C# hierarchical cluster analysis

```
var ca = new ClusterAnalysis( data,
  Distance.PowerFunction( 1.25, 2.0 ), Linkage.CompleteFunction );
```

Code Example – VB hierarchical cluster analysis

```
Dim CA As New ClusterAnalysis(Data,
  Distance.PowerFunction(1.25, 2.0), Linkage.CompleteFunction)
```

After construction, you can retrieve information about the **ClusterAnalysis** configuration using the provided properties:

- `N` gets the total number of objects being clustered.
- `DistanceFunction` gets and sets the distance function delegate used to measure the distance between individual objects. Setting the distance function using the `DistanceFunction` property has no effect until `Update()` is called with new data. (See below.)
- `LinkageFunction` gets and sets the linkage function used to measure the distance between clusters of objects. Setting the linkage delegate using the `LinkageFunction` property has no effect until `Update()` is called with new data. (See below.)

Cluster Analysis Results

The `Distances` property gets the vector of distances between all possible object pairs, computed using the current distance delegate. For `n` objects, the distance vector is of length `(n-1)(n/2)`, with distances arranged in the order:

```
(1,2), (1,3), ..., (1,n), (2,3), ..., (2,n), ..., ..., (n-1,n)
```

`Linkages` gets an `(n-1) x 3` matrix containing the complete hierarchical linkage tree, computed from `Distances` using the current linkage delegate. At each level in the tree, columns `1` and `2` contain the indices of the clusters linked to form the next cluster. Column `3` contains the distances between the clusters. For example, this code clusters `8` random vectors of length `3`, then shows a sample output of the hierarchical cluster tree:

Code Example – C# hierarchical cluster analysis

```
var data = new DoubleMatrix( 8, 3, new RandGenUniform() );
var ca = new ClusterAnalysis( data );
Console.WriteLine( ca.Linkages );
```

Code Example – VB hierarchical cluster analysis

```
Dim Data As New DoubleMatrix(8, 3, New RandGenUniform())
Dim CA As New ClusterAnalysis(Data)
Console.WriteLine(ca.Linkages)
```

Sample output:

```
7x3 [
      4 7 0.194409151975696
      3 5 0.290431894003636
      2 9 0.495557235783239
      1 6 0.508966210536187
      0 11 0.522321103698264
      8 10 0.590187697768796
      12 13 0.621675638177606 ]
```

Each object is initially assigned to its own singleton cluster, numbered `0` to `7`. 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. The first new cluster formed by the linkage function is assigned index `8`, the second is assigned index `9`, and so forth. When these indices appear later in the tree, the clusters are being combined again into a still larger cluster.

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. For example, this code cuts the linkage tree to form `3` clusters:

Code Example – C# hierarchical cluster analysis

```
ca.CutTree( 3 );
```

Code Example – VB hierarchical cluster analysis

```
CA.CutTree(3)
```

This code cuts the linkage tree at a height of `0.75`:

Code Example – C# hierarchical cluster analysis

```
ca.CutTree( 0.75 );
```

Code Example – VB hierarchical cluster analysis

```
CA.CutTree(0.75)
```

The `CutTree()` method returns a **ClusterSet** object, which represents a collection of objects assigned to a finite number of clusters. The `NumberOfClusters` property

gets the number of clusters into which objects are grouped; `N` gets the number of objects. The `Clusters` property returns an array of integers that identifies the cluster into which each object was grouped. Cluster numbers are arbitrary, and range from `0` to `NumberOfClusters - 1`. The indexer gets the cluster to which a given object is assigned. The `Cluster()` method returns the objects assigned to a given cluster as an array of integers. For instance:

Code Example – C# hierarchical cluster analysis

```
// Cluster 10 random vectors of length 4:
var data = new DoubleMatrix( 10, 4, new RandGenUniform() );
var ca = new ClusterAnalysis( data );

// Cut the tree into 5 clusters
ClusterSet cut = ca.CutTree( 5 );

Console.WriteLine( "ClusterSet = " + cut );
Console.WriteLine( "Object 0 is in cluster: " + cut[0] );
Console.WriteLine( "Object 3 is in cluster: " + cut[3] );
Console.WriteLine( "Object 8 is in cluster: " + cut[8] );
int[] objects = cut.Cluster( 1 );
Console.Write( "Objects in cluster 1: " );
for (int i = 0; i < objects.Length; i++ )
{
  Console.Write( objects[i] + " " );
}
Console.WriteLine();
```

Code Example – VB hierarchical cluster analysis

```
'' Cluster 10 random vectors of length 4:
Dim Data As New DoubleMatrix(10, 4, New RandGenUniform())
Dim ca As New ClusterAnalysis(Data)
```

```
'' Cut the tree into 5 clusters
Dim Cut As ClusterSet = CA.CutTree(5)

Console.WriteLine("ClusterSet = " & cut)
Console.WriteLine("Object 0 is in cluster: " & Cut(0))
Console.WriteLine("Object 3 is in cluster: " & Cut(3))
Console.WriteLine("Object 8 is in cluster: " & Cut(8))
Dim Objects() As Integer = Cut.Cluster(1)
Console.Write("Objects in cluster 1: ")
For I As Integer = 0 To Objects.Length - 1
  Console.Write(Objects(I) & " ")
Next
Console.WriteLine()
```

Sample output:

```
ClusterSet = 0,1,2,1,1,1,3,1,4,1
Object 0 is in cluster: 0
Object 3 is in cluster: 1
Object 8 is in cluster: 4
Objects in cluster 1: 1 3 4 5 7 9
```

Lastly, the `CopheneticDistances` property on class **ClusterAnalysis** gets the vector of cophenetic distances between all possible object pairs. The cophenetic distance between two objects is defined to be the intergroup distance when the objects are first combined into a single cluster in the linkage tree. The format is the same as the distance vector returned by `Distances`.

The correlation between the original `Distances` and the `CopheneticDistances` is sometimes taken as a measure of the appropriateness of a cluster analysis relative to the original data:

Code Example – C# hierarchical cluster analysis

```
var ca = new ClusterAnalysis( data );
double r = StatsFunctions.Correlation( ca.Distances,
                                       ca.CopheneticDistances );
```

Code Example – VB hierarchical cluster analysis

```
Dim CA As New ClusterAnalysis(Data)
Dim R As Double = StatsFunctions.Correlation(CA.Distances,
 CA.CopheneticDistances)
```

Reusing Cluster Analysis Objects

Method `Update()` updates an existing **ClusterAnalysis** instance with new data, and optionally with new distance and linkage functions. For example:

Code Example – C# hierarchical cluster analysis

```
var ca = new ClusterAnalysis( data, Linkage.SingleFunction );
Console.WriteLine( ca.Linkages );

ca.Update( data, Linkage.CompleteFunction );
Console.WriteLine( ca.Linkages );
```

Code Example – VB hierarchical cluster analysis

```
Dim CA As New ClusterAnalysis(Data, Linkage.SingleFunction)
Console.WriteLine(CA.Linkages)

CA.Update(Data, Linkage.CompleteFunction)
Console.WriteLine(CA.Linkages)
```

46.4 K-Means Clustering

The *k*-means clustering method assigns data points into *k* groups such that the sum of squares from points to the computed cluster centers is minimized. In **NMath Stats**, class **KMeansClustering** performs *k*-means clustering.

The algorithm used is that of Hartigan and Wong (*A K-means clustering algorithm*. Applied Statistics 28, 100–108. 1979):

1. For each point, move it to another cluster if that would lower the sum of squares from points to the computed cluster centers.
2. If a point is moved, immediately update the cluster centers of the two affected clusters.
3. Repeat until no points are moved, or the specified maximum number of iterations is reached.

Creating KMeansClustering Objects

A **KMeansClustering** instance is constructed from a matrix or a dataframe containing numeric data. Each row in the data set represents an object to be clustered.

Code Example – C# k-means clustering

```
var km = new KMeansClustering( data );
```

Code Example – VB k-means clustering

```
Dim KM As New KMeansClustering(Data)
```

After construction, you can retrieve information about the **KMeansClustering** data using the provided properties:

- `N` gets the total number of objects being clustered.
- `Data` gets and set the data matrix

Stopping Criteria

Iteration stops when either clustering stabilizes, or the maximum number of iterations is reached. You can specify the maximum number of iterations in several ways:

- The static `DefaultMaxIterations` property gets and sets the default maximum number of iterations for instances of **KMeansClustering**. (Initially set to `1000`.)
- You can specify a non-default maximum in the **KMeansClustering** constructor. For instance:

  ```
  var km = new KMeansClustering( data, 100 );
  ```

- The `MaxIterations` property gets and sets the maximum number of iterations on an existing **KMeansClustering** instance.

Clustering

The `Cluster()` method clusters the data into the specified number of clusters. The method accepts either *k*, the number of clusters, or a matrix of initial cluster centers:

- If *k* is given, a set of distinct rows in the data matrix are chosen as the initial centers using the algorithm specified by a `KMeanClustering.Start` enumerated value. By default, rows are chosen at random.
- If a matrix of initial cluster centers is given, *k* is inferred from the number of rows.

For example, this code clusters eight random vectors of length three into two clusters, using random starting cluster centers:

Code Example – C# k-means clustering

```
var data = new DoubleMatrix( 8, 3, new RandGenUniform() );
var cl = new KMeansClustering( data );
ClusterSet clusters = cl.Cluster( 2 );
```

Code Example – VB k-means clustering

```
Dim Data As New DoubleMatrix(8, 3, New RandGenUniform())
Dim CL As New KMeansClustering(Data)
Dim Clusters As ClusterSet = CL.Cluster(2)
```

This code specifies the two starting centers:

Code Example – C# k-means clustering

```
var centers = new DoubleMatrix("2x3 [ 0 0 0  1 1 1 ]");
ClusterSet clusters = cl.Cluster( centers );
```

Code Example – VB k-means clustering

```
Dim Centers As New DoubleMatrix("2x3 [ 0 0 0  1 1 1 ]")
Dim Clusters As ClusterSet = CL.Cluster(Centers)
```

Cluster Analysis Results

The `Cluster()` method returns a **ClusterSet** object, which represents a collection of objects assigned to a finite number of clusters. Properties on the **KMeansClustering** instance give additional information about the clustering just performed:

- `K` gets the number of clusters.
- `InitialCenters` gets the matrix of initial cluster centers.
- `FinalCenters` gets the matrix of final cluster centers.
- `Clusters` gets the cluster assignments.
- `WithinSumOfSquares` gets the within-cluster sum of squares for each cluster.
- `Sizes` gets the number of points in each cluster.
- `Iterations` gets the number of iterations performed.
- `MaxIterationsMet` returns `true` if the clustering stopped because the maximum number of iterations was reached; otherwise, `false`.

For instance, this code clusters `30` random vectors of length three into three clusters, and prints out the results:

Code Example – C# k-means clustering

```
var data = new DoubleMatrix(30, 3, new RandGenUniform());
var km = new KMeansClustering(data);
km.Cluster(3);

Console.WriteLine( "k = {0}", km.K );
```

```
Console.WriteLine( "Initial cluster centers:" );
Console.WriteLine( km.InitialCenters.ToTabDelimited() );
Console.WriteLine( "{0} iterations", km.Iterations );
Console.WriteLine("Stopped because max iterations of {0} met? {1}",
  km.MaxIterations, km.MaxIterationsMet);
Console.WriteLine( "Final cluster centers:" );
Console.WriteLine( km.FinalCenters.ToTabDelimited() );
Console.WriteLine( "Clustering assignments:" );
Console.WriteLine( km.Clusters );
for (int i = 0; i < km.K; i++) {
  Console.WriteLine( "Cluster {0} has {1} items", i, km.Sizes[i] );
}
```

Code Example – VB k-means clustering

```
Dim Data As New DoubleMatrix(30, 3, New RandGenUniform())
Dim KM As New KMeansClustering(Data)
KM.Cluster(3)

Console.WriteLine("k = {0}", KM.K)
Console.WriteLine("Initial cluster centers:")
Console.WriteLine(KM.InitialCenters.ToTabDelimited())
Console.WriteLine("{0} iterations", KM.Iterations)
Console.WriteLine("Stopped because max iterations of {0} met? {1}",
  KM.MaxIterations, KM.MaxIterationsMet)
Console.WriteLine("Final cluster centers:")
Console.WriteLine(KM.FinalCenters.ToTabDelimited())
Console.WriteLine("Clustering assignments:")
Console.WriteLine(KM.Clusters)
For I As Integer = 0 To KM.K - 1
  Console.WriteLine("Cluster {0} has {1} items", I, KM.Sizes(I))
Next
```

CHAPTER 47.
NONNEGATIVE MATRIX FACTORIZATION

Nonnegative matrix factorization (NMF) approximately factors a matrix *V* into two matrices, *W* and *H*:

$$V \approx WH$$

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 *m* *n*-dimensional 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 *h* of *H*, $v \approx Wh$. This extracts underlying features of the data as basis vectors in *W*, which can then be used for identification, classification, and compression. By not allowing negative entries in *W* and *H*, NMF enables a non-subtractive combination of the parts to form a whole.

NMath Stats provides classes for basic NMF, and for data clustering using NMF. This chapter describes how to use the NMF classes.

47.1 Nonnegative Matrix Factorization

NMath Stats provides class **NMFact** for performing basic nonnegative matrix factorization (NMF). **NMFact** uses an iterative algorithm with the goal of minimizing a cost function. The cost function is usually $\|V - WH\|$, where $\|.\|$ denotes the Frobenius matrix norm.

NMFact objects can factor data contained in either a **DoubleMatrix** or a **DataFrame** object. The factors *W* and *H* are then accessed through properties:

Code Example – C# nonnegative matrix factorization (NMF)

```
DataFrame data;       // data to be factored
int k;                // number of columns in W

var fact = new NMFact();
fact.Factor( data, k );
Console.WriteLine( "W = " + fact.W );
```

```
Console.WriteLine( "H = " + fact.H );
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Dim Data As DataFrame '' data to be factored
Dim K As Integer '' number of columns in W

Dim Fact As New NMFact()
Fact.Factor(Data, K)
Console.WriteLine("W = " & Fact.W)
Console.WriteLine("H = " & Fact.H)
```

Parameters governing aspects of the computation are set through properties or passed as constructor arguments. `ComputeCostAtEachStep` determines whether or not the cost is computed at each step of the iteration. This can be an expensive calculation and so should generally be done only when you want to investigate convergence properties, such as the convergence rate. If `ComputeCostAtEachStep` is `true`, the **DoubleVector** of costs can be accessed through the `StepCost` property.

`NumIterations` specifies the number of iterations performed in the computing of the factorization.

For example:

Code Example – C# nonnegative matrix factorization (NMF)

```
fact.ComputeCostAtEachStep = true;
fact.NumIterations = numIterations;
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Fact.ComputeCostAtEachStep = True
Fact.NumIterations = NumIterations
```

Update Algorithms

The iterative update step and cost function are specified in a class implementing the **INMFUpdateAlgorithm** interface. **NMath Stats** provides four such implementations. All matrices of uniform `(0,1)` random deviants as the initial values for *W* and *H*.

- Class **NMFAlsUpdate** uses the Alternating Least Squares (ALS) update algorithm. ALS takes advantage of the fact that while the optimization problem is not simultaneously convex in *W* and *H*, it is convex in either *W* or *H*. Thus, given one matrix, the other can be found with a simple least squares computation:

  1. Solve for H in matrix equation $W^TWH = W^TV$.

  2. Set all negative elements of H to `0`.

3. Solve for W in the matrix equation $HH^TWT = HV^T$.

4. Set all negative elements of W to `0`.

- Class **NMFDivergenceUpdate** minimizes a divergence functional. The functional is related to the Poisson likelihood of generating *V* from *W* and *H*:

$$D = \sum_{i,j} V_{i,j} \log\left(\frac{V_{i,j}}{(WH)_{i,j}}\right) - V_{i,j} + (WH)_{i,j}$$

For more information, see Brunet, Jean-Philippe et al., "Metagenes and Molecular Pattern Discovery Using Matrix Factorization", *Proceedings of the National Academy of Sciences* 101, no. 12 (March 23, 2004): 4164-4169.

- Class **NMFGdClsUpdate** uses the Gradient Descent - Constrained Least Squares (GDCLS) algorithm. In some cases it may be desirable to enforce a statistical sparsity constraint on the *H* matrix. As the sparsity of *H* increases, the basis vectors become more localized—that is, the parts-based representation of the data in *W* becomes more and more enhanced. The GDCLS algorithm enforces sparsity in *H* using a scheme that penalizes the number of non-zero entries in *H*. It is a hybrid algorithm that uses the multiplicative update rule for updating *W*, while *H* is calculated using a constrained least squares model as the metric. The algorithm follows:

$W_{ic} \leftarrow W_{ic}((VH^T)_{ic} / (WHH^T)_{ic})$

Solve for *H* in the constrained least squares problem

$(W^TW + \lambda I)H = W^TV$

Rephrase the constrained least squares step for finding *H* as

$\mathrm{Min}_H \{||V - WH||^2 + \lambda||H||^2\}$

From this it is seen that the parameter λ is a regularization value that is used to balance the reduction of the metric

$||V - WH||$

with the enforcement of smoothness and sparsity of *H*.

- Class **NMFMultiplicativeUpdate** uses a multiplicative update rule for *W* and *H*, as proposed by Lee and Seung.

$H_{cj} \leftarrow H_{cj}((W^TV)_{cj} / (W^TWH)_{cj})$

$W_{ic} \leftarrow W_{ic}((VH^T)_{ic} / (WHH^T)_{ic})$

This multiplicative method can be classified as a diagonally-scaled gradient descent method.

The update algorithm can be specified either as a constructor argument, or using the `UpdateAlgorithm` property. For instance:

Code Example – C# nonnegative matrix factorization (NMF)

```
var alg = new NMFAlsUpdate();
var fact = new NMFact( alg );
fact.Factor( data, k );
Console.WriteLine( "ALS W = " + fact.W );
Console.WriteLine( "ALS H = " + fact.H );

fact.UpdateAlgorithm = new NMFGdClsUpdate();
fact.Factor( data, k );
Console.WriteLine( "GDCLS W = " + fact.W );
Console.WriteLine( "GDCLS H = " + fact.H );
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Dim Alg As New NMFAlsUpdate()
Dim Fact As New NMFact(Alg)
Fact.Factor(Data, K)
Console.WriteLine("ALS W = " & Fact.W)
Console.WriteLine("ALS H = " & Fact.H)

Fact.UpdateAlgorithm = New NMFGdClsUpdate()
Fact.Factor(Data, K)
Console.WriteLine("GDCLS W = " & Fact.W )
Console.WriteLine("GDCLS H = " & Fact.H )
```

47.2 Data Clustering Using NMF

NMath Stats provides class **NMFClustering** for performing data clustering using iterative nonnegative matrix factorization (NMF), where each iteration step produces a new W and H. At each iteration, each column v of V is placed into a cluster corresponding to the column w of W which has the largest coefficient in H. That is, column v of V is placed in cluster i if the entry h_{ij} in H is the largest entry in column h_j of H. Results are returned as an *adjacency matrix* whose i, jth value is `1` if columns i and j of V are in the same cluster, and `0` if they are not.

Iteration stops when the clustering of the columns of the matrix V stabilizes. There are three parameters that control iteration:

- the maximum number of iterations to perform

- the stopping adjacency, which is the number of consecutive times the adjacency matrix remains unchanged before it is considered stabilized
- the convergence check period. Computing the adjacency matrix can be a somewhat expensive operation, so you may want to perform this operation only every *n*th iteration.

For example, running a **NMFClustering** instance with maximum iterations `2000`, stopping adjacency `40`, and convergence check period `10`, computes a new adjacency matrix every `10` iterations, and checks it against the previous adjacency matrix. If they are the same, a count is incremented. The iteration stops when `40` consecutive unchanged adjacency matrices are recorded, or the maximum `2000` iterations are reached.

Creating NMFClustering Instances

Class **NMFClustering** is parameterized on the NMF update algorithm to use (Section 47.1). For instance:

Code Example – C# nonnegative matrix factorization (NMF)

```
var nmfClustering = new NMFClustering<NMFDivergenceUpdate>();
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Dim NMFClustering As New NMFClustering(Of NMFDivergenceUpdate)()
```

The update algorithm can be changed post-construction using the `Updater` property.

Code Example – C# nonnegative matrix factorization (NMF)

```
nmfClustering.Updater = new NMFGdClsUpdate();
```

Code Example – VB nonnegative matrix factorization (NMF)

```
NMFClustering.Updater = New NMFGdClsUpdate()
```

The maximum iterations, stopping adjacency, and convergence check period can be specified either as constructor parameters, or post-construction using the `MaxFactorizationIterations`, `StoppingAdjacency`, and `ConvergenceCheckPeriod` properties, respectively. The default maximum number of iterations is `2000`, the default stopping adjacency is `40`, and the default convergence check period is `10`.

Performing the Factorization

The `Factor()` method performs the actual iterative factorization:

Code Example – C# nonnegative matrix factorization (NMF)

```
DoubleMatrix data;    // data to be factored
int k;                // number of columns in W
nmfClustering.Factor( data, k );
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Dim Data As DoubleMatrix...    '' data to be factored
Dim K As Integer...            '' number of columns in W
NMFClustering.Factor(Data, K)
```

NMFClustering objects can factor data contained in either a **DoubleMatrix** or a **DataFrame** object.

Cluster Results

After clustering, the `Converged` property checks if the iterative factorization converged before hitting the default maximum number of iterations. `Iterations` gets the total number of iterations performed in the most recent calculation. For example:

Code Example – C# nonnegative matrix factorization (NMF)

```
if ( nmfClustering.Converged ) {
  Console.WriteLine( "Factorization converged in {0} iterations.",
    nmfClustering.Iterations );
}
else {
  Console.WriteLine(
    "Factorization failed to converge in {0} iterations.",
    nmfClustering.MaxFactorizationIterations );
}
```

Code Example – VB nonnegative matrix factorization (NMF)

```
If (NMFClustering.Converged) Then
  Console.WriteLine("Factorization converged in {0} iterations.",
    NMFClustering.Iterations)
Else
  Console.WriteLine("Factorization failed to converge in {0}
    iterations.", NMFClustering.MaxFactorizationIterations)
End If
```

If clustering converged, the final factors *W* and *H* are accessed through properties `W` and `H`:

Code Example – C# nonnegative matrix factorization (NMF)

```
Console.WriteLine( "W = " + nmfClustering.W );
Console.WriteLine( "H = " + nmfClustering.H );
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Console.WriteLine("W = " & NMFClustering.W)
Console.WriteLine("H = " & NMFClustering.H)
```

The `Connectivity` property returns the final adjacency matrix as an instance of **ConnectivityMatrix**. The connectivity matrix is an adjacency matrix, *A*, such that columns of the factored matrix are in the same cluster if `A[i,j] == 1`, and are in different clusters if `A[i,j] == 0`. For instance:

Code Example – C# nonnegative matrix factorization (NMF)

```
ConnectivityMatrix connectivity = nmfClustering.Connectivity;
Console.WriteLine( "Connectivity Matrix: " );
Console.WriteLine( connectivity.ToTabDelimited() );
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Dim Connectivity As ConnectivityMatrix = NMFClustering.Connectivity
Console.WriteLine("Connectivity Matrix: ")
Console.WriteLine(Connectivity.ToTabDelimited())
```

The `ClusterSet` property returns a **ClusterSet** (Section 46.3) describing the final clusters:

Code Example – C# nonnegative matrix factorization (NMF)

```
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();
}
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Dim CS As ClusterSet = NMFClustering.ClusterSet

'' Print out the cluster each column belongs to
For I As Integer = 0 To CS.N - 1
  Console.WriteLine("Column {0} belongs to cluster {1}", I, CS(I))
Next
```

```
'' Print out the the members of each cluster
For I As Integer = 0 To CS.NumberOfClusters - 1
  Dim Members() As Integer = CS.Cluster(I)
  Console.Write("Cluster number {0} contains: ", I)
  For J As Integer = 0 To Members.Length - 1
    Console.Write("{0} ", J)
    Next
  Console.WriteLine()
Next
```

Lastly, the `Cost` property gets the value of the cost function for the factorization.

Code Example – C# nonnegative matrix factorization (NMF)

```
double cost = nmfClustering.Cost;
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Dim Cost As Double = NMFClustering.Cost
```

The cost function is the function that is minimized by the NMF update algorithm.

Computing a Consensus Matrix

NMF uses an iterative algorithm with random starting values for *W* and *H*. This, coupled with the fact that the factorization is not unique, means that if you cluster the columns of V multiple times, you may get different final clusterings. The *consensus matrix* is a way to average multiple clusterings, to produce a probability estimate that any pair of columns will be clustered together.

To compute the consensus matrix, the columns of *V* are clustered using NMF *n* times. Each clustering yields a connectivity matrix. Recall that the connectivity matrix is a symmetric matrix whose *i*, *j*th entry is `1` if columns *i* and *j* of *V* are clustered together, and `0` if they are not. The consensus matrix is also a symmetric matrix, whose *i*, *j*th entry is formed by taking the average of the *i*, *j*th entries of the *n* connectivity matrices.

Thus, each *i*, *j*th entry of the consensus matrix is a value between `0`, when columns *i* and *j* are not clustered together on any of the runs, and `1`, when columns i and j were clustered together on all runs. The *i*, *j*th entry of a consensus matrix may be considered, in some sense, a "probability" that columns *i* and *j* belong to the same cluster.

NMath Stats provides class **NMFConsensusMatrix** for compute a consensus matrix. **NMFConsensusMatrix** is parameterized on the NMF update algorithm to use (Section 47.1). Additional constructor parameters specify the matrix to factor, the order *k* of the NMF factorization (the number of columns in *W*), and the number of clustering runs. For example:

Code Example – C# nonnegative matrix factorization (NMF)

```
DoubleMatrix data;    // data to be factored
int k;                // number of columns in W
int numberOfRuns = 70;

var consensusMatrix =
  new NMFConsensusMatrix<NMFDivergenceUpdate>(data, k,
    numberOfRuns);
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Dim Data As DoubleMatrix...    '' data to be factored
Dim K As Integer...    '' number of columns in W
Dim NumberOfRuns As Integer = 70

Dim ConsensusMatrix As New NMFConsensusMatrix(Of
  NMFDivergenceUpdate)(Data, K, NumberOfRuns)
```

The consensus matrix is computed at construction time, so be aware that this may be an expensive operation. Post-construction, the `NumberOfConvergedRuns` property gets the number of clustering runs where the NMF computation converged:

Code Example – C# nonnegative matrix factorization (NMF)

```
Console.WriteLine( "{0} runs out of {1} converged.",
  consensusMatrix.NumberOfConvergedRuns, numberOfRuns );
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Console.WriteLine("{0} runs out of {1} converged.",
  ConsensusMatrix.NumberOfConvergedRuns, NumberOfRuns)
```

NMFConsensusMatrix provides a standard indexer for getting the element value at a specified row and column in the consensus matrix. For example, this code gets the probability that columns `2` and `7` will be clustered together:

Code Example – C# nonnegative matrix factorization (NMF)

```
double p = consensusMatrix[2, 7];
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Dim P As Double = ConsensusMatrix(2, 7)
```

This code prints the entire consensus matrix:

Code Example – C# nonnegative matrix factorization (NMF)

```
Console.WriteLine( "Consensus Matrix:" );
Console.WriteLine( consensusMatrix.ToTabDelimited() );
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Console.WriteLine("Consensus Matrix:")
Console.WriteLine(ConsensusMatrix.ToTabDelimited())
```

A consensus matrix, *C*, can also used to perform a hierarhical clustering of the columns of *V* (Section 46.3), using the distance function:

$$\text{distance}_{i,\,j} = 1.0 - C_{i,\,j}$$

A **ClusterAnalysis** instance is constructed from a matrix containing numeric data. Each row in the data set represents an object to be clustered. In this case, you're simply clustering the column numbers of *V*, so construct a matrix with one colunm containing the numbers `0` to `n-1`, where *n* is the number of columns of *V* (and the order of of the consensus matrix):

Code Example – C# nonnegative matrix factorization (NMF)

```
var colNumbers =
  new DoubleMatrix( consensusMatrix.Order, 1, 0, 1 );

Distance.Function distance =
  delegate( DoubleVector data1, DoubleVector data2 ) {
    int i = (int)data1[0];
    int j = (int)data2[0];
    return 1.0 - consensusMatrix[i, j];
  };

var ca = new ClusterAnalysis( colNumbers, distance );
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Dim ColNumbers As New DoubleMatrix(ConsensusMatrix.Order, 1, 0, 1)

Dim distance As Distance.Function = Function(Data1 As DoubleVector,
  Data2 As DoubleVector)
  Dim I As Integer = CType(Data1(0), Integer)
  Dim J As Integer = CType(Data2(0), Integer)
  Return 1.0 - ConsensusMatrix(I, J)
End Function

Dim CA As New ClusterAnalysis(ColNumbers, distance)
```

After you've created a **ClusterAnalysis** object, 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. For example, this code cuts the linkage tree to form three clusters:

Code Example – C# nonnegative matrix factorization (NMF)

```
ClusterSet clusters = ca.CutTree( 3 );
```

```
for ( int i = 0; i < clusters.NumberOfClusters; i++ ) {
  int[] members = clusters.Cluster( i );
  Console.Write( "Cluster {0} contains: ", i );
  for ( int j = 0; j < members.Length; j++ ) {
    Console.Write( "{0} ", members[j] );
  }
  Console.WriteLine();
}
```

Code Example – VB nonnegative matrix factorization (NMF)

```
Dim Clusters As ClusterSet = CA.CutTree(3)

For I As Integer = 0 To Clusters.NumberOfClusters - 1
  Dim Members() As Integer = Clusters.Cluster(I)
  Console.Write("Cluster {0} contains: ", I)
  For J As Integer = 0 To Members.Length - 1
    Console.Write("{0} ", Members(J))
  Next
  Console.WriteLine()
Next
```

CHAPTER 48.
PARTIAL LEAST SQUARES

Partial Least Squares (PLS) is a technique that generalizes and combines features from principal component analysis (Section 46.1) and multiple linear regression (Chapter 42). It is particularly useful when you need to predict a set of response (dependent) variables from a large set of predictor (independent variables).

As in multiple linear regression, the goal of PLS regression is to construct a linear model

$$Y = XB + E$$

where Y is n cases by m variables response matrix, X is a n cases by p variables predictor matrix, B is a p by m regression coefficients matrix, and E is a noise term for the model which has the same dimensions as Y.

As in principal components regression, PLS regression produces factor scores as linear combinations of the original predictor variables, so that there is no correlation between the factor score variables used in the predictive regression model. For example, suppose that we have a matrix of response variables Y, and a large number of predictive variables X (in matrix form), some of which may be highly correlated. A regression using factor extraction for this data computes the score matrix $T=XW$ for an appropriate matrix of weights W, and then considers the linear regression model $Y=TQ+E$, where Q is a matrix of regression coefficient, called loadings, for T, and E is an error term. Once the loadings Q are computed, the above regression model is equivalent to $Y=XB+E$, with $B=WQ$, which can be used as a predictive model.

PLS regression differs from principal components regression in the methods used for extracting factor scores. While principal components regression computes the weight matrix W reflecting the covariance structure between predictor variables, PLS regression produces the weight matrix W reflecting the covariance structure between the predictor and response variables.

For establishing the model with c factors, or components, PLS regression produces a p by c weight matrix W for X such that $T=XW$. These weights are computed so that each of them maximizes the covariance between responses and the corresponding factor scores. Ordinary least squares regression of Y on T are then performed to produce Q, the loadings for Y (or weights for Y) such that $Y=TQ+E$. Once Q is computed, we have $Y=XB+E$, where $B=WQ$.

48.1 Computing a PLS Regression

NMath Stats provides two classes for performing partial least squares (PLS) regression, **PLS1** and **PLS2**:

- **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:

Code Example – C# partial least squares (PLS)

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

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

Code Example – VB partial least squares (PLS)

```
Dim A As DoubleMatrix = ...
Dim Y As DoubleVector = ...
Dim NumComponents As Integer = 3

Dim PLS As New PLS1(A, Y, NumComponents)
```

You can also invoke the `Calculate()` function on **PLS1** or **PLS2** to calculate a regression on an existing instance:

Code Example – C# partial least squares (PLS)

```
pls.Calculate( A, y, numComponents );
```

Code Example – VB partial least squares (PLS)

```
PLS.Calculate(A, Y, NumComponents)
```

48.2 Error Checking

After computing a PLS regression, always check the `IsGood` property to ensure that there were no errors in performing the calculation. If `IsGood` returns the `false`, the `Message` property will contain a message indicating the nature of the error. For example, the following code checks that the calculation succeeded, and if not, prints out the error message and returns:

Code Example – C# partial least squares (PLS)

```
if (pls.IsGood) {
  Console.WriteLine("Success");
}
else {
  Console.WriteLine("PLS calculation failed: " + pls.Message);
  return;
}
```

Code Example – VB partial least squares (PLS)

```
If (PLS.IsGood) Then
  Console.WriteLine("Success")
Else
  Console.WriteLine("PLS calculation failed: " & PLS.Message)
  Return
End If
```

One common source of calculation failure occurs when the number of components specified for the calculation is greater than the rank of X, the matrix of predictor variables. If this occurs, try decreasing the number of components for the regression until the calculation succeeds. You can also use Cross Validation (Section 48.6) to determine the optimal number of components.

If the calculation fails due to the non-convergence of the Iterative Power Method for computing dominant eigenvectors, you may want to adjust the maximum number of iterations and/or the tolerance for this method (Section 48.5).

48.3 Predicted Values

Once you've performed a PLS regression (Section 48.1), you can calculate the predicted value of the response variable for a given value of the predictor variable.

Code Example – C# partial least squares (PLS)

```
double plsYhat = pls.Predict(x);
```

Code Example – VB partial least squares (PLS)

```
Dim PLSYHat As Double = PLS.Predict(X)
```

or for a set of predictor values:

Code Example – C# partial least squares (PLS)

```
DoubleVector plsYhatVec = pls.Predict(A);
```

Code Example – VB partial least squares (PLS)

```
Dim PLSTYHatVec As DoubleVector = PLS.Predict(A)
```

48.4 Analysis of Variance

NMath Stats provides the classes **PLS1Anova** and **PLS2Anova** for performing a classic analysis of variance (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. For instance:

Code Example – C# partial least squares (PLS)

```
var plsAnova = new PLS2Anova(pls);
DoubleVector ssTotal = plsAnova.SumOfSquaresTotal;
DoubleVector ssResiduals = plsAnova.SumOfSquaresResiduals;
DoubleVector se = plsAnova.StandardError;
DoubleVector rms = plsAnova.RootMeanSqrErrorPrediction;
DoubleVector rSquared = plsAnova.CoefficientOfDetermination;
```

Code Example – VB partial least squares (PLS)

```
Dim PlsAnova As New PLS2Anova(PLS)
Dim SSTotal As DoubleVector = PlsAnova.SumOfSquaresTotal
Dim SSResiduals As DoubleVector = PlsAnova.SumOfSquaresResiduals
Dim SE As DoubleVector = PlsAnova.StandardError
Dim RMS As DoubleVector = PlsAnova.RootMeanSqrErrorPrediction
Dim RSquared As DoubleVector = PlsAnova.CoefficientOfDetermination
```

48.5 PLS Algorithms

NMath Stats provides classes **PLS1NipalsAlgorithm** and **PLS2NipalsAlgorithm** which implement the Nonlinear Iterative PArtial Least Squares (NIPALS) algorithm for **PLS1** and **PLS2** respectively, and class **PLS2SimplsAlgorithm** which implements the Straightforward IMplementation of PLS (SIMPLS) algorithm for **PLS2**.

The algorithm to use may be specified in the constructor for a **PLS1** or **PLS2** object, or set through the `Calculator` property:

Code Example – C# partial least squares (PLS)

```
var calculator = new PLS2SimplsAlgorithm();
pls.Calculator = calculator;
```

Code Example – VB partial least squares (PLS)

```
Dim Calculator As New PLS2SimplsAlgorithm()
PLS.Calculator = Calculator
```

NOTE—Note that setting the calculator through the property forces a recalculation if data is present.

The SIMPLS algorithm for **PLS2** uses the Iterative Power Method for computing dominant eigenvectors. This algorithm produces a candidate eigenvector during each iteration which is normalized with respect to the l-infinity norm. When the two-norm of the difference between the current eigenvector, *ei*, and the eigenvector computed on the previous iteration, *ei-1*, is less than a specified tolerance, the algorithm stops. The maximum number of iteration to perform as well as the tolerance may be specified on the algorithm object.

If your **PLS2** with SIMPLS calculation fails because the power method failed to converge, you may want to adjust these values.(If the calculation failure is due to non-convergence of the power method, this will be indicated in the `Message` property of the **PLS2** object.

48.6 Cross Validation

Cross validation is a model evaluation method which measures how well a model makes predictions for data that it has not already sees (as with residuals). To accomplish this, some of the data is removed before the model is constructed. Once the model is constructed, the data that was removed can be used to test the performance of the model on the "new" data. The following methods are typically used:

- **The Holdout Method**

  The simplest kind of cross validation is the *holdout method*. The data set is separated into two sets, called the *training set* and the *testing set*. The PLS regression is constructed using the training set, then the regression model is asked to make predictions for the responses for the predictor data in the training set. The errors it makes are accumulated to give the mean square error.

- ***K*-fold Cross Validation**

  In *k-fold cross validation*, the data set is divided into *k* subsets, and the hold-out method is repeated *k* times. Each time one of the *k* subsets is used as the test set and the other *k-1* subsets are put together to form a training set. The average mean square error is then computed across all *k* trials.

- **Leave-One-Out Cross Validation**

  Leave-one-out cross validation is the result of taking *k*-fold cross validation to its logical extreme, with *k* equal to *n*, the number of data points in the set. That means that *n* separate times, the PLS model is computed using all the data except for one point and a prediction is made for that point. As before the average mean square error is computed and used to evaluate the model.

NMath Stats provides two classes for doing *k*-fold cross validation on PLS models. **PLS1CrossValidation** is used when the response data is univariate, and **PLS2CrossValidation** is used when the response data is multivariate. To perform a cross validation calculation, you need to specify the data (Section 48.1), a PLS calculation algorithm (Section 48.5), and an algorithm for dividing the data into subsets.

To specify how subsets for *k*-fold cross validation are generated from the data, you must provide the cross validation class with an object implementing the **ICrossValidationSubsets** interface. **NMath Stats** provides classes **LeaveOneOutSubsets**, which implement the leave-one-out strategy, and **KFoldSubsets**, which implements *k*-fold with arbitrary *k*.

The average mean square error for the cross validation calculation is available as a property on the cross validation object. Also available is an array of **PLS1CrossValidationResult** or **PLS2CrossValidationResult** objects. Each result object contains testing and training data that was used for each cross validation calculation and the associated mean square error.

Jackknifing of Regression Coefficients

NMath Stats also provides class **PLS2CrossValidationWithJackknife** for evaluation of multivariate PLS models with model coefficient variance estimates and confidence intervals.

The jackknife estimator of a parameter is found by systematically leaving out each observation from a dataset and calculating the estimate and then finding the average of these calculations. Given a sample of size *N*, the jackknife estimate is found by aggregating the estimates of each *N-1* estimate in the sample.

The original Tukey jackknife variance estimator is defined as

$$\frac{(g-1)}{g}\sum(B_i - \bar{B})$$

where g is the number of subsets used in cross validation, B_i is the estimated coefficients when subset i is left out (called the j*ackknife replicates*), and $\bar{B}$ is the mean of the B_i.

However, Martens and Martens (2000) defined the estimator as

$$\frac{(g-1)}{g}\sum(B_i - \hat{B})$$

where $\hat{B}$ is the coefficient estimate using the entire data set—that is, they use the original fitted coefficients instead of the mean of the jackknife replicates. This is the default for class **PLS2CrossValidationWithJackknife**, but you can set `UseMean` to `true` for the original Tukey definition. For example:

Code Example – C# PLS cross-validation with jackknife

```
int numComponents = 2;

var cv = new PLS2CrossValidationWithJackknife
{
  Scale = false,
  UseMeans = true
};
cv.DoCrossValidation( X, Y, numComponents );
Console.WriteLine( cv.CoefficientVariance );
```

Code Example – VB PLS cross-validation with jackknife

```
Dim NumComponents As Integer = 2

Dim CV As New PLS2CrossValidationWithJackknife
CV.Scale = False
CV.UseMeans = True

CV.DoCrossValidation(X, Y, NumComponents)
Console.WriteLine(CV.CoefficientVariance)
```

48.7 Partial Least Squares Discriminant Analysis

Partial least squares Discriminant Analysis (PLS-DA) is a variant used when the response variable is categorical. Three classes are provided for performing PLS-DA:

- **SparsePlsDa** performs Discriminant Analysis (DA) using a classical sparse PLS regression (sPLS), but where the response variable is categorical. The response vector Y is qualitative and is recoded as a dummy block matrix where each of the response categories are coded via an indicator variable. PLS-DA is then run as if Y was a continuous matrix. **SparsePlsDa** inherits from **PLS2**.

- **SparsePls** performs a sparse PLS calculation with variable selection. The LASSO penalization is used on the pairs of loading vectors. **SparsePls** implements **IPLS2Calc**.
- **SparsePLSDACrossValidation** performs an evaluation of a PLS model. Evaluation consists of dividing the data into two subsets: a training subset and a testing subset. A PLS calculation is performed on the training subset and the resulting model is used to predict the values of the dependent variables in the testing set. The mean square error between the actual and predicted dependent values is then calculated. Usually, the data is divided up into several training and testing subsets and calculations are done on each of these. In this case the average mean square error over each PLS calculation is reported. (The individual mean square errors are available as well.)

  The subsets to use in the cross validation are specified by providing an implementation of the **ICrossValidationSubsets** interface. Classes that implement this interface generate training and testing subsets from PLS data.

For example, if `X` is the predictor data and y the corresponding observed factor levels, this code calculates the sparse PLS-DA:

Code Example – C# Partial Least Squares Discriminant Analysis (PLS-DA)

```
int ncomp = 3;
int numXvarsToKeep = (int) Math.Round( X.Cols * 0.66 );
int[] keepX = Enumerable.Repeat( numXvarsToKeep, ncomp ).ToArray();
var splsda = new SparsePlsDa( X, y, ncomp, keepX );
```

Code Example – VB Partial Least Squares Discriminant Analysis (PLS-DA)

```
Dim NComp As Integer = 3
Dim NumXvarsToKeep As Integer = CType(Math.Round(X.Cols * 0.66),
  Integer)
Dim KeepX As Integer() = Enumerable.Repeat(NumXvarsToKeep,
  NComp).ToArray()
Dim SPLSDA As New SparsePlsDa(X, Y, NComp, KeepX)
```

The number of components to keep in the model is specified, as well as the number of predictor variables to keep for each of the components (about two thirds, in this case).

Because **SparsePlsDa** is a **PLS2**, you can use the **PLS2Anova** class to perform an ANOVA (Section 48.4).

Code Example – C# Partial Least Squares Discriminant Analysis (PLS-DA)

```
var anova = new PLS2Anova( splsda );
Console.WriteLine( "Rsqr: " + anova.CoefficientOfDetermination );
Console.WriteLine( "MSE Prediction: " +
  anova.RootMeanSqrErrorPrediction );
```

Code Example – VB Partial Least Squares Discriminant Analysis (PLS-DA)

```
Dim Anova As New PLS2Anova(SPLSDA)
Console.WriteLine("Rsqr: " & Anova.CoefficientOfDetermination)
Console.WriteLine("MSE Prediction: " &
  Anova.RootMeanSqrErrorPrediction)
```

You can also do cross validation using class **SparsePLSDACrossValidation**.

Code Example – C# Partial Least Squares Discriminant Analysis (PLS-DA)

```
var subsetGenerator = new LeaveOneOutSubsets();
var crossValidation =
  new SparsePLSDACrossValidation( subsetGenerator );

crossValidation.DoCrossValidation( X, yFactor, ncomp, keepX );

Console.WriteLine( "Cross validation average MSE: " +
  crossValidation.AverageMeanSqrError );
```

Code Example – VB Partial Least Squares Discriminant Analysis (PLS-DA)

```
Dim SubsetGenerator As New LeaveOneOutSubsets()
Dim CrossValidation As New
  SparsePLSDACrossValidation(SubsetGenerator)

CrossValidation.DoCrossValidation(X, YFactor, NComp, KeepX)

Console.WriteLine("Cross validation average MSE: " &
  CrossValidation.AverageMeanSqrError)
```

CHAPTER 49.
GOODNESS OF FIT

NMath Stats provides classes **GoodnessOfFit** and **GoodnessOfFitParameter** for testing the goodness of fit of least squares model-fitting classes, such as **LinearRegression**, **PolynomialLeastSquares**, and **OneVariableFunctionFitter**:

Available statistics include the residual standard error, the coefficient of determination (R2 and "adjusted" R2), the F-statistic for the overall model with its numerator and denominator degrees of freedom, and standard errors, t-statistics, and corresponding (two-sided) p-values for the model parameters.

This chapter describes how to use the goodness of fit classes.

NOTE—GoodnessOfFit and GoodnessOfFitParameter are a generalization of classes LinearRegressionAnova and LinearRegressionParameter (Chapter 42), respectively. As such, they duplicate the functionality of those classes for testing the goodness of fit of a LinearRegression, with the exception of the beta coefficients.

49.1 Significance of the Overall Model

Class **GoodnessOfFit** tests the overall model significance for least squares model-fitting classes, such as **LinearRegression**, **PolynomialLeastSquares**, and **OneVariableFunctionFitter**.

GoodnessOfFit instances can be constructed from:

- A **LinearRegression** object.
- A **PolynomialLeastSquares** object, plus the vectors of x and y data.
- A **OneVariableFunctionFitter** object, plus the vectors of x and y data and the solution found by the fitter.

For example:

Code Example – C# goodness of fit

```
var x = new DoubleVector(0.3330, 0.1670, 0.0833, 0.0416,
  0.0208, 0.0104, 0.0052);
var y = new DoubleVector(3.636, 3.636, 3.236, 2.660,
  2.114, 1.466, 0.866);

int degree = 2;
```

```
var pls =
  new PolynomialLeastSquares(degree, x, y);

var gof = new GoodnessOfFit(pls, x, y);
```

Code Example – VB goodness of fit

```
Dim X As New DoubleVector(0.333, 0.167, 0.0833, 0.0416, 0.0208,
  0.0104, 0.0052)
Dim Y As New DoubleVector(3.636, 3.636, 3.236, 2.66, 2.114, 1.466,
  0.866)

Dim Degree As Integer = 2
Dim PLS As New PolynomialLeastSquares(Degree, X, Y)

Dim GoF As New GoodnessOfFit(PLS, X, Y)
```

A variety of properties are provided for assessing the significance of the overall model:

- `RegressionSumOfSquares` gets the regression sum of squares. This quantity indicates the amount of variability explained by the model. It is the sum of the squares of the difference between the values predicted by the model and the mean.
- `ResidualSumOfSquares` gets the residual sum of squares. This is the sum of the squares of the differences between the predicted and actual observations.
- `ModelDegreesOfFreedom` gets the number of degrees of freedom for the model, which is equal to the number of predictors in the model.
- `ErrorDegreesOfFreedom` gets the number of degress of freedom for the model error, which is equal to the number of observations minus the number of model paramters.
- `RSquared` gets the coefficient of determination.
- `AdjustedRsquared` gets the adjusted coefficient of determination.
- `MeanSquaredResidual` gets the mean squared residual. This quantity is the equal to `ResidualSumOfSquares / ErrorDegreesOfFreedom` (equals the number of observations minus the number of model parameters).
- `MeanSquaredRegression` gets the mean squared for the regression. This is equal to `RegressionSumOfSquares / ModelDegreesOfFreedom` (equals the number of predictors in the model).
- `FStatistic` gets the overall F statistic for the model. This is equal to the ratio of `MeanSquaredRegression / MeanSquaredResidual`. This is the statistic for the hypothesis test where the null hypothesis, H_0 is that all the

parameters are equal to `0` and the alternative hypothesis is that at least one paramter is nonzero.

- `FStatisticPValue` gets the *p*-value for the *F* statistic.

For example, if `lr` is a **LinearRegression** object:

Code Example – C# goodness of fit

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

Code Example – VB goodness of fit

```
Dim GoF As New GoodnessOfFit(LR)
Dim SSE As Double = GoF.ResidualSumOfSquares
Dim R2 As Double = GoF.RSquared
Dim FStat As Double = GoF.FStatistic
Dim FStatPval As Double = GoF.FStatisticPValue
```

Lastly, the `FStatisticCriticalValue()` function computes the critical value for the *F* statistic at a given significance level:

Code Example – C# goodness of fit

```
double critVal = gof.FStatisticCriticalValue(.05);
```

Code Example – VB goodness of fit

```
Dim CritVal As Double = GoF.FStatisticCriticalValue(0.05)
```

49.2 Significance of Parameters

Instances of class **GoodnessOfFitParameter** test statistical hypothesis about individual parameters in a least squares model-fit.

Creating Goodness of Fit Parameter Objects

You can get an array of test objects for all parameters in a **GoodnessOfFit** using the `Parameters` property:

Code Example – C# goodness of fit

```
GoodnessOfFitParameter[] params = gof.Parameters;
```

Code Example – VB goodness of fit

```
Dim Params() As GoodnessOfFitParameter = gof.Parameters
```

Properties of Goodness of Fit Parameters

Class **GoodnessOfFitParameter** provides the following properties:

- `Index` gets the index of the parameter in the overall model.
- `Value` gets the value of the parameter.
- `StandardError` gets the standard error of the parameter.
- `DegreesOfFreedom` gets the degrees of freedom of the parameter.

Hypothesis Tests

Class **GoodnessOfFitParameter** provides the following methods for testing statistical hypotheses regarding parameter values:

- `TStatisticPValue()` returns the *p*-value for a two-sided *t* test with the null hypothesis that the parameter is equal to a given test value, versus the alternative hypothesis that it is not.
- `TStatistic()` returns the value of the *t* statistic for the null hypothesis that the parameter value is equal to a given test value.
- `TStatisticCriticalValue()` gets the critical value for the t-statistic for a given alpha level.
- `ConfidenceInterval()` returns a $1 - \alpha$ confidence interval for the parameter for a given alpha level.

For example, this code tests whether a parameter in a model is significantly different than zero:

Code Example – C# goodness of fit

```
double tstat = param.TStatistic( 0.0 );
double pValue = param.TStatisticPValue( 0.0 );
double criticalValue = param.TStatisticCriticalValue( 0.05 );
Interval confidenceInterval = param.ConfidenceInterval( 0.05 );
```

Code Example – VB goodness of fit

```
Dim TStat As Double = param.TStatistic(0.0)
Dim PValue As Double = param.TStatisticPValue(0.0)
Dim CriticalValue As Double = param.TStatisticCriticalValue(0.05)
Dim ConfidenceInterval As Interval = param.ConfidenceInterval(0.05)
```

CHAPTER 50.
PROCESS CONTROL

Statistical process control uses statistical measures to monitor and control a process. **NMath** provides classes for measuring process quality capability (Cp, Cpm, and Cpk), performance (Pp and Ppk), and Z bench.

50.1 Process Capability

Class **ProcessCapability** computes the process capability parameters Cp, Cpm, Cpk for normally distributed data. If the data are not normal, the **BoxCox** transform can be used.

Instance of **ProcessCapability** are constructed from a vector of input data measurements, a subgroup size (the data must laid out in continuous subgroups of equal size), lower and upper specification limits, and the control target process mean.

Code Example – C# process control

```
DoubleVector data = ...
int size = 5;
double LSL = 73.95;
double USL = 74.05;
double target = 74.0;
var pc = new ProcessCapability( data, size, LSL, USL, target );
```

Code Example – VB process control

```
Dim Data As DoubleVector = ...
Dim Size As Integer = 5
Dim LSL As Double = 73.95
Dim USL As Double = 74.05
Dim Target As Double = 74.0
Dim PC As New ProcessCapability(Data, Size, LSL, USL, Target)
```

If no target is given, the mean of the specification limits is used.

The standard deviation is computed using the mean of the ranges method, referred to as the `UWAVE-R` method in the R `qcc` package.

ProcessCapability provides the following properties:

- `CI95` gets the 95% confidence interval. 95% of the time the process mean will reside within this interval. The estimate is based on the t-distribution (t-score) if there are 30 or fewer samples; otherwise, the normal distribution is used (z-score).
- `Cp` gets the process capability.
- `Cpk` gets the process capability index.
- `Cpm` gets the Taguchi capability index.
- `ProcessSigma` gets the estimate of the process standard deviations used to compute Cp, Cpk, and Cpm. The standard deviation is estimated using the unweighted averages of the subgroup ranges.
- `IQR` gets the interquartile range using the Minitab interpolation method. This method uses interpolation to find the upper and lower quartiles before returning the IQR. Therefore, the IQR may be computed from points that do not exist in the data set.

50.2 Process Performance

Class **ProcessPerformance** computes the process performance indices Ppk and Pp for normally distributed data. If the data are not normal, the **BoxCox** transform can be used.

Instance of **ProcessPerformance** are constructed from a vector of input data measurements, and lower and upper specification limits.

ProcessPerformance provides the following properties:

- `Ppk` gets the process performance index.
- `Pp` gets the process performance.

For example:

Code Example – C# process control

```
DoubleVector data = ...
double LSL = 1.90;
double USL = 2.10;
var pp = new ProcessPerformance( data, LSL, USL );
Console.WriteLine( pp.Ppk );
```

Code Example – VB process control

```
DoubleVector Data = ...
Dim LSL As Double = 1.9
```

```
Dim USL As Double = 2.1
Dim PP As New ProcessPerformance(Data, LSL, USL)
Console.WriteLine(PP.Ppk)
```

50.3 Z Bench

Class **ZBench** computes the Z bench (the Z value that corresponds to the total probability of a defect,) the percent defective, and the parts per million defective.

Instance of **ZBench** are constructed from a vector of input data measurements, and lower and upper specification limits.

Code Example – C# process control

```
DoubleVector data = ...
double LSL = 1.90;
double USL = 2.10;
var zb = new ZBench( data, LSL, USL );
```

Code Example – VB process control

```
DoubleVector Data = ...
Dim LSL As Double = 1.9
Dim USL As Double = 2.1
Dim ZB As New ZBench(Data, LSL, USL)
```

Alternatively, a single-sided test can be performed using only a lower or upper specification limit. The test type is specified using a value from the **ControlLimits** enumeration: `DoubleEnded`, `LowerOnly`, or `UpperOnly`. For example:

Code Example – C# process control

```
DoubleVector data = ...
double USL = 2.10;
var zb = new ZBench( data, ControlLimits.UpperOnly, USL );
```

Code Example – VB process control

```
DoubleVector Data = ...
Dim USL As Double = 2.1
Dim ZB As New ZBench(Data, ControlLimits.UpperOnly, USL)
```

Class **ZBench** provides the following properties:

- `ZBench` gets the Z Bench.
- `PercentDefective` gets the percent defective.
- `PPMDefective` gets the parts per million defective.

PART VI - MISCELLANEOUS TOPICS

Chapter 51. Serialization

NMath data classes are fully persistable using standard .NET mechanisms. All 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.

This chapter describes how to persist **NMath** objects in binary, SOAP, and XML formats.

51.1 Binary Serialization

The **System.Runtime.Serialization.Formatters.Binary.BinaryFormatter** class provides `Serialize()` and `Deserialize()` methods for persisting an object in binary format to a given stream. For example, this code serializes two **FloatComplexMatrix** objects to a file:

Code Example – C# binary serialization

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

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();
```

Code Example – VB binary serialization

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

Dim A As New FloatComplexMatrix(
  "2x2[ (5,9.8) (-6,.9) (7,-8) (8,8) ]")
Dim B As New FloatComplexMatrix(4, 4, 0.1F, 0.1F)
```

```
Dim BinStream As FileStream = File.Create("myData.dat")
Dim BinFormatter As New BinaryFormatter()

BinFormatter.Serialize(BinStream, A)
BinFormatter.Serialize(BinStream, B)

BinStream.Close()
```

This code restores the objects from the file:

Code Example – C# binary serialization

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

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

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

Code Example – VB binary serialization

```
BinStream = File.OpenRead("myData.dat")

Dim A2 = CType(BinFormatter.Deserialize(BinStream),
  FloatComplexMatrix)
Dim B2 = CType(BinFormatter.Deserialize(BinStream),
  FloatComplexMatrix)

BinStream.Close()
File.Delete("myData.dat")
```

51.2 SOAP Serialization

The **System.Runtime.Serialization.Formatters.Soap.SoapFormatter** class provides `Serialize()` and `Deserialize()` methods for persisting an object in SOAP format to a given stream. For example, this code serializes a **FloatComplexTriDiagFact** object to a file:

Code Example – C# SOAP serialization

```
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;

int rows = 8, cols = 8;
FloatComplexVector data =
```

```
  new FloatComplexVector( cols*3, new RandGenUniform(-1, 1) );
var A =
  new FloatComplexTriDiagMatrix( data, rows, cols );
var F = new FloatComplexTriDiagFact( A );

FileStream xmlStream = File.Create( "myData.xml" );
var xmlFormatter = new SoapFormatter();
xmlFormatter.Serialize( xmlStream, F );
xmlStream.Close();
```

Code Example – VB SOAP serialization

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Soap

Dim Rows As Integer = 8
Dim Cols As Integer = 8

Dim Data As New FloatComplexVector(Cols * 3,
  New RandGenUniform(-1.0, 1.0))
Dim A As New FloatComplexTriDiagMatrix(Data, Rows, Cols)
Dim F As New FloatComplexTriDiagFact(A)

Dim XMLStream As FileStream = File.Create("myData.xml")
Dim XMLFormatter As New SoapFormatter()
XMLFormatter.Serialize(XMLStream, F)
XMLStream.Close()
```

This code restores the objects from the file:

Code Example – C# SOAP serialization

```
xmlStream = File.OpenRead( "myData.xml");
var F =
  (FloatComplexTriDiagFact)xmlFormatter.Deserialize( xmlStream );

xmlStream.Close();
File.Delete( "myData.xml" );
```

Code Example – VB SOAP serialization

```
XMLStream = File.OpenRead("myData.xml")
Dim F = CType(XMLFormatter.Deserialize(XMLStream),
  FloatComplexTriDiagMatrix)

XMLStream.Close()
File.Delete("myData.xml")
```

51.3 XML Serialization

XML serialization in .NET does not make use of CLR **Formatter** classes, as do binary serialization (Section 51.1) and SOAP serialization (Section 51.2). Instead, the framework provides the **System.Xml.Serialization.XmlSerializer** class for persisting to XML documents.

However, because **NMath** data classes implement the **IEnumerable** interface, **XmlSerializer** persists *only* the enumerated data. Thus, though a matrix or vector object can be serialized in XML, it cannot be restored.

If you want to serialize and deserialize **NMath** objects in XML format, you can easily overcome this limitation by writing a simple wrapper class that contains all the information necessary to restore the object, without implementing **IEnumerable**. For example, this code defines an **MyNameSpace.MyWrapper** class that wraps a **DoubleMatrix**:

Code Example – C# XML serialization

```
using CenterSpace.NMath.Core;

namespace MyNamespace
{
  public class MyWrapper
  {
    public int Rows;
    public int Columns;
    public StorageType Storage = StorageType.ColumnMajor;
    public double[] Data;

    public MyWrapper() {}

    public MyWrapper( DoubleMatrix A )
    {
      Rows = A.Rows;
      Columns = A.Cols;
      DoubleMatrix B = (DoubleMatrix)A.Clone();
      Data = B.DataBlock.Data;
    }

  }  // class

} // namespace
```

Code Example – XML binary serialization

```
Imports CenterSpace.NMath.Core

Namespace MyNamespace
```

```
Public Class MyWrapper

  Public Rows As Integer
  Public Columns As Integer
  Public Storage As StorageType = StorageType.ColumnMajor
  Public Data() As Double

  Public Sub New()
  End Sub

  Public Sub New(A As DoubleMatrix)
    Rows = A.Rows
    Columns = A.Cols
    Dim B As DoubleMatrix = CType(A.Clone(), DoubleMatrix)
    Data = B.DataBlock.Data
  End Sub

End Class

End Namespace
```

Note that the constructor uses the `Clone()` method to ensure that the data is not referenced by any other objects, and that it is in contiguous storage.

You could then use the wrapper class to serialize a matrix object, as shown below:

Code Example – C# XML serialization

```
using System.IO;
using System.Xml.Serialization;
using MyNamespace;

var A = new DoubleMatrix( "3x3[1 2 3 4 5 6 7 8 9]" );
var AWrap = new MyWrapper( A );

var x = new XmlSerializer( typeof( MyWrapper ) );
FileStream s = File.Create( "myData.xml" );
x.Serialize( s, A );
s.Close();
```

Code Example – VB XML serialization

```
Imports System.IO
Imports System.Xml.Serialization
Imports MyNamespace

Dim A As New DoubleMatrix("3x3[1 2 3 4 5 6 7 8 9]")
Dim AWrap As New MyWrapper(A)

Dim X As New XmlSerializer(GetType(MyWrapper))
Dim S As FileStream = File.Create("myData.xml")
```

```
X.Serialize(S, W)
X.Close()
```

To restore the object:

Code Example – C# XML serialization

```
s = File.OpenRead( "myData.xml" );
MyWrapper AWrap = (MyWrapper)x.Deserialize( s );
var A = new DoubleMatrix( AWrap.Rows, AWrap.Columns,
  AWrap.Data, AWrap.Storage );
```

Code Example – VB XML serialization

```
S = File.OpenRead("myData.xml")
Dim AWrap As MyWrapper = CType(X.Deserialize(S), MyWrapper)
Dim A As New DoubleMatrix(AWrap.Rows, AWrap.Columns,
  AWrap.Data, AWrap.Storage)
```

CHAPTER 52.
DATABASE INTEGRATION

The .NET platform defines a number of types in the `System.Data` namespace—such as **DataTable**, **DataRow**, **DataRowCollection**, and **DataView**—that enable you to define and manipulate in-memory tables of data. **NMath** provides convenience methods for creating ADO.NET objects from vectors and general matrices, and for creating vectors and matrices from database objects.

52.1 Creating ADO.NET Objects from Vectors and Matrices

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.

NOTE—Values are copied by all methods that create data tables.

For example, this code creates a data table of one column containing the values in a vector:

Code Example – C#

```
using System.Data;

var v = new FloatVector( "45.4 -0.032 99 2.34" );
DataTable table = v.ToDataTable();
```

Code Example – VB

```
Imports System.Data

Dim V As New FloatVector("45.4 -0.032 99 2.34")
Dim Table As DataTable = V.ToDataTable()
```

By default, the table is named `Table`. You can also pass a non-default table name to the `ToDataTable()` method. Thus, this code creates a data table named `MyMatrixTable` containing the values in a **DoubleMatrix**:

Code Example – C#

```
using System.Data;
```

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

Code Example – VB

```
Imports System.Data

Dim A As New DoubleMatrix(8, 5, 3.1415)
Dim Table As DataTable = A.ToDataTable("MyMatrixTable")
```

This code illustrates creating paired data tables containing the real and imaginary parts a **FloatComplexMatrix**:

Code Example – C#

```
using System.Data;

string s =
   "2 x 2 [ (4.54,9.78) (3.2,-4.78) (-4.32,2.23) (4.3234,-1.0) ]";
var A = new FloatComplexMatrix( s );

DataTable reals = A.ToRealDataTable( “RealParts” );
DataTable imags = A.ToImagDataTable( “ImaginaryParts” );
```

Code Example – VB

```
Imports System.Data

Dim S As String =
  "2 x 2 [ (4.54,9.78) (3.2,-4.78) (-4.32,2.23) (4.3234,-1.0) ]"
Dim A As New FloatComplexMatrix(S)

Dim Reals As DataTable = A.ToRealDataTable("RealParts")
Dim Images As DataTable = A.ToImagDataTable("ImaginaryParts")
```

By default, the columns in a data table created from a vector or matrix are named `column1`, `column2`, and so on. If you wish to specify non-default column names, call `Columns()` on the returned **DataTable** object to obtain a **DataColumnCollection**, then iterate over the collection and set the `ColumnName` property on each **DataColumn** object to the desired name.

52.2 Creating Vector and Matrices from ADO.NET Objects

You can 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. Complex number vector and matrix class constructors accept paired database objects containing the real and imaginary parts, respectively.

For example, assuming `table` is a **DataTable** instance:

Code Example – C#

```
var A = new DoubleMatrix( table );
```

Code Example – VB

```
Dim A As New DoubleMatrix(Table)
```

The resulting matrix has the same number of rows and columns as the data table. Note that all values must be able to be converted to a double through a cast. If not, the constructor throws an **InvalidCastException**.

Similarly, assuming `view1` and `view2` are **DataView** objects, this code creates a **FloatComplexVector** instance whose real parts are derived from the first column of `view1`, and whose imaginary parts are derived from the first column of `view2`:

Code Example – C#

```
var v = new FloatComplexVector( view1, view2 );
```

Code Example – VB

```
Dim V As New FloatComplexVector(View1, View2)
```

In this case, all values must be able to be converted to a float through a cast.

Chapter 53.

Error Handling

All exceptions in **NMath** inherit from the **NMathException** class, enabling you to easily catch all **NMath** exceptions. This chapter lists the exception classes and the conditions under which they are thrown.

53.1 Exception Types

The following exception classes inherit from **NMathException**.

Table 28 – Exception classes

Exception	Description
FFTKernelException	Thrown when MKL returns an error condition when computing an FFT.
IndexOutOfRangeException	Thrown when an out of range index is passed to an **NMath** function.
InvalidArgumentException	Thrown when an invalid argument is passed to an **NMath** function.
InvalidBinBdryException	Thrown when a histogram operation results in invalid bin boundaries.
KernelLoadException	Thrown when **NMath** cannot load a kernel assembly.
MatrixNotSquareException	Thrown when a matrix operation requiring a square matrix is presented with a non-square one.
MismatchedSizeException	Exception thrown when an operation is performed with operands whose sizes are incompatible with the operation; for example, if you try to add two vectors with different lengths, or take the inner product of matrices A and B when the number of columns of A is not equal to the number of rows of B.

Table 28 – Exception classes

Exception	Description
NMathFormatException	Thrown when a method encounters a faulty text representation; for example, if you try to create a vector from a string that has an invalid format.
SingularMatrixException	Thrown when a matrix operation requiring a non-singular matrix is presented with a singular one.

For example, this code attempts to multiply two matrices with different dimensions, and catches a **MismatchedSizeException**:

Code Example – C#

```
DoubleComplexMatrix A =
   new DoubleComplexMatrix( 3, 3, new DoubleComplex(1,0) );
DoubleComplexMatrix B =
   new DoubleComplexMatrix( 2, 2, new DoubleComplex(1,0) );

DoubleComplexMatrix C;
try
{
   C = A * B;
}
catch( MismatchedSizeException e )
{
   Console.WriteLine( "Oops - " + e.Message );
}
```

Code Example – VB

```
Dim A As New DoubleComplexMatrix(3, 3, New DoubleComplex(1, 0))
Dim B As New DoubleComplexMatrix(2, 2, New DoubleComplex(1, 0))

Dim C As DoubleComplexMatrix
Try
  C = A * B
Catch E As MismatchedSizeException
  Console.WriteLine("Oops - " & E.Message)
End Try
```

Matrices must have the same dimensions to be combined using the element-wise operators.

INDEX

E

G

M

N

O

S

T

U

V

W