Chapter No. 3
"Financial Mathematics and Numerical Analysis"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Financial Mathematics and Numerical Analysis"

A synopsis of the book’s content

Information on where to buy this book

About the Author

**Johan Astborg** is the developer and architect of various kinds of software systems and applications, financial software systems, trading systems, as well as mobile and web applications. He is interested in computer science, mathematics, and quantitative finance, with a special focus on functional programming. Johan is passionate about languages such as F#, Clojure, and Haskell, and operating systems such as Linux, Mac OS X, and Windows for his work. Most of Johan’s quantitative background comes from Lund University, where he studied courses in computer science, mathematics, and physics. Currently Johan is studying pure mathematics at Lund University, Sweden, and is aiming for a PhD in the future, combining mathematics and functional programming. Professionally, Johan has worked as a part-time developer for Sony Ericsson and various smaller firms in Sweden. He also works as a part-time consultant focusing on web technologies and cloud solutions. You can easily contact him by sending an e-mail to joastbg@gmail.com or visit his GitHub page at https://github.com/joastbg.

For More Information:

F# for Quantitative Finance

F# is a functional programming language that allows you to write simple code for complex problems. Currently, it is most commonly used in the financial sector. Quantitative finance makes heavy use of mathematics to model the real world. If you are interested in using F# for your day-to-day work or research in quantitative finance, this book is for you.

This book covers everything you need to know about using functional programming for quantitative finance. Using a functional programming language for quantitative finance will enable you to concentrate more on the model itself rather than the implementation details. Tutorials and snippets are summarized into a trading system throughout this book.

F#, together with .NET, provides a wide range of tools needed to produce high quality and efficient code, from prototyping to production. The example code snippets in this book can be extended into larger blocks of code, and reused and tested easily in a functional language. F# is considered one of the default functional languages of choice for financial and trading-related applications.

What This Book Covers

Chapter 1, Introducing F# Using Visual Studio, introduces you to F# and its roots in functional languages. You will learn how to use F# in Visual Studio and write your first application.

Chapter 2, Learning More About F#, teaches you more about F# as a language and illustrates the many sides of this paradigm language.

Chapter 3, Financial Mathematics and Numerical Analysis, introduces the toolset we'll need throughout the book to implement financial models and algorithms.

Chapter 4, Getting Started with Data Visualization, introduces some of the most common ways to use F# to visualize data and display information in a GUI.

Chapter 5, Learning Option Pricing, teaches you about options, the Black-Scholes formula and ways of exploring options using the tools at hand.

Chapter 6, Exploring Volatility, digs deeper into the world of Black-Scholes and teaches you about implied volatility.

For More Information:
Chapter 7, *Getting Started with Order Types and Market Data*, takes a rather pragmatic approach towards finance and implements a basic order management system.

Chapter 8, *Setting Up the Trading System Project*, builds the foundation for the project and shows how to connect to SQL Server and use LINQ for queries.

Chapter 9, *Trading Volatility for Profit*, studies various ways of monetizing through movements in volatility and the arbitrage opportunity defining the trading strategy for the project.

Chapter 10, *Putting the Pieces Together*, shows the final steps towards the complete trading system using a volatility arbitrage strategy and FIX 4.2.

For More Information:
In this chapter, the reader will be introduced to the basic numerical analysis and algorithm implementation in F#. We will look at how integer and floating-point numbers are implemented, and we will also look at their respective limitations. The basic statistics are covered, and the existing functions in F# are studied and compared with custom implementations.

This chapter will build up the foundation of numerical analysis that can be used when we look at option pricing and volatility later on. We'll also use some of the functionality covered in the previous chapter to implement the mathematical functions for aggregate statistics and to illustrate their usefulness in real life.

In this chapter, you will learn:

- Implementing algorithms in F#
- Numerical concerns
- Implementing basic financial equations
- Curve fitting and regression
- Matrices and vectors in F#

For More Information:
Understanding the number representation

In this section, we will show you how numbers are represented as integers or floating-point numbers in computers. Numbers form the foundation of computers and programming. Everything in a computer is represented by the binary numbers, ones and zeroes. Today, we have 64-bit computers that enable us to have a 64-bit representation of integers and floating-point numbers naively in the CPU. Let's take a deeper look at how integers and floating-point numbers are represented in the following two sections.

Integers

When we talk about integers, denoted as \( \mathbb{Z} \), we are talking specifically about machine-precision integers that are represented exactly in the computer with a sequence of bits. Also, an integer is a number that can be written without a fractional or decimal component and is denoted as \( \mathbb{Z} \) by convention. For example, 0 is represented as 000000..., 1 is represented as ...00001, 2 is represented as ...000010, and so on. As you can see from this pattern, numbers are represented in the power of two. To represent negative numbers, the number range is divided into two halves and uses two's complement.

When we talk about integer representation without any negative numbers, that is, numbers from zero and up, we talk about unsigned integers.

Two's complement

Two's complement is a way of dividing a range of binary numbers into positive and negative decimal numbers. In this way, both positive and negative numbers can be represented in the computer. On the other hand, this means that the range of numbers is the half for two's complement in relation to the unsigned representation. Two's complement is the main representation used for signed integers.
The representation of integers in two's complement can be thought of as a ring, as illustrated in the preceding figure. The overflow occurs when the maximum allowed positive or negative value increases. Overflow simply means that we pass the barrier between positive and negative numbers.

The following table shows some integers and the representation of their two's complement:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Two's complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>0111 1111</td>
</tr>
<tr>
<td>64</td>
<td>0100 0000</td>
</tr>
<tr>
<td>1</td>
<td>0000 0001</td>
</tr>
<tr>
<td>0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>-1</td>
<td>1111 1111</td>
</tr>
<tr>
<td>-64</td>
<td>1100 0000</td>
</tr>
<tr>
<td>-127</td>
<td>1000 0001</td>
</tr>
<tr>
<td>-128</td>
<td>1000 0000</td>
</tr>
</tbody>
</table>

As you can see, the range for the 8-bit signed integers is from -128 to -127. In more general terms:

\[-2^{n-1} \leq \text{R} \leq 2^{n-1}-1\]

**Floating-point numbers**

Floating-point numbers, denoted as R, represent a quantity where decimals are needed to define them. Another way of describing these numbers is to think of them as values represented as a quantity along a continuous line. They are needed to model things in real life, such as economic, statistical, and physical quantities. In the machine, floating-point numbers are represented by the IEEE 754 standard.

**The IEEE 754 floating-point standard**

The IEEE 754 floating-point standard describes floating-points using a mantissa and an exponent; see the following figure.

For More Information:

For example, a 64-bit floating point number is made up of the following bit pattern:

<table>
<thead>
<tr>
<th>Sign bit</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>11 bits</td>
<td>52 bits</td>
</tr>
</tbody>
</table>

\[
1.2345 = \frac{12345}{10^{4}}
\]

An example of floating-point numbers and their binary representations are shown in the following table:

<table>
<thead>
<tr>
<th>Binary representation</th>
<th>Floating-point number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000000000</td>
<td>0.0</td>
</tr>
<tr>
<td>0x3ff0000000000000</td>
<td>1.0</td>
</tr>
<tr>
<td>0xc000000000000000</td>
<td>-2.0</td>
</tr>
<tr>
<td>0x4000000000000000</td>
<td>2.0</td>
</tr>
<tr>
<td>0x402E000000000000</td>
<td>15.0</td>
</tr>
</tbody>
</table>

F# Interactive is capable of decoding the representations of floating-point numbers in hexadecimal format into floating-points:

```
> 0x402E000000000000LF;;
val it: float = 15.0
```

Try out the preceding binary representations in F# Interactive.

**Learning about numerical types in F#**

In F#, as in most other modern languages, there is a variety of numerical types. The main reason for this is to enable you, as a programmer, to choose the most appropriate numerical type at any given situation. Sometimes there is no need for a 64-bit integer as 8-bit will be enough for small numbers. Another aspect is memory efficiency and consumption, that is, 64-bit integers will consume eight times as much as 8-bit integers.
The following is a table with the most common numerical types used in the F# code. They come in two main varieties; integers and floating-point numbers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8-bit unsigned integers</td>
<td>10uy, 0xA0uy</td>
</tr>
<tr>
<td>sbyte</td>
<td>8-bit signed integers</td>
<td>10y</td>
</tr>
<tr>
<td>int16</td>
<td>16-bit signed integers</td>
<td>10s</td>
</tr>
<tr>
<td>uint16</td>
<td>16-bit unsigned integers</td>
<td>10us</td>
</tr>
<tr>
<td>int, int32</td>
<td>32-bit signed integers</td>
<td>10</td>
</tr>
<tr>
<td>uint32</td>
<td>32-bit unsigned integers</td>
<td>10u</td>
</tr>
<tr>
<td>int64</td>
<td>64-bit signed integers</td>
<td>10L</td>
</tr>
<tr>
<td>uint64</td>
<td>64-bit unsigned integers</td>
<td>10UL</td>
</tr>
<tr>
<td>nativeint</td>
<td>Hardware-sized signed integers</td>
<td>10n</td>
</tr>
<tr>
<td>unativeint</td>
<td>Hardware-sized signed integers</td>
<td>10un</td>
</tr>
<tr>
<td>single, float32</td>
<td>32-bit IEEE 754 floating-point</td>
<td>10.0f</td>
</tr>
<tr>
<td>double, float</td>
<td>64-bit IEEE 754 floating-point</td>
<td>10.0</td>
</tr>
<tr>
<td>decimal</td>
<td>High-precision decimal</td>
<td>10.0M</td>
</tr>
<tr>
<td>bigint</td>
<td>Arbitrary precision integers</td>
<td>10I</td>
</tr>
<tr>
<td>complex</td>
<td>Complex numbers using 64-bit floats</td>
<td>Complex(10.0, 10.0)</td>
</tr>
</tbody>
</table>

The following are some examples of how to use suffixes for integers:

```fsharp
> let smallestInteger = 10uy;;
val smallestInteger : byte = 10uy

> let smallerInteger = 10s;;
val smallerInteger : int16 = 10s

> let smallInteger = 10us;;
val smallInteger : uint16 = 10us

> let integer = 10L;;
val integer : int64 = 10L
```

For More Information:
Arithmetic operators

Arithmetic operators should be familiar to you; however, we'll cover them in this section for consistency. The operators work as expected, and the succeeding table illustrates this using an example for each one. The remainder operator that returns the remainder from an integer division is worth noticing.

Let's look at an example to see this in more detail. First, we try to divide 10 by 2, and the remainder is 0 as expected:

```plaintext
> 10 % 2;;
val it : int = 0
```

If we try to divide 10 by 3, we get 1 as the remainder, because $3 \times 3 = 9$, and $10 - 9 = 1$:

```plaintext
> 10 % 3;;
val it : int = 1
```

The following table shows arithmetic operators with examples and a description:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>x + y</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>x - y</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>x * y</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>x / y</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>x % y</td>
<td>Remainder</td>
</tr>
<tr>
<td>-</td>
<td>-x</td>
<td>Unary minus</td>
</tr>
</tbody>
</table>

Arithmetic operators do not check for overflow. If you want to check for overflow, you can use the Checked module. You can find more about the Checked module at http://msdn.microsoft.com/en-us/library/vstudio/ee340296.aspx.

Learning about arithmetic comparisons

Arithmetic comparisons are used to compare two numbers for relationships. It's good to know all the operators that are shown in the following table:

For More Information:

Some examples of arithmetic comparisons are as follows:

```fsharp
> 5.0 = 5.0;;
val it : bool = true
> 1 < 4;;
val it : bool = true
> 1.0 > 3.0;;
val it : bool = false
```

It is also worth noticing that you can't compare numbers of different types in F#. To do this, you have to convert one of them as follows:

```fsharp
> 5.0 >= 10;;
5.0 >= 10
--------^^
stdin(10,8): error FS0001: This expression was expected to have type float but here has type int
```

## Math operators

The following table of mathematical operators covers the most basic mathematical functions that are expected to be found in a programming language or its standard libraries:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>abs x</td>
<td>Overloaded absolute value</td>
</tr>
<tr>
<td>acos</td>
<td>acos x</td>
<td>Overloaded inverse cosine</td>
</tr>
<tr>
<td>asin</td>
<td>asin x</td>
<td>Overloaded inverse sine</td>
</tr>
<tr>
<td>atan</td>
<td>atan x</td>
<td>Overloaded inverse tangent</td>
</tr>
<tr>
<td>ceil</td>
<td>ceil x</td>
<td>Overloaded floating-point ceil</td>
</tr>
</tbody>
</table>

For More Information:  
Financial Mathematics and Numerical Analysis

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cos</td>
<td>cos x</td>
<td>Overloaded cosine</td>
</tr>
<tr>
<td>exp</td>
<td>exp x</td>
<td>Overloaded exponent</td>
</tr>
<tr>
<td>floor</td>
<td>floor x</td>
<td>Overloaded floating-point floor</td>
</tr>
<tr>
<td>log</td>
<td>log x</td>
<td>Overloaded natural logarithm</td>
</tr>
<tr>
<td>log10</td>
<td>log10 x</td>
<td>Overloaded base-10 logarithm</td>
</tr>
<tr>
<td>**</td>
<td>x ** y</td>
<td>Overloaded exponential</td>
</tr>
<tr>
<td>pown</td>
<td>pown x y</td>
<td>Overloaded integer exponential</td>
</tr>
<tr>
<td>round</td>
<td>round x</td>
<td>Overloaded rounding</td>
</tr>
<tr>
<td>sin</td>
<td>sin x</td>
<td>Overloaded sine function</td>
</tr>
<tr>
<td>sqrt</td>
<td>sqrt x</td>
<td>Overloaded square root function</td>
</tr>
<tr>
<td>tan</td>
<td>tan x</td>
<td>Overloaded tangent function</td>
</tr>
</tbody>
</table>

**Conversion functions**

There are no implicit conversions in F# as conversions have to be done manually using conversion routines. Conversion must be made explicitly between types using the operators that are described in the following table:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>byte x</td>
<td>Overloaded conversion to a byte</td>
</tr>
<tr>
<td>sbyte</td>
<td>sbyte x</td>
<td>Overloaded conversion to a signed byte</td>
</tr>
<tr>
<td>int16</td>
<td>int16</td>
<td>Overloaded conversion to a 16-bit integer</td>
</tr>
<tr>
<td>uint16</td>
<td>uint16</td>
<td>Overloaded conversion to an unsigned 16-bit integer</td>
</tr>
<tr>
<td>int32, int</td>
<td>int32 x , int x</td>
<td>Overloaded conversion to a 32-bit integer</td>
</tr>
<tr>
<td>uint32</td>
<td>uint32</td>
<td>Overloaded conversion to an unsigned 32-bit integer</td>
</tr>
<tr>
<td>int64</td>
<td>int64 x</td>
<td>Overloaded conversion to a 64-bit integer</td>
</tr>
<tr>
<td>uint64</td>
<td>uint64 x</td>
<td>Overloaded conversion to an unsigned 64-bit integer</td>
</tr>
<tr>
<td>nativeint</td>
<td>nativeint x</td>
<td>Overloaded conversion to a native integer</td>
</tr>
<tr>
<td>unativeint</td>
<td>unativeint x</td>
<td>Overloaded conversion to an unsigned native integer</td>
</tr>
<tr>
<td>float, double</td>
<td>float x, double x</td>
<td>Overloaded conversion to a 64-bit IEEE floating-point number</td>
</tr>
<tr>
<td>float32, single</td>
<td>float32 x, single x</td>
<td>Overloaded conversion to a 32-bit IEEE floating-point number</td>
</tr>
</tbody>
</table>

For More Information:
This means that there will never be any automatic type conversion behind the scenes that may lead to loss of precision. For example, numbers are not converted from floating-points to integers just to fit the code that you have written. The compiler will tell you that there is an error in the code before converting it (it will never be converted by the compiler). The positive side about this is that you always know the representation of your numbers.

**Introducing statistics**

In this section, we'll look at statistics using both built-in functions and simple custom ones. Statistics are used a lot throughout quantitative finance. Larger time series are often analyzed, and F# has great support for sequences of numbers; some of its power will be illustrated in the examples mentioned in this section.

**Aggregate statistics**

Aggregated statistics is all about statistics on aggregated data such as sequences of numbers collected from measurements. It's useful to know the average value in such a collection; this tells us where the values are centered. The min and max values are also useful to determine the extremes in the collection.

In F#, the Seq module has this functionality built-in. Let's take a look at how to use it in each case using an example.

**Calculating the sum of a sequence**

Consider a sequence of 100 random numbers:

```fsharp
let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 100 -> rnd()]
```

We can calculate the sum of the preceding sequence, `data`, using the pipe operator and then the module function `Seq.sum`:

```fsharp
let sum = data |> Seq.sum
```
Note that the result, \( \text{sum} \), will vary from time to time due to the fact that we use a random number generator:

```fsharp
> sum;;
val it : float = 42.65793569
```

You might think that the \( \text{sum} \) function is not the most useful one; but there are times you need it, and knowing about its existence in the module library will save you time.

### Calculating the average of a sequence

For this example, let's modify the random seed function a bit to generate 500 numbers between 0 and 10:

```fsharp
let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 500 -> rnd() * 10.0]
```

The expected value of this sequence is 5 because of the distribution of the random function:

```fsharp
let avg = data |> Seq.average
```

The value may vary a bit due to the fact that we generate numbers randomly:

```fsharp
> avg;;
val it : float = 4.983808457
```

As expected, the average value is almost 5. If we generate more numbers, we will soon come closer and closer to the theoretical expected value of 5:

```fsharp
let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 10000 -> rnd() * 10.0]
let avg = data |> Seq.average

> avg;;
val it : float = 5.006555917
```

### Calculating the minimum of a sequence

Instead of iterating the sequence and keeping track of the minimum value using some kind of a loop construct with a temporary variable, we lend ourselves towards the functional approach in F#. To calculate the minimum of a sequence, we use the module function `Seq.min`:

```fsharp
let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 10000 -> rnd() * 10.0]
let avg = data |> Seq.average

> avg;;
val it : float = 5.006555917
```
let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 10 -> rnd() * 10.0]

val data : float list =
    [5.0530272; 6.389536232; 6.126554094; 7.276151291; 0.9457452972;
     7.774030933; 7.654594368; 8.517372011; 3.924642724; 6.572755164]

let min = data |> Seq.min
> min;;
val it : float = 0.9457452972

This looks a lot like the preceding code seen, except that we generate 10 random
numbers and inspect the values in the list. If we manually look for the smallest value
and then compare it to the one calculated by F#, we see that they match.

Calculating the maximum of a sequence
In the following example, we'll use Seq.max to calculate the maximum number
of a list:

let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 5 -> rnd() * 100.0]

val data : float list =
    [7.586052086; 22.3457242; 76.95953826; 59.31953153; 33.53864822]

let max = data |> Seq.max
> max;;
val it : float = 76.95953826

Calculating the variance and standard deviation
of a sequence
So far, we have already used the existing functions for our statistical analysis.
Now, let's implement variance and standard deviation.
Calculating variance
Let's use the following function and calculate the variance for a dice:

```fsharp
let variance(values=  
let average = Seq.average values  
values  
|> Seq.map (fun x -> (1.0 / float (Seq.length values)) * (x -  
    average) ** 2.0)  
|> Seq.sum
```

A dice has six discrete outcomes, one to six, where every outcome has equal probability. The expected value is 3.5, \((1 + 2 + 3 + 4 + 5 + 6)/6\). The variance of a dice is calculated using the following function:

```fsharp
> variance [1.0 .. 6.0];;
val it : float = 2.916666667
```

Calculating standard deviation
We start by implementing the standard deviation function using the previously defined variance function. According to statistics, standard deviation is the square root of the variance:

```fsharp
let stddev1(values:seq<float>) = sqrt(variance(values))
```

The preceding function works just fine, but to illustrate the power of sequences, we will implement the standard deviation using the fold function. The fold function will apply a given function to every element and accumulate the result. The 0.0 value in the end just means that we don't have an initial value. You may remember this from the section about fold in the previous chapter. If we were to fold using multiplication, 1.0 is used instead as the initial value. In the end, we just pass the sum to the square root function, sqrt, and we are done:

```fsharp
let stddev2(values) =  
let avg = Seq.average values  
values  
|> Seq.fold (fun acc x -> acc + (1.0 / float (Seq.length  
    values)) * (x -avg) ** 2.0) 0.0  
|> sqrt
```

Let's verify using some sample data:

```fsharp
> stddev1 [2.0; 4.0; 4.0; 4.0; 5.0; 5.0; 7.0; 9.0];;
val it : float = 2.0
> stddev2 [2.0; 4.0; 4.0; 4.0; 5.0; 5.0; 7.0; 9.0];;
val it : float = 2.0
```
Now, we can go back and analyze the random data that was generated and used earlier when we looked at the build in the sequence functions:

```ml
let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 100 -> rnd() * 10.0]

let var = variance data
let std = stddev2 data
```

We can check the fact that the square of the standard deviation is equal to the variance:

```ml
> std * std = var;;
val it : bool = true
```

**Looking at an example application**

The example application that we'll look at in this section is a combination of the parts that we looked at in this chapter and will simply produce an output with statistics about the given sequence of data:

```ml
/// Helpers to generate random numbers
let random = new System.Random()
let rnd() = random.NextDouble()
let data = [for i in 1 .. 500 -> rnd() * 10.0]

/// Calculates the variance of a sequence
let variance(values:seq<float>) = values
    |> Seq.map (fun x -> (1.0 / float (Seq.length values)) * (x - (Seq.average values)) ** 2.0)
    |> Seq.sum

/// Calculates the standard deviation of a sequence
let stddev(values:seq<float>) = values
    |> Seq.fold (fun acc x -> acc + (1.0 / float (Seq.length values)) * (x - (Seq.average values)) ** 2.0) 0.0
    |> sqrt

let avg = data |> Seq.average
let sum = data |> Seq.sum
let min = data |> Seq.min
let max = data |> Seq.max
let var = data |> variance
let std = data |> stddev
```
Evaluating the code results in an output of the statistical properties of the random sequence generated is shown as follows:

```fsharp
val avg : float = 5.150620541
val sum : float = 2575.310271
val min : float = 0.007285140458
val max : float = 9.988292227
val var : float = 8.6539651
val std : float = 2.941762244
```

This means that the sequence has an approximate mean value of 5.15 and an approximate standard deviation of 2.94. Using these facts, we can rebuild the distribution assuming that the numbers are distributed according to any of the known distributions, such as normal distribution.

### Using the Math.NET library

Instead of implementing your own functions for numerical support, there is an excellent library called Math.NET. It's an open-source library covering fundamental mathematics such as linear algebra and statistics.

The Math.NET library consists of the following libraries:

- **Math.NET numerics**: Numerical computing
- **Math.NET neodym**: Signal processing
- **Math.NET linq algebra**: Computer algebra
- **Math.NET yttrium**: Experimental network computer algebra

In this section, we'll look at Math.NET numerics, and see how it can help us in our F# programming. First, we need to make sure that Math.NET is installed on our system.

### Installing the Math.NET library

The Math.NET library can be installed using the built-in Package Manager.
1. Open the **Package Manager Console** by going to **View** | **Other Windows** | **Package Manager Console**.

2. Type in the following command:
   
   ```
   Install-Package MathNet.Numerics
   ```

3. Wait for the installation to complete.

   
   You can read more about the Math.NET project on the project's website: http://www.mathdotnet.com/.

---

**Introduction to random number generation**

Let's start by looking at the various ways of generating random numbers. Random numbers are frequently used in statistics and simulations. They are used extensively in the Monte Carlo simulations. Before we start looking at Math.NET and how to generate random numbers, we need a little bit of the background theory.

**Pseudo-random numbers**

In computers and programming, random numbers often refer to pseudo-random numbers. Pseudo-random numbers appear to be random, but they are not. In other words, they are deterministic if some properties of the algorithm and the seed that is used are known. The seed is the input to the algorithm to generate the number. Often, one chooses the seed to be the current time of the system or another unique number.
The random number generator in the System.Random class that is provided with the .NET platform is based on a subtractive random number generator algorithm by Donald E. Knuth. This algorithm will generate the same series of numbers if the same seed is used.

**Mersenne Twister**

The Mersenne Twister pseudo random number generator is capable of producing less deterministic numbers in an efficient way. These properties make this algorithm one of the most popular ones used today. The following is an example of how to use this algorithm in Math.NET:

```fsharp
open MathNet.Numerics.Random

let mersenneTwister = new MersenneTwister(42);
let a = mersenneTwister.NextDouble();
```

In F# Interactive, we can generate some numbers using Mersenne Twister:

```fsharp
> mersenneTwister.NextDouble();;
val it : float = 0.7965429842

> mersenneTwister.NextDouble();;
val it : float = 0.9507143116

> mersenneTwister.NextDouble();;
val it : float = 0.1834347877

> mersenneTwister.NextDouble();;
val it : float = 0.7319939383

> mersenneTwister.NextDouble();;
val it : float = 0.7796909974
```

**Probability distributions**

Probability distributions are commonly used in statistics and finance. They are used to analyze and categorize a set of samples to investigate their statistical properties.

**Normal distribution**

Normal distribution is one of the most commonly used probability distributions.
In Math.NET, normal distribution can be used in the following way:

```csharp
open MathNet.Numerics.Distributions

let normal = new Normal(0.0, 1.0)
let mean = normal.Mean
let variance = normal.Variance
let stddev = normal.StdDev
```

By using the preceding example, we create a normal distribution with zero mean and a standard deviation of one. We can also retrieve the mean, variance, and standard deviation from a distribution:

```csharp
> normal.Mean;;
val it : float = 0.0

> normal.Variance;;
val it : float = 1.0

> normal.StdDev;;
val it : float = 1.0
```

In this case, the mean and standard deviation is the same as we specified in the constructor of the `Normal` class. It's also possible to generate random numbers from a distribution. We can use the preceding distribution to generate some random numbers, from the properties that are defined:

```csharp
> normal.Sample();;
val it : float = 0.4458429471

> normal.Sample();;
val it : float = 0.4411828389

> normal.Sample();;
val it : float = 0.9845689791

> normal.Sample();;
val it : float = -1.733795869
```

In the Math.NET library, there are also other distributions such as:

- Poisson
- Log normal
- Erlang
- Binomial

For More Information:

Statistics

In Math.NET, there is also great support for descriptive statistics that can be used to determine the properties of a collection of samples. The samples can be numbers from a measurement or generated by the same library.

In this section, we'll look at an example where we'll analyze a collection of samples with known properties, and see how the `DescriptiveStatistics` class can help us out.

We start by generating some data to be analyzed:

```fsharp
let dist = new Normal(0.0, 1.0)
let samples = dist.Samples() |> Seq.take 1000 |> Seq.toList
```

Notice the conversion from `Seq` to `List`; this is done because otherwise, `samples` will be a lazy collection. This means that the collection will be a set of different numbers every time it's used in the program, which is not what we want in this case. Next we instantiate the `DescriptiveStatistics` class:

```fsharp
let statistics = new DescriptiveStatistics(samples)
```

It will take the samples that were previously created and create an object that describes the statistical properties of the numbers in the `samples` list. Now, we can get some valuable information about the data:

```fsharp
// Order Statistics
let maximum = statistics.Maximum
let minimum = statistics.Minimum

// Central Tendency
let mean = statistics.Mean

// Dispersion
let variance = statistics.Variance
let stdDev = statistics.StandardDeviation
```

If we look closer at the mean, variance, and standard deviation respectively, we see that they correspond well with the expected values for the collection:

```fsharp
> statistics.Mean;;
val it : float = -0.002646746232

> statistics.Variance;;
val it : float = 1.000011159

> statistics.StandardDeviation;;
val it : float = 1.00000558
```
Linear regression

Linear regression is heavily used in statistics where sample data is analyzed. Linear regression tells the relationship between two variables. It is currently not part of Math.NET but can be implemented using it. Regression in Math.NET is an asked-for feature; and hopefully, it will be supported natively by the library in the future.

Using the least squares method

Let's look at one of the most commonly used methods in linear regression, the least squares method. It's a standard approach to find the approximate solution using the least squares method. The least squares method will optimize the overall solution with respect to the squares of the error, which means that it will find the solution that best fits the data.

The following is an implementation of the least squares method in F# using Math.NET for the linear algebra part:

```fsharp
open System
open MathNet.Numerics
open MathNet.Numerics.LinearAlgebra
open MathNet.Numerics.Distributions

/// Linear regression using least squares

let X = DenseMatrix.ofColumnsList 5 2 [ List.init 5 (fun i ->
    1.0); [ 10.0; 20.0; 30.0; 40.0; 50.0 ] ] X
let y = DenseVector [| 8.0; 21.0; 32.0; 40.0; 49.0 |]
let p = X.QR().Solve(y)
printfn "X: %A" X
printfn "y: %s" (y.ToString())
printfn "p: %s" (p.ToString())

let (a, b) = (p.[0], p.[1])
```

The independent data $y$ and the dependent data $x$ are used as inputs to the solver. You can use any linear relationship here, between $x$ and $y$. The regression coefficients will tell us the properties of the regression line, $y = ax + b$.
Using polynomial regression

In this section, we're going to look at a method for fitting a polynomial to data points. This method is useful when the relationship in the data is better described by a polynomial, such as a second or third degree one. We'll use this method in Chapter 6, Exploring Volatility, where we'll fit a second degree polynomial to a option data to construct a graph over the volatility smile. We'll lay out the foundations needed for that use case in this section.

We'll continue to use Math.NET for our linear algebra calculations, to solve for the coefficients for a polynomial of a second degree.

We'll start out by generating some sample data for a polynomial:

\[ y = x^2 - 3x + 5 \]

Then, we generate x-values from -10.0 to 10.0 with increments of 0.2, and y-values using these x-values and the preceding equation with added noise. To accomplish this, the normal distribution with zero mean and 1.0 in standard deviation is used:

```fsharp
let noise = (Normal.WithMeanVariance(0.0, 0.5))

/// Sample points for x^2 - 3x + 5
let xdata = [-10.0 .. 0.2 .. 10.0]
let ydata = [for x in xdata do yield x ** 2.0 - 3.0 * x + 5.0 + noise.Sample()]
```

Next, we use the linear algebra functions from Math.NET to implement the least square estimation for the coefficients. In mathematical terms, this can be expressed as:

\[
\hat{c} = (A^T A)^{-1} A^T \tilde{y}
\]

This means we will use the matrix A, which stores the x-values and the y-vector to estimate the coefficient vector c. Let's look at the following code to see how this is implemented:

```fsharp
let N = xdata.Length
let order = 2

/// Generating a Vandermonde row given input v
let vandermondeRow v = [for x in [0..order] do yield v ** (float x)]
```

For More Information:

/// Creating Vandermonde rows for each element in the list
let vandermonde = xdata |> Seq.map vandermondeRow |> Seq.toList

/// Create the A Matrix
let A = vandermonde |> DenseMatrix.ofRowsList N (order + 1)
A.Transpose()

/// Create the Y Matrix
let createYVector order l = [for x in [0..order] do yield l]
let Y = (createYVector order ydata |> DenseMatrix.ofRowsList (order + 1) N).Transpose()

/// Calculate coefficients using least squares
let coeffs = (A.Transpose() * A).LU().Solve(A.Transpose() * Y).Column(0)

let calculate x = (vandermondeRow(x) |> DenseVector.ofList) * coeffs

let fitxs = [(Seq.min xdata).. 0.02 ..(Seq.max xdata)]
let fitys = fitxs |> List.map calculate
let fits = [for x in [(Seq.min xdata).. 0.2 ..(Seq.max xdata)] do yield (x, calculate x)]

The values in the coefficient vector are in reverse order, which means they correspond to a polynomial that fits the data, but the coefficient is reversed:

```fsharp
> coeffs;;
val it : seq<real> = seq [4.947741224; -2.979584718; 1.001216438]
```

For More Information:  
The values are pretty close to the polynomial we used as the input in the preceding code. The following is a graph of the sample data points together with the fitted curve. The graph is made using FSharpChart, which we'll look into in the next chapter.

The curious reader can use the following snippet to produce the preceding graph:

```fsharp
open FSharp.Charting

fsi.AddPrinter(fun (ch:FSharp.Charting.ChartTypes.GenericChart) ->
    ch.ShowChart(); "FSharpCharting")

let chart = Chart.Combine [Chart.Point(List.zip xdata ydata);
    Chart.Line(fits).WithTitle("Polynomial regression")]
```

Learning about root-finding algorithms

In this section, we'll learn about the different methods used in numerical analysis to find the roots of functions. Root-finding algorithms are very useful, and we will learn more about their applications when we talk about volatility and implied volatility.
The bisection method

In this section, we will look at a method for finding the roots of a function using the bisection method. This method will be used later in this book to numerically find the implied volatility for an option that is given a certain market price. The bisection method uses iteration and repeatedly bisects an interval for the next iteration.

The following function implements bisection in F#:

```fsharp
let rec bisect n N (f:float -> float) (a:float) (b:float) (t:float) : float =
    if n >= N then -1.0
    else
        let c = (a + b) / 2.0
        if f(c) = 0.0 || (b - a) / 2.0 < t then
            // Solution found
            c
        else
            if sign(f(c)) = sign(f(a)) then
                bisect (n + 1) N f c b t
            else
                bisect (n + 1) N f a c t
```

Looking at an example

Now, we will look at an example of solving the roots of a quadratic equation. The equation, $x^2 - x - 6$, is plotted in the following figure:

![Plot of the quadratic equation $x^2 - x - 6$](image)

For More Information:

The roots of the preceding quadratic equation can easily be seen in the figure. Otherwise, there are analytical methods of solving it; for example, the method of completing the square. The roots of the equation are -2 and 3.

Next, we create an anonymous function in F# to describe the one that we are interested in solving the roots for:

```fsharp
let f = (fun x -> (x**2.0 - x - 6.0))
```

We can test the preceding function using the roots that we found in the preceding figure:

```fsharp
> f(-2.0);;
val it : float = 0.0
> f(3.0);;
val it : float = 0.0
```

The results are as expected. Now, we can continue and use the lambda function as an argument to the `bisetct` function:

```fsharp
// First root, on the positive side
let first = bisect 0 25 f 0.0 10.0 0.01

// Second root, on the negative side
let second = bisect 0 25 f -10.0 0.0 0.01
```

The first two arguments, 0 and 25, are used to keep track of the iterations. We pass in 0 because we want to start from iteration 0 and then iterate 25 times. The next argument is the function itself that we defined in the preceding code as `f`. The next two arguments are limits, that is, the range within which we can look for the root. And the last one is just a value for the accuracy used for comparison inside the iteration.

We can now inspect the two variables and see if we find the roots:

```fsharp
> first;;
val it : float = -2.001953125
> second;;
val it : float = 2.998046875
```

They are almost equal to the analytical solution of -2 and 3 respectively. This is something that is typical for numerical analysis. The solutions will almost never be exact. In every step, some inaccuracy is added due to the floating-point numbers, rounding, and so on.
Finding roots using the Newton–Raphson method

The Newton-Raphson method, or simply Newton's method, usually converges faster than the bisection method. The Newton-Raphson method also needs the derivative of the function, which can be a problem in some cases. This is especially true when there is no analytical solution available. The following implementation is a modification of the bisection method using the derivative of the function to determine if a solution has been found. Let's look at the implementation of Newton's method in F#:

```fsharp
// Newton's Method
let rec newtonraphson n N (f:float -> float) (fprime:float -> float) (x0: float) (tol:float) : float =
    if n >= N then -1.0
    else
        let d = fprime(x0)
        let newtonX = x0 - f(x0) / d
        if abs(d) < tol then
            -1.0
        else
            if abs(newtonX - x0) < tol then
                newtonX // Solution found
            else
                newtonraphson (n +1) N f fprime newtonX tol
```

One of the drawbacks of using the preceding method is that we use a fixed point convergence criteria, \( \text{abs}(\text{newtonX} - x0) < \text{tol} \), which means that we can be far from the actual solution when this criteria is met.

Looking at an example

We can now try to find the square root of two, which is expected to be 1.41421. First, we need the function itself, \( \text{fun } x \rightarrow (x**2.0 - 2.0) \). We also need the derivative of the same function, \( x \rightarrow (2.0*x) \):

```fsharp
let f = (fun x -> (x**2.0 - 2.0))
let fprime = (fun x -> (2.0*x))
let sqrtOfTwo = newtonraphson 0 25 f fprime 1.0 10e-10
```

Now, we use the Newton-Raphson method to find the root of the function, \( x^2 - 2 \). Using F# Interactive, we can investigate this as follows:

```fsharp
> newtonraphson 0 25 f fprime 1.0 10e-10;;
val it : float = 1.414213562
```
This is the answer we would expect, and the method works for finding roots! Notice that if we change the starting value, \(x_0\), from 1.0 to -1.0, we'll get the negative root:

```fsharp
> newtonraphson 0.25 f fprime -1.0 10e-10;;
val it : float = -1.414213562
```

This is also a valid solution to the equation, so be aware of this when you use this method for solving the roots. It can be helpful to plot the function, as we did in the section about the bisection method, to get a grip on where to start from.

**Finding roots using the secant method**

The secant method, which doesn't need the derivative of the function, is an approximation of the Newton-Raphson method. It uses the finite difference approximation in iterations. The following is a recursive implementation in F#:

```fsharp
// Secant method
let rec secant n N (f:float -> float) (x0:float) (x1:float) (x2:float) : float =
    if n >= N then x0
    else
        let x = x1 - (f(x1))*((x1 - x0)/(f(x1) - f(x0)))
        secant (n + 1) N f x x0
```

**Looking at an example**

Let's look at an example where we use the secant method to find one root of a function. We'll try to find the positive root of 612, which is a number just under the square of 25 (25 x 25 = 625):

```fsharp
let f = (fun x -> (x**2.0 - 612.0))

> secant 0 10 f 0.0 10.0 30.0;;
val it : float = 24.73863375
```
Summary

In this chapter, we looked deeper into F# and numerical analysis and how well the two fit together because of the functional syntax of the language. We covered algorithm implementation, basic numerical concerns, and the Math.NET library. After reading this chapter, you will be more familiar with both F# and numerical analysis and be able to implement algorithms yourself. At the end of the chapter, an example using the bisection method was covered. This method will prove to be very useful later on when we talk about Black-Scholes and implied volatility.

In the next chapter, we will build on what we learned so far and extend our current knowledge with data visualization, basic GUI programming, and plotting financial data.

For More Information:
Where to buy this book

You can buy F# for Quantitative Finance from the Packt Publishing website: http://www.packtpub.com/fsharp-for-quantitative-finance/book. Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.