NMath Analysis User's Guide

Version 2.0

CenterSpace Software
Corvallis, Oregon

NMath Analysis User's Guide

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

The correct bibliographic reference for this document is:
NMath Analysis User's Guide, *Version 2.0*, CenterSpace Software, Corvallis, OR.

Printed in the United States.
Printing Date: April, 2009

CenterSpace Software

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

CONTENTS

CHAPTER 1. INTRODUCTION

Welcome to the *NMath Analysis User's Guide*.

NMath Analysis™ is part of CenterSpace Software's **NMath**™ product suite, which provides object-oriented components for mathematical, engineering, scientific, and financial applications on the .NET platform. **NMath Analysis** is an advanced function manipulation library that extends the general analysis classes of **NMath Core** to include function minimization, root-finding, and linear programming.

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

1.1 Product Features

The features of **NMath Analysis** include:

- 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.
- Linear programming using the simplex method.
- Least squares polynomial fitting.
- Nonlinear least squares minimization, curve fitting, and surface fitting.
- Classes for finding roots of univariate functions using the secant method, Ridders' method, and the Newton-Raphson method.
- Numerical methods for double integration of functions of two variables.

1.2 Software Requirements

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

- **NMath Analysis** depends on **NMath Core**, the foundational library in the **NMath** product suite. **NMath Core** must be installed on your system prior to building or executing **NMath Analysis** code.
- To use the **NMath Analysis** library, you need the Microsoft .NET Framework installed on your system. The .NET Framework is available without cost from:

  `http://msdn.microsoft.com/netframework/`
- 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`

1.3 Namespaces

All types in **NMath Analysis** are in the `CenterSpace.NMath.Analysis` namespace. To avoid using fully qualified names, preface your code with an appropriate namespace statement. For example, in C#:

```
using CenterSpace.NMath.Analysis;
```

In Visual Basic.NET:

```
Imports CenterSpace.NMath.Analysis
```

All **NMath Analysis** code shown in this manual assumes the presence of such a namespace statement.

NOTE—In most cases, you must also preface your code with a namespace statement for the CenterSpace.NMath.Core namespace.

1.4 Documentation

NMath Analysis includes the following documentation:

- The *NMath Analysis 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 Analysis User's Guide* is installed in:

  installdir`/Docs/NMath.Analysis.UsersGuide.pdf`

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

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

- The *NMath Analysis Reference*

  This document contains complete API reference documentation in compiled HTML Help format, enabling you to browse the **NMath Analysis** library just like the .NET Framework Class Library. The *NMath Analysis Reference* is installed in:

  installdir`/Docs/NMath.Analysis.Reference.chm`

  NOTE—Links to types in the .NET Framework will be broken unless you have the .NET Framework installed on your machine.

  HTML reference documentation may be viewed online using your browser at:

  `http://www.centerspace.net/doc/NMath/Suite/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 use C#; a few are shown in Visual Basic.NET. However, all **NMath Analysis** routines are callable from any .NET language.

This manual uses the following typographic conventions:

Table 1 – Typographic conventions

Convention	Purpose	Example
`Courier`	Function names, code, directories, file names, examples, and operating system commands.	`Minimize()` the `Assemblies` directory
italic	Conventional uses, such as emphasis and new terms.	...the *Newton-Raphson Method*
bold	Class names, product names, and commands from an interface.	**DownhillSimplexMinimizer** **NMath Analysis** Click **OK**.

1.5 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.

CHAPTER 2.

ENCAPSULATING MULTIVARIATE FUNCTIONS

NMath Core includes classes for encapsulating univariate functions, including base class **OneVariableFunction**, and derived types **Polynomial** and **TabulatedFunction**. For more information on these types, see the *NMath Core User's Guide*.

NMath Analysis extends the function encapsulation of **NMath Core** by providing class **MultiVariableFunction**, which encapsulates an arbitrary function of one or more variables, and works with other **NMath Analysis** classes to approximate integrals and minima.

This chapter describes how to create and manipulate **MultiVariableFunction** function objects.

2.1 Creating Multivariate Functions

A **MultiVariableFunction** is constructed from an `NMathFunctions.DoubleVectorDoubleFunction`, a delegate that takes a single **DoubleVector** parameter and returns a `double`. For example, suppose you wish to encapsulate this function:

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

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

```
NMathFunctions.DoubleVectorDoubleFunction d =
  new NMathFunctions.DoubleVectorDoubleFunction( MyFunction );
```

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

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

An `NMathFunctions.DoubleVectorDoubleFunction` is also implicitly converted to a **MultiVariableFunction**. Thus:

```
MultiVariableFunction f = d;
```

Class **MultiVariableFunction** provides a `Function` property that gets the encapsulated function delegate after construction.

2.2 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:

```
DoubleVector point = new DoubleVector( 0.0, 1.0, 0.0, -1.0 );
double z = f.Evaluate( point );
```

2.3 Algebraic Manipulation of Multivariate Functions

NMath Analysis 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 2 lists the equivalent operators and methods.

Table 2 – 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:

```
MultiVariableFunction g = f/2;
MultiVariableFunction sum = f + g;
MultiVariableFunction neg = -f;
```

This Visual Basic.NET code uses the equivalent named methods:

```
Dim g As MultiVariableFunction =
  MultiVariableFunction.Divide( f, 2 );
Dim sum As MultiVariableFunction =
  MultiVariableFunction.Add( f, g );
Dim neg As MultiVariableFunction =
  MultiVariableFunction.Negate( f );
```

CHAPTER 3. MINIMIZING UNIVARIATE FUNCTIONS

NMath Analysis 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 Analysis** 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.

3.1 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 Analysis** 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**:

```
Bracket bracket = new Bracket( function, 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.

3.2 Minimizing Functions Without Calculating the Derivative

NMath Analysis 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:

```
int maxIter = 50;
GoldenMinimizer minimizer = 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 3.1). For example, the function

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

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

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

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

This code finds a minimum of *f* in the interval `(0,2)` using golden section search:

```
GoldenMinimizer minimizer = new GoldenMinimizer();
int lower = 0;
int upper = 2;
double min = 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*:

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

3.3 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 3.2, *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 3.2), 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:

```
DBrentMinimizer minimizer = new DBrentMinimizer();
```

This code uses an error tolerance of `10^-4` and a maximum of `50` iterations:

```
double tol = 1e-4
int maxIter = 50;
DBrentMinimizer minimizer = 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 3.1). 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:

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

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

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

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

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

CHAPTER 4. MINIMIZING MULTIVARIATE FUNCTIONS

NMath Analysis 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 3, 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.

4.1 Minimizing Functions Without Calculating the Derivative

NMath Analysis 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.[1] A simplex in

[1] J. A. Nelder and R. Mead (1965), "A Simplex Method for Function Minimization," Computer Journal, Vol. 7, p. 308-313.

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).

- 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 3.2) 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:

```
int maxIter = 20;
PowellMinimizer minimizer = 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 2) of three variables, this code minimizes the function using the downhill simplex method, starting at the origin:

```
DownhillSimplexMinimizer minimizer =
  new DownhillSimplexMinimizer();
DoubleVector start = new DoubleVector( 0.0, 0.0, 0.0 );
DoubleVector min = 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. If you provide only a starting point to the `Minimize()` method, as illustrated above, the starting direction set is simply the unit vectors. An overload of `Minimize()` also enables you to specify the initial direction set.

4.2 Minimizing Derivable Functions

NMath Analysis 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 Analysis** 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:

```
double tol = 1e-5;
int maxIter = 20;
VariableMetricMinimizer minimizer =
  new VariableMetricMinimizer( tol, maxIter );
```

Class **ConjugateGradientMinimizer** and **VariableMetricMinimizer** implement the **IMultiVariableDMinimizer** interface, which provides a single `Minimize()` method with the following signature:

```
DoubleVector Minimize( MultiVariableFunction f,
                       MultiVariableFunction[] df,
                       DoubleVector x );
```

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:

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

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

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

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

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

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

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

CHAPTER 5. SIMULATED ANNEALING

In **NMath Analysis**, 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**.

5.1 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.

5.2 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 3 has four steps.

Table 3 – 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 Analysis**, **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 3:

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

LinearAnnealingSchedule schedule =
  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:

```
int[] iterations = new int[] { 50, 30, 20, 20 };
double[] temps = new double[] { 75.3, 20.0, 10.5, 0.0 };

CustomAnnealingSchedule schedule =
  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.

5.3 Minimizing Functions by Simulated Annealing

Instances of **AnnealingMinimizer** are constructed from an annealing schedule (Section 5.2). For instance:

```
AnnealingScheduleBase schedule =
  new LinearAnnealingSchedule( 5, 25, 100.0 );;
AnnealingMinimizer minimizer = 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 2) of five variables, this code minimizes the function using the downhill simplex method, starting at `(0.2,0.2,-.2,0.0,0.0)`:

```
AnnealingMinimizer minimizer = new AnnealingMinimizer( schedule );
DoubleVector start = new DoubleVector( 0.2, 0.2, -0.2, 0.0, 0.0 );
DoubleVector min = 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 5.4).

5.4 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:

```
AnnealingMinimizer minimizer = 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:

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

```
AnnealingHistory history = minimizer.AnnealingHistory;
foreach( AnnealingHistory.Step step in history )
{
  Console.WriteLine( step );
}
```

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:

```
Console.WriteLine( history[3] );
```

CHAPTER 6. LINEAR PROGRAMMING

In **NMath Analysis**, class **SimplexLPSolver** solves linear programming problems using the simplex method. 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
```

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.

This chapter describes how to use class **SimplexLPSolver**.

6.1 Solving LP Problems

Instances of **SimplexLPSolver** are constructed by specifying an error tolerance, or by accepting the default tolerance. For example, this code constructs a **SimplexLPSolver** with an error tolerance of 10^{-8}:

```
SimplexLPSolver solver = new SimplexLPSolver( 1e-8 );
```

The `Solve()` method on **SimplexLPSolver** accepts a vector of coefficients representing the objective function, a matrix of constraint coefficients, a vector of right-hand sides, the number of each constraint type (less than, greater than, or

equal to), and optionally vectors of lower and upper variable bounds. For example, this code solves the LP problem shown above:

```
// the objective function
DoubleVector obj = new DoubleVector( "[1 4 9]" );

// the constraints
DoubleMatrix constraints =
  new DoubleMatrix( "3x3 [1 1 0  1 0 1  0 -1 1]" );
DoubleVector rhs = new DoubleVector( "[5 10 7]" );
int numLessThan = 1;
int numGreaterThan = 1;
int numEqual = 1;

// the variable bounds
DoubleVector lowerBounds = new DoubleVector( "[0 0 0]" );
DoubleVector upperBounds = new DoubleVector( "[4 1 Infinity]" );

SimplexLPSolver solver = new SimplexLPSolver();
DoubleVector solution = solver.Solve( obj,
                                      constraints,
                                      rhs,
                                      numLessThan,
                                      numGreaterThan,
                                      numEqual,
                                      lowerBounds,
                                      upperBounds );
```

NOTE—All right-hand sides must be non-negative. The default bounds are 0 to PositiveInfinity. The bounds may be restricted further, but they must remain non-negative.

It is important to check the `Status` property after optimization, which returns a value from the `SimplexLPSolver.SolutionStatus` enumeration, since your problem may be unbounded or infeasible. The `IsGood` property returns `true` if `Status == SolutionStatus.FiniteSolutionFound`. Thus:

```
if ( solver.IsGood )
{
  // ...
}
```

If a finite solution was found, you can access the solution using the `Solution` property. (The solution is also returned by the `Solve()` method). The `OptimalValue` property gets the value of the objective function evaluated at the solution.

CHAPTER 7. FITTING POLYNOMIALS

NMath Core includes classes for calculating least squares fits of linear functions to a set of points. For instance, this code uses class **DoubleLeastSquares** to calculate the slope and intercept of a linear least squares fit through five data points, then prints out the properties of the solution:

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

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

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

For more information on linear least squares fitting, see the *NMath Core User's Guide*.

NMath Analysis extends the least squares functionality of the **NMath Suite** by providing class **PolynomialLeastSquares**, which performs a least squares fit of a **Polynomial** to a set of points.

This chapter describes how to use class **PolynomialLeastSquares**.

7.1 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:

```
int degree = 4;
PolynomialLeastSquares fit =
  new PolynomialLeastSquares( degree, x, y );
```

7.2 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:

```
Console.WriteLine( fit.CoeffErrorEstimate(0.01) );
```

CHAPTER 8.
NONLINEAR LEAST SQUARES

NMath Analysis 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 Analysis** uses the Trust-Region method, a variant of the Levenberg-Marquardt method.

NMath Analysis provides nonlinear least squares classes for:

- solving nonlinear least squares problems, with or without linear boundary constraints
- 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.

8.1 Trust-Region Minimization

NMath Analysis provides class **TrustRegionMinimizer** for solving both constrained and unconstrained nonlinear least squares problems using the Trust-Region method. 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:

```
int iter = 1000;
TrustRegionMinimizer minimizer = new TrustRegionMinimizer( iter );
```

Minimization

Class **TrustRegionMinimizer** provides the `Minimize()` method for minimizing a given function. Functions may be multidimensional in both their domain, *x*, and range, *y*.

To encapsulate such functions, **NMath** uses either:

- Delegate `NMathFunctions.DoubleArrayDoubleArrayFunction`, which takes an array of `double` and returns an array of `double`; or
- Delegate `NMathFunctions.DoubleVectorDoubleVectorFunction`, which takes a **DoubleVector** and returns a **DoubleVector**.

For example, this code encapsulates a function that has three input variables and four output variables:

```
public DoubleVector MyFunction( DoubleVector x )
{
  DoubleVector y = new DoubleVector( 4 );
  y[0] = 5 * x[1] * x[1] + x[2] * x[2];
  y[1] = 4 * x[0] * x[0] - x[2] * x[2] + 45;
  y[2] = x[0] * 3 * x[0] - x[1] * x[1] + 9;
  y[3] = y[0] + y[1] + y[2] * y[2] - 43;
  return y;
}

NMathFunctions.DoubleVectorDoubleVectorFunction f = new
  NMathFunctions.DoubleVectorDoubleVectorFunction( MyFunction );
```

The `Minimize()` method takes:

- the function, *f*, to minimize
- the starting point (from which the dimensionality of *x* can be determined)
- the dimensionality of *y*
- (optionally) an array of delegates encapsulating the partial derivatives of *f*.
- (optionally) a lower bound on the solution
- (optionally) an upper bound 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 given function, starting at the specified point:

```
NMathFunctions.DoubleVectorDoubleVectorFunction f = new
  NMathFunctions.DoubleVectorDoubleVectorFunction( MyFunction );
int ydim = 4;
DoubleVector start = new DoubleVector( 50.0, 30.0, -1000.0 );

TrustRegionMinimizer minimizer = new TrustRegionMinimizer();
DoubleVector solution = minimizer.Minimize( f, start, ydim );
```

Since problems can have multiple local minima, trying different starting points is recommended for better solutions.

Partial Derivatives

The Trust-Region algorithm requires the partial derivatives of the function being minimized. A numerical approximation is used by default, but if the closed form partial derivatives are available, they can be given to the `Minimize()` method.

For example, this code encapsulates a function that has two input variables and two output variables, encapsulates the array of four partial derivatives, then calls `Minimize()`:

```
NMathFunctions.DoubleVectorDoubleVectorFunction f =
  delegate( DoubleVector x )
  {
    DoubleVector y = new DoubleVector( 2 );
    y[0] = x[0] * x[1] + 1.0;
    y[1] = Math.Pow( x[0], 3 ) + 2 * x[1] + 2.0;
    return y;
  };

NMathFunctions.DoubleVectorDoubleVectorFunction[] df =
  new NMathFunctions.DoubleVectorDoubleVectorFunction[ 4 ]
  {
    delegate( DoubleVector x )
    {
      DoubleVector y = new DoubleVector( 2 );
      y[0] = x[1];
      return y;
    },
    delegate( DoubleVector x )
    {
      DoubleVector y = new DoubleVector( 2 );
      y[0] = x[0];
      return y;
    },
    delegate( DoubleVector x )
    {
      DoubleVector y = new DoubleVector( 2 );
      y[1] = 3 * Math.Pow( x[0], 2 );
      return y;
    },
    delegate( DoubleVector x )
    {
      DoubleVector y = new DoubleVector( 2 );
      y[1] = 2;
      return y;
    }
  };

TrustRegionMinimizer minimizer = new TrustRegionMinimizer();
DoubleVector solution = minimizer.Minimize( f, start, ydim, df );
```

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 sets the lower bound to `(0,0,0)`, and the upper bounds to `(1,1,1)`:

```
DoubleVector lowerBounds = new DoubleVector( “0 0 0” );
DoubleVector upperBounds = new DoubleVector( “1 1 1” );

DoubleVector solution = minimizer.Minimize( f, start, ydim,
  lowerBounds, upperBounds );
```

Minimization Results

The `Minimize()` method returns the solution found by the minimization:

```
DoubleVector solution = minimizer.Minimize( f, start, ydim );
```

Additional information about the last performed fit is available from properties on `TrustRegionMinimizer`:

- `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.
- `MaxIterationsMet` returns `true` if the minimum just computed stopped because the maximum number of iterations was reached; otherwise, `false`.
- `StopCriterion` returns the reason for stopping.

For example:

```
double initialResidual = minimizer.InitialResidual;
double finalResidual = minimizer.FinalResidual;
int iterations = minimizer.Iterations;
TrustRegionMinimizer.Criterion stopCriterion =
  minimizer.StopCriterion;
```

The stopping criterion is returned as a value from the `TrustRegionMinimizer.Criterion` enumeration, shown in Table 4.

Table 4 – 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`:

```
minimizer.ToleranceTrialStep = 1e-12;
```

The `SetAllTolerances()` method can be used after construction to set all tolerances to the same value.

8.2 Nonlinear Least Squares Curve Fitting

NMath Analysis provides class **OneVariableFunctionFitter** for fitting *generalized* one variable functions to a set of points. You must supply at least as many data points to fit as your function has parameters.

In the space of the function parameters, beginning at a specified starting point, class **OneVariableFunctionFitter** finds a minimum (possibly local) in the sum of the squared residuals with respect to a set of data points. Minimization is performed using the Trust-Region method (Section 8.1).

The Trust-Region method requires the partial derivatives of the function, but a numerical approximation may be used if the closed form is not available.

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 $y = a\sin(bx + c)$:

```
public double MyFunction( DoubleVector p, double x )
{
  return p[0] * Math.Sin( p[1] * x + p[2] );
}
```

Delegate **NMathFunctions.GeneralizedDoubleUnaryFunction** encapsulates generalized one variable functions:

```
NMathFunctions.GeneralizedDoubleUnaryFunction f =
  new NMathFunctions.GeneralizedDoubleUnaryFunction( MyFunction );
```

Predefined Functions

For convenience, class **AnalysisFunctions** includes a selection of common generalized one variable functions, as shown in Table 5.

Table 5 – Predefined Generalized One Variable Functions

Delegate	Function
TwoParameterAsymptoticFunction	$y = a + \frac{b}{x}$
ThreeParameterExponentialFunction	$y = ae^{bx} + c$
ThreeParameterSineFunction	$y = a\sin(bx + c)$

Table 5 – Predefined Generalized One Variable Functions

Delegate	Function
FourParameterLogisticFunction	$y = d + \frac{a - d}{1 + \left(\frac{x}{c}\right)^b}$
FiveParameterLogisticFunction	$y = d + \frac{a - d}{\left[1 + \left(\frac{x}{c}\right)^b\right]^g}$

For example:

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

Constructing a OneVariableFunctionFitter

Instances of **OneVariableFunctionFitter** are constructed from a delegate encapsulating the generalized one variable function to fit, and optionally an array of delegates containing the partial derivatives of the function with respect to each parameter. For example, this code uses one of the predefined curves in **AnalysisFunctions**:

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

NMathFunctions.GeneralizedDoubleUnaryFunction[] df =
  AnalysisFunctions.FourParameterLogisticPartialDerivatives;

OneVariableFunctionFitter fitter =
  new OneVariableFunctionFitter( f, df );
```

This code defines a new curve, and the partial derivatives, then constructs a **OneVariableFunctionFitter** instance:

```
NMathFunctions.GeneralizedDoubleUnaryFunction f =
  delegate( DoubleVector p, double x )
  {
    // y = a(x^2 + bx)/(x^2 + cx + d)
    double x2 = x * x;
    return (p[0] * (x2 + p[1] * x)) / (x2 + p[2] * x + p[3]);
  };
```

```
NMathFunctions.GeneralizedDoubleUnaryFunction[] df =
  new NMathFunctions.GeneralizedDoubleUnaryFunction[4]
  {
    delegate(DoubleVector p, double x)
    {
      double x2 = x * x;
      return (x2 + p[1] * x) / (x2 + p[2] * x + p[3]);
    },
    delegate(DoubleVector p, double x)
    {
      double x2 = x * x;
      return (p[0] * x) / (x2 + p[2] * x + p[3]);
    },
    delegate(DoubleVector p, double x)
    {
      double x2 = x * x;
      return (-p[0] * x * (x2 + p[1] * x)) /
             (Math.Pow((x2 + p[2] * x + p[3]), 2 ));
    },
    delegate(DoubleVector p, double x)
    {
      double x2 = x * x;
      return -(p[0] * (x2 + p[1] * x)) /
             (Math.Pow((x2 + p[2] * x + p[3]), 2));
    }
  };

OneVariableFunctionFitter fitter =
  new OneVariableFunctionFitter( f, df );
```

If the partial derivatives are not available, they can be omitted, and a numerical approximation is used instead:

```
OneVariableFunctionFitter fitter =
  new OneVariableFunctionFitter( f );
```

Fitting Data

Once you've constructed an instance of **OneVariableFunctionFitter** containing a curve, you can fit that curve to a set of points using the `Fit()` method. The `Fit()` method takes vectors of *x* and *y* values representing the data points, and a starting position in the function parameter space. For instance:

```
DoubleVector 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 );
```

```
DoubleVector 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 );

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

DoubleVector solution = 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.

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:

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

To compute the residuals relative to the data points at the solution, use the `ResidualVector()` method:

```
DoubleVector residuals = fitter.ResidualVector( x, y, solution );
```

Additional information about the last performed fit is available from the underlying **TrustRegionMinimizer** 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:

```
TrustRegionMinimizer minimizer = fitter.Minimizer;

double initialResidual = minimizer.InitialResidual;
double finalResidual = minimizer.FinalResidual;
int iterations = minimizer.Iterations;
TrustRegionMinimizer.Criterion stopCriterion =
  minimizer.StopCriterion;
```

8.3 Nonlinear Least Squares Surface Fitting

NMath Analysis provides class **MultiVariableFunctionFitter** for fitting *generalized* multivariable functions to a set of points. The interface is analogous to **OneVariableFunctionFitter** (Section 8.2), 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.

As in curve fitting, **MultiVariableFunctionFitter** uses the Trust-Region method to find a minimum (possibly local) in the space of the function parameters in the sum of the squared residuals with respect to a set of data points The Trust-Region method requires the partial derivatives of the function, but a numerical approximation may be used if the closed form is not available.

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 $y = ax_1^2x_2 + b\sin(x_1) + cx_2^3$:

```
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 );
};
```

Delegate **NMathFunctions.GeneralizedDoubleVectorDoubleFunction** encapsulates generalized multivariable functions:

```
NMathFunctions.GeneralizedDoubleVectorDoubleFunction f =
  new NMathFunctions.GeneralizedDoubleVectorDoubleFunction(
    MyFunction );
```

Constructing a MultiVariableFunctionFitter

Instances of **MultiVariableFunctionFitter** are constructed from a delegate encapsulating the generalized multivariable function to fit, and optionally an array of delegates containing the partial derivatives of the function with respect to each

parameter. For example, this code defines a surface, and the partial derivatives, then constructs a **MultiVariableFunctionFitter** instance:

```
NMathFunctions.GeneralizedDoubleVectorDoubleFunction f =
  delegate( DoubleVector p, DoubleVector x )
  {
    // z = ayx^2 + bsin(x) + cy^3
    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 );
  };

NMathFunctions.GeneralizedDoubleVectorDoubleFunction[] df =
  new NMathFunctions.GeneralizedDoubleVectorDoubleFunction[3]
  {
    delegate(DoubleVector p, DoubleVector x)
    {
       return Math.Pow(x[0], 2.0) * x[1];
    },
    delegate(DoubleVector p, DoubleVector x)
    {
      return Math.Sin(x[0]);
    },
    delegate(DoubleVector p, DoubleVector x)
    {
      return Math.Pow(x[1], 3.0);
     ,
  };

MultiVariableFunctionFitter fitter =
  new MultiVariableFunctionFitter( f, df );
```

If the partial derivatives are not available, they can be omitted, and a numerical approximation is used instead:

```
MultiVariableFunctionFitter fitter =
  new MultiVariableFunctionFitter( f );
```

Fitting Data

Once you've constructed an instance of **MultiVariableFunctionFitter** containing a curve, you can fit that curve to a set of points using the `Fit()` method. The `Fit()` method 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:

```
DoubleMatrix 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");

DoubleVector y = new DoubleVector("95.09 23.11 60.63 48.59 89.12
                                   76.97 45.68 1.84 82.17 44.47");

DoubleVector start = new DoubleVector("10 10 10");

DoubleVector solution = 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.

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:

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

To compute the residuals relative to the data points at the solution, use the `ResidualVector()` method:

```
DoubleVector residuals = fitter.ResidualVector( x, y, solution );
```

Additional information about the last performed fit is available from the underlying **TrustRegionMinimizer** 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:

```
TrustRegionMinimizer minimizer = fitter.Minimizer;

double initialResidual = minimizer.InitialResidual;
double finalResidual = minimizer.FinalResidual;
int iterations = minimizer.Iterations;
TrustRegionMinimizer.Criterion stopCriterion =
  minimizer.StopCriterion;
```

CHAPTER 9.

FINDING ROOTS OF UNIVARIATE FUNCTIONS

NMath Analysis 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 Analysis** root-finding classes derive from the abstract base class **RootFinderBase**. The interface and behavior is the same as for **MinimizerBase** (Section 3.1)—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.

9.1 Finding Function Roots Without Calculating the Derivative

NMath Analysis provides two 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.

Instances of **SecantRootFinder** and **RiddersRootFinder** 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:

```
int maxIter = 50;
SecantRootFinder finder = new SecantRootFinder( maxIter );
```

Instances of **SecantRootFinder** and **RiddersRootFinder** provide `Find()` methods for minimizing a given function within a given interval. For instance, the cosine function has a root at $\pi/2$:

```
OneVariableFunction f = new OneVariableFunction(
  new NMathFunctions.DoubleUnaryFunction( Math.Cos ) );

RiddersRootFinder finder = new RiddersRootFinder();
double lower = 0;
double upper = Math.PI;
double root = finder.Find( f, lower, upper );
```

9.2 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 9.1), 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:

```
double tol = 1e-8;
int maxIter = 100;
NewtonRalphsonRootFinder finder =
  new NewtonRalphsonRootFinder( 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)`:

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

CHAPTER 10. INTEGRATING MULTIVARIABLE FUNCTIONS

NMath Core includes classes for computing an approximation of the integral of a **OneVariableFunction** over some interval. These classes include **RombergIntegrator** and **GaussKronrodIntegrator**, which implement the **IIntegrator** interface. For more information on these types, see the *NMath Core User's Guide*.

NMath Analysis extends the quadrature functionality of the **NMath Suite** by providing class **TwoVariableIntegrator**, which computes the integral of a function of two variables. Class **TwoVariableIntegrator** computes the double integral by breaking up the problem into repeated one-dimensional integrals.

The chapter describes how to use class **TwoVariableIntegrator**.

10.1 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:

```
TwoVariableIntegrator integrator = new TwoVariableIntegrator();
```

Instances of **GaussKronrodIntegrator** are used by default. Alternatively, you can provide non-default univariate integrators:

```
GaussKronrodIntegrator gauss1 = new GaussKronrodIntegrator();
GaussKronrodIntegrator gauss2 = new GaussKronrodIntegrator();
gauss2.Tolerance = 1e-6;
TwoVariableIntegrator integrator =
  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.

10.2 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:

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

First write the function:

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

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

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

Finally, compute the integral:

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

```
private double F( DoubleVector v )
{
  return ( 9.0 * v[0] * v[0] ) - ( 3.0 * v[1] );
}
```

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

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

Then define the *y* bounding functions and encapsulate them as **OneVariableFunction** objects:

```
private double YUpperF( double x )
{
  return Math.Sqrt( 9.0 - ( x * x ) );
}

private double YLowerF( double x )
{
  return -YUpperF( x );
}

OneVariableFunction yLowerFunction = new OneVariableFunction(
  new NMathFunctions.DoubleUnaryFunction( YLowerF ) );
OneVariableFunction yUpperFunction = new OneVariableFunction(
  new NMathFunctions.DoubleUnaryFunction( YUpperF ) );
```

Finally, compute the integral:

```
TwoVariableIntegrator integrator = new TwoVariableIntegrator();
double xLower = -3;
double xUpper = 3;
double integral = integrator.Integrate( function, xLower, xUpper,
  yLowerFunction, yUpperFunction );
```

INDEX

V