Numerical Methods, Algorithms and Tools in C#

Waldemar Dos Passos

CRC Press
Taylor & Francis Group

Numerical Methods, Algorithms and Tools in C#

Numerical Methods, Algorithms and Tools in C#

Waldemar Dos Passos

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

All the source codes for the material contained in this book can be downloaded directly from the publisher's website: http://www.crcpress.com/product/isbn/9780849374791 followed by selecting the option for "Downloads & Updates."

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number: 978-0-8493-7479-1 (Hardback)

Library of Congress Cataloging-in-Publication Data

Dos Passos, Waldemar.
Numerical methods, algorithms, and tools in C# / Waldemar Dos Passos.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-8493-7479-1 (hardcover : alk. paper)
1. Numerical analysis--Data processing. 2. Algorithms. 3. C# (Computer program language) I. Title.

QA297.D684 2010
518.0285'5133--dc22 2009031461

Preface

Today, more than at any other time in the history of mankind, computers are increasingly and successfully being exploited to gain a better understanding of our physical world and as a result, also deepen our appreciation and reverence for God's Creation. Consequently, as computers evolve, so must the means to control them through advancements not just in hardware but also in software.

In order to satisfy this demand for better software, Microsoft released an entirely new programming language called C# that incorporates the best features of all the other existing popular programming languages such as Java, C/C++, and Visual Basic. In spite of considerable resistance by some people who persist on clinging on to the past and continue to program computers the hard way, C# has now firmly established itself worldwide as arguably the preferred language for software application development. Although many excellent books on the topic of general programming in C# have been written, there is still a considerable lack of published material on the topic of numerical methods in C#.

Accordingly, *Numerical Methods, Algorithms and Tools in C#* is a book containing a large collection of very useful ready-to-use mathematical routines, algorithms and other computational tools aimed at programmers, mathematicians, statisticians, scientists, engineers and anyone else interested in developing mathematically oriented computer applications in the relatively new and easy-to-learn object-oriented C# programming language from Microsoft. With a heavy emphasis on using well established numerical methods, object-oriented techniques and the latest state-of-the-art Microsoft .NET programming environment, this book provides readers with working C# code including practical examples that can be easily customized and implemented to solve complex engineering and scientific problems typically found in real-world applications.

For the benefit of those readers who are not yet familiar with C#, Chapter 1 provides a brief outline of the .NET Framework, the C# programming language and the basic concepts of Object Oriented Programming (OOP). Special attention is given to topics that illustrate how to best utilize these and other tools to develop accurate and robust numerical methods in C#.

Chapter 2 is entirely focused on the .NET Framework Math Class Library which already comes built into Microsoft's Visual Studio software development system. Additional material is introduced where appropriate in order to supplement, complete or otherwise enhance the features already available with this library.

Chapter 3 introduces data structures along with their associated functions that are particularly useful for programming and working with vectors and matrices. These routines are often used in more advanced applications in later chapters.

Chapter 4 is entirely dedicated to the topic of complex numbers. Since timing issues can sometimes pose a substantial problem when doing numerical calculations, complex number functions are presented using both elegant state-of-the-art object-oriented methods which, although slick, can at times carry some overhead and the old fashioned but proven methods which at times have been found to actually run faster on some computers. In addition, important overflow and underflow issues are also discussed and alternative solutions to avoid those problems are proposed.

Chapter 5 is devoted solely to sorting and searching algorithms. Computers are often required to perform various types of data sorting for which many different algorithms exist. Consequently, choosing the most efficient sorting algorithm is a very important decision that developers frequently have to make. In this chapter, readers are provided with both a wide selection of sorting and searching algorithms from which to choose along with a brief explanation of how each algorithm works.

Chapter 6 is centered on the topic of bit manipulation which is typically used in a variety of programming applications ranging anywhere from computer interfacing to image processing.

Chapter 7 is focused on interpolation methods. Equations that cannot be solved analytically often need to be solved using some kind of interpolation scheme, and this chapter has plenty of practical examples to illustrate how one might handle this kind of problem.

Chapter 8 centers on the numerical manipulation of linear algebraic equations. This is actually a huge topic by itself and quite worthy of its own book. Nevertheless, a substantial amount of useful information can be readily obtained from just a handful of these powerful tools.

Chapter 9 is focused on numerical methods for calculating approximate solutions to nonlinear equations which often appear naturally in various branches of science and engineering.

Chapter 10 is devoted exclusively to the topic of random numbers. Although C# comes with its own internal random number generator function, it is not regarded to be sufficiently robust for use in advanced secured applications or in computer simulations that require thousands and sometimes even millions of random numbers in order to produce reliable and accurate results. Alternate ways to obtain both computer generated pseudo-random numbers and *real* random numbers obtained from naturally occurring physical phenomena are also discussed. In addition, routines are also provided for generating random numbers that follow a particular probability distribution function.

Chapter 11 describes various methods for approximating numerical differentiation of functions. This is a very tricky and controversial topic whose approximations can give fairly good to atrociously bad results. Nevertheless, numerical methods do exist for calculating these types of functions. The trick is really in learning to recognize the difference between good and bad results and in choosing the best available method for use in a particular situation.

Chapter 12 centers on developing numerical methods for approximating integrals of specific functions as well as from collections of raw data points. Other more exotic ways of calculating integrals, such as by using Monte Carlo methods, are also briefly discussed.

Chapter 13 contains a considerable number of routines for use in performing statistical analysis of data.

Chapter 14 is devoted to developing numerical methods for approximating special functions which are typically found in various branches of mathematics, physics and engineering.

Chapter 15 is focused on least squares and numerical curve fitting methods that are frequently used in analyzing experimental data. A brief discussion of the χ^2 goodness-of-fit test is also included.

Chapter 16 centers on developing routines to find numerical solutions to ordinary differential equations. Although this is really a huge topic, there are some basic numerical methods which can be used successfully to solve a lot of these types of equations in many real-world applications.

Chapter 17 introduces some numerical methods for solving partial differential equations. Although this is also a huge topic by itself and quite deserving of its own book, there are some standard types of partial differential equations that arise naturally in many areas of science and engineering, and whose solutions can be approximated by well established numerical methods.

Chapter 18 focuses on optimization methods which are primarily aimed at the minimization or maximization of functions and thus have many practical scientific and engineering applications. Since this is still a very active area of ongoing research, the examples presented here are more narrowly focused on just a few established topics with the explicit purpose of illustrating how such methods may be individually customized and then applied towards solving more advanced problems.

Lastly, I would like to point out that most of the numerical methods described in this book have actually been around in one form or another for years, and sometimes even for centuries, and it is only their computer implementation in C# that makes this book uniquely different from some other book on the topic of numerical analysis. Accordingly, I have made every effort to track down and give proper credit to original sources whenever possible as the size of this book's reference section can easily attest. In addition, I have also made every effort to provide my readers with accurate, reliable information to help them in their efforts to successfully complete their programming projects. Unfortunately, unwanted mistakes including typographical errors may inadvertently creep up somewhere in this book. As a result, I would greatly appreciate if my readers would be so kind as to bring to my attention if such errors are ever found so that I may promptly have the problem corrected for any future editions of this book. Also, as with just about everything we do in life, there is always room for improvement. Accordingly, I would also very much welcome any constructive criticism that my readers may have regarding this book so that I can perhaps make appropriate changes. Finally, there is an old saying that states, "an author never finishes a book, but merely abandons it." I have certainly come to appreciate that observation after working on this project for so long and making countless revisions. Nevertheless, this has certainly been a very enjoyable project where just about every word was carefully chosen and every topic was meticulously researched and documented. Therefore, if it is indeed true that I have willingly chosen to abandon writing this book, it is only with the modest hope that it may be useful to my readers in spite of any possible shortcomings.

Waldemar Dos Passos, Ph.D.
Concord, California
e-mail: waldemar007@hotmail.com
website: www.waldemardospassos.com

Acknowledgements

It gives me great pleasure to thank the many people who made this book possible. First, I would very much like to thank my publisher, Nora Konopka, for not only accepting this book for publication but also for her exceptional patience as I underwent a series of unforeseen tumultuous events in my life during the course of writing this book which unfortunately led to some regrettable delays in its original publication target date. I would also like to particularly thank both my project director, Theresa Delforn, and my editor, Amy Rodriguez, for their excellent expert guidance in various aspects of this project. I would also like to thank Dawn Snider for her excellent artistic skills in designing the cover for this book. Many thanks to Ashley Gasque for guiding me through the necessary bureaucratic paperwork and to Shashi Kumar for some expert LaTeX tips he gave me. I would also like to thank all those other wonderful people at Taylor & Francis who have worked tirelessly behind the scenes to make this project a success but whose exact names I may likely never come to know.

I am also very grateful for the support I received from the H.E. Martin Foundation under grant 13011938. Without their most kind and extraordinary generous financial assistance, the writing of this book would not have been possible.

I am especially grateful to my third grade teacher, Miss Daly, for all her help, patience, kindness, and enthusiasm which ultimately sparked my interest in mathematics and eventually, physics. Looking back over all these years that have elapsed since I was a student in her class, I can now say unequivocally that Miss Daly was by far the very best and most caring teacher, professor, or instructor I ever had.

Lastly, I would also like to express my deepest and most heartfelt thanks to my parents, Helenice and Waldemar Dos Passos (Sr.)

This book is dedicated with all my love and care to my parents,
Helenice and Waldemar Dos Passos (Sr.)
for all their hard work, genuine love, and selfless sacrifices
made on my behalf throughout my entire life.

"In this life we cannot do great things; only small things with great love."
Mother Teresa

Ad Majorem Dei Gloriam

Contents

1 Introduction

The main objective of this first chapter is to provide my readers with a brief outline of the .NET Framework, the C# programming language and the basic concepts of Object Oriented Programming (OOP). Special attention will be given to materials that illustrate how to best utilize these tools to develop accurate and robust numerical methods in C# primarily for use in scientific and engineering applications.

1.1 C# and the .NET Framework

In the late 1990s, Microsoft embarked on a project to update and improve its flagship software application development system, more commonly known as Visual Studio, and as a result of this effort, an entirely new programming language named C# emerged that, among other things, essentially incorporates the best features of all the other popular programming languages of the time such as Java, C/C++ and Visual Basic. Consequently, since its first release in July of 2000, C# has quickly established itself worldwide as perhaps the preferred language for software application development. Besides being a very powerful general purpose, object-oriented programming language, C# enjoys the full advantage and benefits of being fully integrated with the Microsoft .NET Framework system.

The Microsoft .NET Framework is a fundamental Windows operating system component that supports building and running both software applications as well as Web services. It consists of a large set of class libraries of pre-coded solutions to common programming problems and also provides a new environment for building applications that can be deployed and executed across multiple architectures and operating systems. The .NET Framework was designed to be installed on top of the Windows operating system and is divided into two key components: a runtime environment called the Common Language Runtime (CLR), which provides the runtime services to manage and execute applications originally written in any one of the .NET programming languages, and a large library of pre-coded object oriented classes called the Framework Class Library (FCL) which provides the required services for developing .NET applications. Conceptually, .NET applications reside above the .NET Framework architecture and can be illustrated abstractly as shown in Table 1.1.

The C# language specification [1] has been standardized by ECMA International,

TABLE 1.1
Outline of the .NET Framework Architecture

	.NET Applications	
Visual Basic	Visual C#	Visual C++
	.NET Framework Class Library (FCL)	
	Common Language Runtime (CLR)	
	Operating System	

which is an industry association dedicated to the standardization of information and communication systems. As a result, consumers can now choose to buy their C# compilers from among several different manufacturers such as Microsoft [2], SharpDevelop [3], DotGNU [4] and Mono [5]. However, the most popular C# compiler on the market today comes bundled with Microsoft's Visual Studio software development system which, in addition to having a C# compiler, also provides a full featured integrated development environment (IDE) that standardizes support for many of the other popular programming languages like Visual Basic and Visual C++. Accordingly, all the code and examples contained in this book were written and tested using the latest version of Microsoft Visual Studio.

As part of the natural evolution of programming languages, C# has incorporated and exploited familiar features from C/C++ and Java that already had a proven record of success. In addition, C# also has a number of unique new features that make it a very attractive programming language. For example, C# controls access to hardware and memory resources and, by default, does not allow for the explicit usage and manipulation of pointers as C and C++ do except for sections of code that have been specifically designated as *unsafe*. This feature, along with the support of a more powerful garbage collector that automatically manages all aspects of memory allocation and de-allocation during runtime, has now made frustrating memory leaks and dangling pointers, often hard to find and debug in C/C++ programs, a thing of the past. In addition, improvements made in exception handling provide a well structured and extensible approach to error detection and recovery. C# is also designed with type-safe features that make it impossible to have non-initialized variables, to index arrays beyond their bounds, or to perform unchecked type casts. The C# language also supports more advanced features such as multi-threading and Just-in-Time (JIT) compilation from byte code to native code to name a few.

1.2 Installing C# and the .NET Framework

You can buy Visual C# either by itself or as part of the Visual Studio IDE, which also includes, in addition to Visual C#, support for other programming languages such as Visual Basic and Visual C++. Visual C# comes in several editions. If you want Visual C# all by itself, you have only one choice: the Express edition. However, if you buy Visual C# as part of the Visual Studio package, you have three choices: Standard, Professional and Team editions to accommodate every budget, work environment and skill level. There is also the Academic version of the Professional edition which is available at a substantial discount for students and teachers. The key differences between these various editions center primarily on the number of development environment features available to the programmer. If you do computer programming as a hobby, the Express edition should work just fine for most of the applications described in this book. However, if you do a significant amount of software development in various languages and platforms, then you will likely derive most benefit from the multipurpose Professional edition.

The Visual C# installation kit may consist of one or more CDs depending on the edition chosen. The installation itself is relatively easy and is simply a matter of following the directions displayed on the screen. If you do not have the required .NET Framework already installed on your system, the installation program will perform that task automatically for you prior to doing anything else. Due to the huge size of the program, it may take some time to install. But patience is a virtue in programming and it begins with the installation of Microsoft Visual Studio. Installing the latest version of the MSDN reference libraries directly on your computer is also highly recommended so that help files can be retrieved and promptly consulted as needed.

1.3 Overview of Object-Oriented Programming (OOP)

There are primarily two methods of programming in use today: procedural and object-oriented. Procedural programming has its roots in the earliest forms of programming languages and essentially involves creating and naming computer memory locations that can hold data which can then be changed and manipulated through a series of sequential steps. The named computer memory locations are called variables because they hold values that might vary at some point during the life of the program. Sometimes a finite number of these sequential steps used in computer programs can be grouped into smaller logical units called procedures. Hence the objective of procedural programming is to focus on the creation of procedures to operate and potentially alter data. If, at some later point in time, the program's original specifications somehow change significantly enough to warrant a corresponding fundamental change in the program's original data structure then the original code must also be changed and rewritten to accept the new data format. Unfortunately, such changes often result in additional work for programmers and this may ultimately lead to potential project release delays, higher production costs and perhaps most importantly, can also increase the chances for unwanted bugs to appear in the code.

Object-oriented programming can be thought of as a major significant improvement of procedural programming. Whereas procedural programming is focused on creating procedures to manipulate data, object-oriented programming centers on creating abstract, self-contained software entities called objects that contain both attributes and methods, previously also known as data and procedures. The attributes of an object provide information about its characteristics whereas the methods of an object provide information about how to manipulate those attributes. More formally, an object is said to be an instance of a class and a class is said to be a template or a blueprint for describing how an object is created and how it behaves at runtime. Hence, a class defines behavior, properties and other attributes of an object that is an instance, or example, of that class.

Object-oriented programs have attributes and methods that are said to be encapsulated into objects. *Encapsulation* is the object oriented principle associated with hiding the internal structural details of an object from the outside world so that programmers can only use a well defined interface to interact with an object's internal structure. This feature is intended to prevent programmers from easily and perhaps even recklessly altering an object's internal structure or behavior. *Polymorphism* is the object-oriented programming principle that allows the creation of methods that act appropriately depending on the context within which a particular task is carried out. *Inheritance* is an object oriented principle relating to how one class, called the *derived* or *child* class, can share the characteristics and behavior from another class, called the *base* or *parent* class. In addition to its own new and unique attributes and methods, the derived class is said to *inherit* and thus contain nearly all the attributes and methods of the existing base class.

Therefore, besides retaining all the familiar and well established concepts of data (*i.e.* attributes) and procedures (*i.e.* methods), object-oriented programming also contains six additional unique features that are called: *objects, classes, encapsulation, interfaces, polymorphism, and inheritance.*

1.4 Your First C# Program

There are at least four general types of applications that can be created with C#: Console applications, Windows Form applications, Web Services and ASP.NET applications. Console applications run from the command line and are the most fundamental and easiest applications to understand. Consequently, console applications will be the preferred type of application used to illustrate the numerical examples given throughout this book. Window Form applications are applications that use the graphical user interface (GUI) provided by Microsoft Windows. Web Services are routines that can be called across the Web. ASP.NET applications are executed on a Web Server to generate dynamic Web pages.

It may come as a complete surprise for most people that they can actually start programming in C# for free. All that is needed to get started are two things: (1) a text editor like Notepad that already comes installed on your computer with the Windows operating system and (2) the .NET Framework which also comes with a simple command line C# compiler that you can easily download for free from the Microsoft website [6]. However, as your programs begin to grow in size, you will very likely want to eventually migrate towards a full featured integrated development environment that is much easier to use and is also rich with exciting features and tools. For now, however, let's start by examining the simplest possible C# program that can be written and then learn how to compile it and make it run. It is a long standing tradition in computer programming to introduce a new language with an example that displays the phrase, *Hello World!* on the console screen. The C# program to accomplish this task looks like this:

```
class MyFirstProgram
{
   static void Main()
   {
      System.Console.WriteLine("Hello, World!");
   }
}
```

The code consists of a class called MyFirstProgram and a method called Main. Each C# program has one and only one Main method, also called the entry point, which is called when the program is first started. `WriteLine(...)` is a method of the Console class that is found in the System namespace. Namespaces are used to group type declarations and classes that belong together into a cohesive unit. By using separate namespaces, collisions with identically named types and classes can be avoided. The System namespace, for example, which comes already built into the C# compiler, holds commonly used classes and is used to eliminate the need to write a lot of repeated code. When combined together these program instructions cause the computer to produce an output directed at the console screen. Using any text editor, type and save the above program to a file having a name of your choice but preferably ending with the extension `.cs`, such as `Example01.cs`. The command line Microsoft C# compiler, `csc.exe`, is located in the directory of the latest version of the .NET Framework installed in your computer. For the .NET Framework version 3.5 installed on a Windows XP operating system, for example, the Microsoft C# command line compiler, `csc.exe`, can be found in the following directory:

```
C:\WINDOWS\Microsoft.NET\Framework\3.5\
```

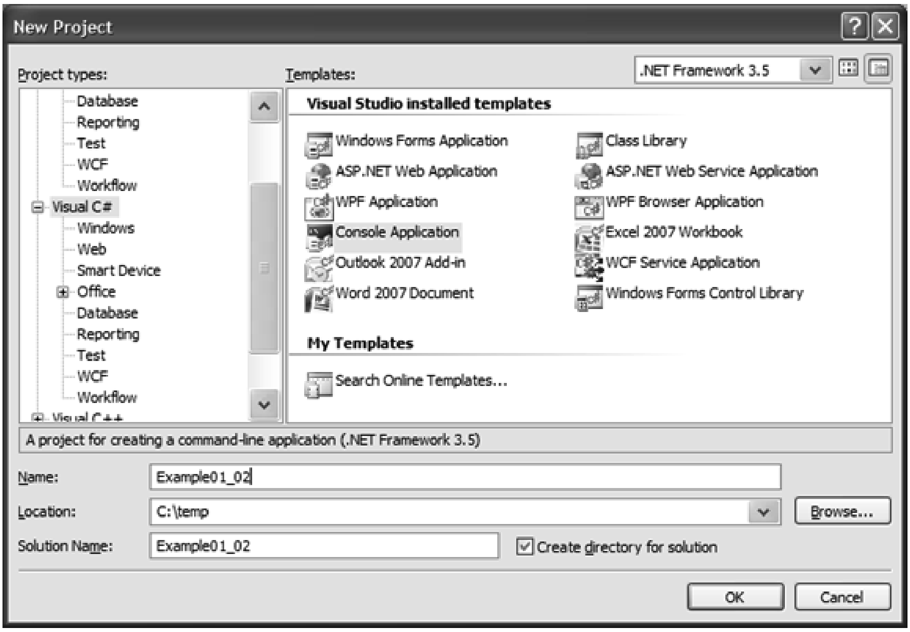
If you only want to use the C# command line compiler, then you should also add its location to the path variable in the Windows environment so that you will then be able to call it from any directory in your computer. To compile your program then enter: `csc Example01.cs` at the command prompt of your console window. If the compiling process went well and there are no scary looking error messages displayed on your screen, you can then run your program by entering its name at the command prompt after which you should see the resulting output: Hello World! displayed on your monitor screen. Alternatively, if you have installed Microsoft's Visual Studio IDE, you do not need to worry about setting up the path. Instead just open the Visual Studio command prompt by using the following steps: (1) click the Start menu button, (2) select All Programs, (3) select Microsoft Visual Studio, and finally (4) select the Visual Studio Command Prompt. This will not just open up a command line prompt window for you to use but will also automatically add the location of the command line compiler `csc.exe` to your operating system's path.

A useful feature of C# programs is that they can be spread across several files that can then be compiled together or separately. You can find more information about alternate and more elaborate ways to compile C# programs from the command line prompt by visiting Microsoft's MSDN website [7]. However, as you create larger C# programs, compiling them this way can quickly become very tedious. Fortunately, there are far easier ways to compile C# programs than using long, cumbersome and hard-to-remember compiler options in command line arguments.

Another way to write and compile your C# programs is to use the Visual Studio IDE. There are several advantages to using this approach. First, some of the code you need will be automatically created for you. Second, there are fantastic built-in debugging tools available for you to use which will, among many other things, automatically identify and place the cursor where errors are found in the code. Third, there is automatic statement completion which, together with the other extraordinary software resources already built into the IDE for you to use, will very likely save you countless hours of additional work.

Before using the Visual Studio IDE, you have to decide what type of project you wish to create. A *Windows Forms Application* project will allow you to create complete Windows applications, including dynamic linked libraries which can be linked to or referenced by other programs that do not necessarily even need to be written in C#. Unfortunately, as exciting as all these and other features may sound, developing Windows applications is beyond the scope of this book and is also not necessary for learning how to write useful numerical routines in C#. Instead, simpler project types using the *Console Application* option will be chosen to illustrate most of the material contained in this book. Once the basic numeric routines have been written and tested, one can then just add a reference to them or even copy and paste them inside more intricate Windows applications or even embed them into versatile dynamic linked libraries.

To compile and execute programs using the Visual Studio IDE you must first call up Visual Studio from your desktop. Once the Start Page is displayed on your screen, go to the menu bar at the top and click File followed by New Project. A new dialog window (see Fig.1-1) will pop up prompting the user to select the *Project Type* (Visual C#) and the *Templates* (Console Application) to use. The user is also asked to enter a project name and a location in the computer for it to be stored. Also, be sure to select the *.NET Framework 3.5* option (or higher, if available) on the combo box



A project for creating a command-line application (.NET Framework 3.5)

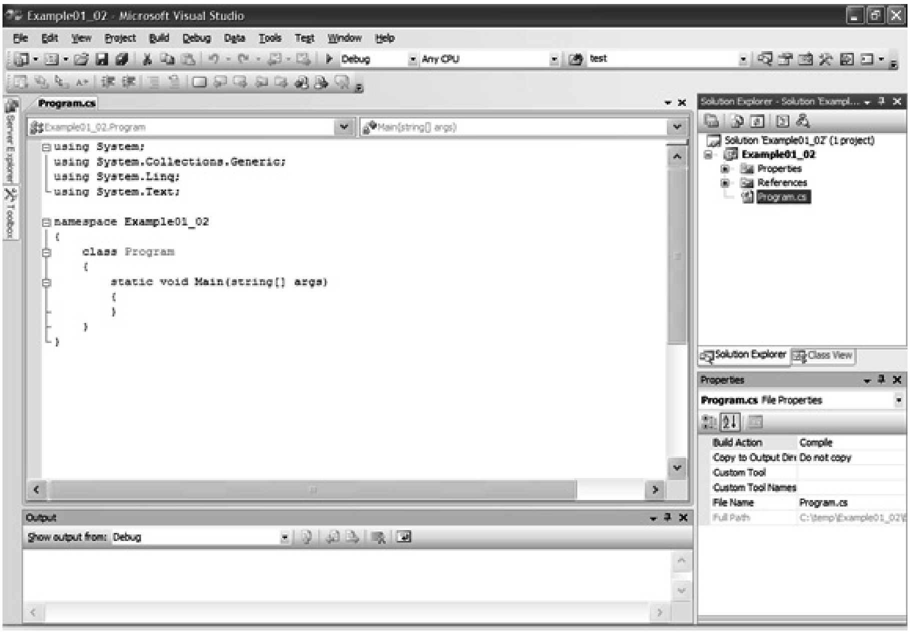
Name: Example01_02

Location: C:\temp

Solution Name: Example01_02

[✓] Create directory for solution

FIGURE 1.1
Setting up for a new project in Visual Studio IDE.


Example01_02 - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Test Window Help
Debug
Any CPU
test
Program.cs
Example01_02.Program
Main(string[] args)
Server Explorer
Toolbox
Solution Explorer - Solution 'Exampl...
Solution 'Example01_02' (1 project)
Example01_02
Properties
References
Program.cs
Solution Explorer
Class View
Properties
Program.cs File Properties
Output
Show output from: Debug

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Example01_02
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Build Action	Compile
Copy to Output Dir	Do not copy
Custom Tool	
Custom Tool Names	
File Name	Program.cs
Full Path	C:\temp\Example01_02\

FIGURE 1.2
Default Console Application project option in Visual Studio IDE.

located on the upper right corner of the screen as this will insure that your project files will be setup using the latest and greatest version of the .NET Framework.

After clicking `Ok` and waiting a few seconds, a new dialog window (see Fig.1-2) will pop up enabling the user to start entering code. Note that the IDE has automatically generated some code to get you started on your merry way to programming bliss. Nevertheless, you need to exercise some discretion since not every line of the automatic generated code is really necessary for the particular application you may have in mind. However, leaving any automatically generated code untouched should not cause problems when you try to compile your program.

In the region where you see the automatic generated code you can clear everything and then re-enter the original *Hello World* program described earlier or you can enter a slightly different version of that same program as shown below.

```
using System;
namespace Example02
{
    class MySecondProgram
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.ReadLine();
        }
    }
}
```

As the reader may have noticed, there were three additional *using* directives which were automatically generated when the *Console Application* template was selected. However, these were all manually deleted in the final version of Example02.cs because they are not needed to successfully compile this program. *using* directives are optional but when declared at the start of a code file, they are used to import specific namespaces for later use by the program and judicious use of such directives can save programmers from having to do a lot of additional typing.

You can also change the name Program of the original class to whatever name you prefer, such as MyFirstClass, but you should do it from the Solution Explorer because it is used to manage all the project files that make up a complete solution. Therefore, any changes you make using the Solution Explorer will be immediately and globally recognized by all the other code references contained within your project. Failure to do this step in the way just described will likely cause the C# compiler to generate unwanted error messages when you attempt to compile your program. To do this step simply take your mouse and right click on the highlighted item labeled Program.cs, select *rename* followed by entering the new name of your choice while preferably retaining the .cs file extension. Then finally confirm that you want the IDE to automatically rename all the references related to this item contained inside this project. An additional line of code containing the statement `Console.ReadLine();` was added in order to pause the output display on the screen until the user hits the *enter* key on the keyboard to continue. This prevents the output window from immediately closing after displaying its output: *Hello World!*.

To compile a program using the Visual Studio IDE, go to the menu toolbar at the top of the screen and click *Build* followed by *Build Solution*. Alternatively you can just press F6. If you are really lucky, and the compiling was a total success without displaying any error messages, then you should see status messages like *Build* followed by *Succeeded* appear in the Output Window. Finally, to do a test run of your program again go to the menu toolbar at the top of the screen and select *Debug* followed by *Start Without Debugging*. Alternatively you can also just press F5 from the outset. Either way, you should now see the output of your program appear on your monitor screen. While it's good practice to first build your project before attempting to run it so that any unwanted bugs in your code will be immediately detected and corrected, you can also press F5 which will automatically build your project and then immediately run it without any pause unless, of course, errors are found somewhere in your code.

Unlike when you use the Microsoft command line compiler `csc`, if you create and compile a C# program using the Visual Studio IDE, many additional files are created along with some folders. Of these, the innermost folder contains a bin folder, an obj folder and a properties folder along with some additional files. If you explore the directories down further, you should find that the bin folder contains a Debug folder which contains the executable file of the program you just created. Although at first the Visual Studio editor seems to create a lot of extraneous useless files, these extra files will become vitally important as you create more advanced C# projects.

1.5 Overview of the IDE Debugger

By default a program will enter into *break mode* whenever it encounters some kind of problem that is not properly handled, while executing its source code. This sequence of events is more commonly known as *throwing an exception*. When an application throws an exception that is not properly handled, the offending code statement is immediately highlighted upon entering the break mode and the *Exception Assistant* tries very hard, at various levels, to automatically determine the cause and location of the exception for you. However, the resulting error message is sometimes difficult to interpret so that the source code of the program can then be properly fixed. Fortunately, the Microsoft Visual Studio IDE comes equipped with a very powerful full feature *debugger* to help you locate and correct errors that prevent your application from running properly. With it you can step through program source code one line at a time to see how its execution is carried out, you can monitor form property values and you can even reset values at runtime. Features like this can help you understand the logic flow of a program and allow you to experiment with and even alter your code while the program is still running.

For a short descriptive tour of the basic features contained in the Visual Studio debugger, simply open your project file inside the IDE to bring up your source code on the console screen. Then in the code editor you can set one or more *breakpoints*, which pauses the program's execution at a specific code command, by clicking in the gray area on the left side of the code editor. A red dot will then appear on the left edge of the line that contains the breakpoint and the IDE will reverse highlight the code in red. Then run your application as usual and after the program pauses at the breakpoint, an arrow will mark the command that executes next. To step through the program's commands, click Step Into on the toolbar. To see the value of a particular item, place the mouse pointer on a reference to the item in the code editor. While the program's execution has paused at a breakpoint, you can also place the mouse pointer over a variable, property or expression to display its current value in a data tip. In addition, the *Edit and Continue* feature allows you to immediately fix an error and confirm that the correction you just made actually fixed the problem. In some cases this feature is very useful because it lets you fix one or more bugs in the code in just one test run. You can also reset the position of the program's execution arrow to run one or more commands again. To do this, right click the command on which you want to reset the execution arrow and then click the *Set Next* statement.

While in debug mode, you can also use the *Immediate Window* to display current program code values and even change them while the program is still running. To display the Immediate Window, click Debug on the menu bar, point to Windows, then click Immediate. To display a current program data value in the Immediate Window, type a question mark (?), followed by the item name and then press *Enter*. To change a program's data value, type an assignment statement that resets the value, and then press *Enter*.

Another useful debugging tool is the *Locals Window* which displays all current code variables and their associated values during program execution. As the execution proceeds, the variable values are all updated automatically. To open the Locals Window, click Debug on the menu bar, point to Windows, then click Locals. For more complex programs with many variables, the Locals Window can display a very long list of variables and values. Since you ordinarily track only a few variables at a time during a typical debugging session, it can sometimes become somewhat hard to find all the values you need in the Locals Window. However, by creating a *Watch*, you can create a list similar to the one given by the Locals Window with the exception that it now shows only the selected variables and values you want to watch. To create a Watch, place the mouse pointer on the variable you want to track or highlight the expression you want to track. Then click Debug on the menu bar, followed by *QuickWatch* which then opens a dialog window that allows you to setup and configure the watch. To delete a watch, right click the watch in the Watch window, then click Delete Watch. Finally, you can clear the current breakpoint by clicking on the red dot on the gray section on the left side of the Code editor and the breakpoint disappears.

In addition to using this amazing IDE debugger and sending program output to the command prompt window, you can also send program output results to the *Output Window*. The Output Window is used by Visual C# to display various status messages including any build errors found during the code compilation process and can also be a very useful tool in debugging code. However, before using this tool, you must add the `using System.Diagnostics;` directive at the start of your code file and then add the output statement `Debug.WriteLine(...);` anywhere you wish to retrieve output information. If this Output Window is not being displayed on your system, you can easily bring it up by clicking *View* in the menu toolbar at the top of the screen and then selecting *Output*.

To summarize, the IDE comes equipped with a full feature debugger that can be very useful in finding and fixing faulty program source code. However, because of the lengthy and complex nature of the debugging process there is far more material than can possibly be included in this brief outline of C#. Interested readers are strongly encouraged to take a few moments to fully familiarize themselves with the latest and greatest debugging tools that come with most recent version of Microsoft's Visual Studio IDE. Throwing exceptions and how to properly handle them will be discussed later in this chapter.

1.6 Overview of the C# Language

The major organizational building blocks of the C# programming language are entities called programs, types, members, namespaces, and assemblies. Project solutions consist of several miscellaneous files that include program files containing source code that may declare types containing members and these can all be organized into namespaces. Examples of members include fields, methods, properties and events. Examples of types include classes and interfaces. Whenever C# programs are compiled, these and other essential files are all physically packaged into assemblies having *.exe* or *.dll* as their file extensions depending on whether they are implemented as applications or dynamically linked libraries.

An assembly that has a single unique entry point is called an application. Whenever an application is started, the execution environment calls a designated method, that is always named Main(), which marks the application's entry point. This method Main() can have any one of the following four specific signatures:

```
static void Main(string[] args) { }
static void Main() { }

static int  Main(string[] args) { }
static int  Main() { }
```

The static keyword indicates that you can access the method `Main()` directly without first instantiating an object of your class. Normally, methods can be called only if you have first instantiated an object, but static methods are special and can be called without first having to create an object. If `Main()` requires an integer return value, it is preceded by the modifier `int` and if no return value is required, it is preceded by the modifier void. Integer return values are used as a return code for the runtime environment. For example, console applications can return traditional result codes between 0 (success) and 16 (fatal error). If parameters are required to execute a program, then a string array of arguments traditionally called `args` is used to hold the command line arguments. More detailed descriptions on all of these topics will be given later in this chapter.

It is possible to also have an assembly without an entry point. In that case, the byte code of the assembly can be reused over and over again in other applications as a referenced component. Visual Studio offers several ways for creating components. One option is to compile the source code files directly into a dynamically linked library file having a `.dll` file extension. Then any application that needs to use that component simply adds a reference to the corresponding `dll` file and it becomes part of the application's private assembly. This is one way to create increasingly larger software applications.

In general, programming languages are essentially nothing more than a collection of instructions for the computer to perform a specific task. As such, each language must follow a unique syntax that characterizes that particular language and any actions taken by a program must first be expressed using explicit statements. A statement is a source code instruction directing the program to execute a certain action. Certain punctuators are also often used to help demarcate the structure of a program. In C#, for example, the semicolon `;` is used to terminate a statement and also allows statements to wrap multiple lines. In addition, delimiters `{` and `}` are used to group multiple statements into what is called a statement block.

Comments strategically placed throughout the source code offer an important way for programmers to record notes and document functionality details of specific sections of code. There are two different ways to produce source code documentation: single-line and multi-line comments. A single-line comment begins with a double-forward slash `//` and continues until the end of the line. A multi-line comment begins with a `/*` and ends with a `*/` and can extend over many lines. Like all computer languages, the C# language contains some keywords that are predefined reserved identifiers that have special meanings to the compiler and therefore cannot be used as identifiers in your program unless they include `\@` as a prefix. For example, `\@struct` is a legal identifier but `struct` is not because it is a reserved keyword. Table 1-2 contains a complete list of reserved keywords in C#.

1.6.1 Data Types

As a program is processed by a computer, its data must somehow be stored in memory. Data can be categorized either as a variable or as a constant. Constants do not change value over the lifetime of a program whereas variable data items have named locations in computer memory that can hold different values at different points in time. Where (the stack or the heap) and how (by value or by reference) the data item is stored in memory including how much memory is required by the data item and what range of values are allowed to be stored in the data item all depend on the data type of the item in question.

The .NET Framework provides a generic Object class so that every other class implicitly inherits the `System.Object` class. This means that the Object class supports

TABLE 1.2
Reserved keywords in C#

abstract	double	int	readonly	true
as	else	interface	ref	try
base	enum	internal	return	typeof
bool	event	is	sbyte	unit
break	explicit	lock	sealed	unlong
byte	extern	long	set	unchecked
case	false	namespace	short	unsafe
catch	finally	new	sizeof	ushort
char	fixed	null	stackalloc	using
checked	float	object	static	value
class	for	operator	string	virtual
const	foreach	out	struct	volatile
continue	get	override	protected	void
decimal	goto	params	public	while
default	if	private	switch	
delegate	implicit	protected	this	
do	in	public	throw	

all the other classes in the .NET Framework and is therefore the root base class for every other class, including user-defined classes. Consequently, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from the Object class. Because all classes in the .NET Framework are derived from the Object class, every method defined in the Object class is available in all objects in the system and you can assign values of any type to variables of type Object. This means that variables can also be thought of as objects that are instantiations of the data type class to which they belong.

Data types in C# fall into four broad categories: value types, reference types, type-parameter types and pointer types. Variables that are value types directly contain and store their own data. In other words, value types simply hold a value. Reference type variables contain only a reference to their actual data. That is, reference type variables do not directly contain their data but instead they hold a memory address that points to the memory location that stores their data. Type-parameter types are associated with generics and pointer types store the memory address of their data but not the actual data itself. Although reference types seem equivalent to pointer types, the way C# handles each type is very different and will be further explained in the sections that follow.

1.6.2 Value Types

Value type variables directly contain and store their data in a single segment of memory in a region called the stack. The stack can be abstractly thought of as an array of memory that acts like a last-in-first-out (LIFO) data structure where data can only

TABLE 1.3
List of Available Value Types in C#

Category		**Description**
value types	simple types	signed integral: sbyte, short, int, long unicode strings: string unsigned integral: byte, ushort, uint, ulong unicode characters: char IEEE floating point: float, double high-precision decimal: decimal boolean: bool
	enum types	user-defined type
	struct types	user-defined type

be added to or deleted from the top of the stack. Placing a data item at the top of the stack is called pushing the item onto the stack. Deleting an item from the top of the stack is called popping the item from the stack. Because of these features, the heap is often used to store temporary data. Value types can be further subdivided into simple numeric types, and user-defined `enum` and `struct` types as shown in Table 1.3. The C# compiler provides for 13 basic simple value types in the System `namespace` as show in Table 1-4. The C# Data Type column lists the data type names you would ordinarily use to declare data type variables in a C# program. These names are actually aliases for those much longer names listed in the column labeled *System Type* found in the System `namespace`. For example, `int` is an alias for `System.Int32` and so on. The Size is the amount of memory in bytes that is taken up by the specific data type. The Description gives a description of the data type. The Range gives the range of allowed values of the data type. The default value is the value that is automatically assigned by the default constructor to variables that are declared but not explicitly initialized.

1.6.3 Reference Types

Reference type variables hold a memory address that points to the memory location that stores their actual data. As such, reference types require two segments of memory upon declaration: The first segment contains the actual data and is allocated in the region of memory called the heap. The second segment is allocated on the stack but contains a reference (*i.e.* memory address) that points to the location in the heap where the data is stored. Because of the way reference types are allocated in memory, they are also referred to as objects in the sense that they are actually instantiations of the data type class to which they belong.

The heap is another region of memory outside of what is allocated for the code and stack during runtime and is used primarily for dynamically allocating and deallocating objects used by the program. The heap is used when the number and size of objects needed by the program are not known ahead of time or when an object is

TABLE 1.4
Pre-Defined Data Types in C#

C# Data Type	**System Type**	**Size (bytes)**	**Description**	**Range**	**Default Value**
byte	Byte	1	unsigned byte	0 to 255	0
sbyte	Sbyte	1	signed byte	128 to 127	0
short	Int16	2	signed short	32,768 to 32,767	0
ushort	UInt16	2	unsigned short	0 to 65,535	0
int	Int32	4	signed integer	2,147,483,648 to 2,147,483,647	0
uint	UInt32	4	unsigned integer	0 to 4,294,967,295	0
long	Int64	8	signed long integer	9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
ulong	UInt64	8	unsigned long integer	0 to 18,446,744,073,709,551,615	0
float	Single	4	floating point	3.4 x 10^{38} to 3.4 x 10^{38}	0.0f
double	Double	8	double-precision floating-point	1.80 x 10^{308} to 1.8 x 10^{308}	0.0d
decimal	Decimal	16	fixed precision number	-7.9 x 10^{28} to 7.9 x 10^{28}	0.0m
char	Char	2	unicode char	u0000 to uFFFF	0
bool	Boolean	1	boolean value	False(0) and True(1) only	0

TABLE 1.5
List of Available Reference Types in C#

Category		**Description**
Reference Types	Class Types	ultimate base class of all other types: object unicode strings: string user-defined types of the form: class
	Interface Types	user-defined types of the form: interface
	Array Types	single and multi-dimensional array types
	Delegate Types	user-defined types of the form: delegate

too large to fit into a stack allocator. Therefore, the heap provides a somewhat more stable data storage area as memory is allocated dynamically and the heap remains in existence for the duration of a program. However, once your program stores items in the heap, it cannot explicitly delete them. Instead, the IDE's garbage collector automatically cleans up orphaned heap objects when it determines that your code will no longer need to access them. Consequently, this unique C# compiler feature frees you from what in other programming languages such as C/C++, can be a very tedious and error prone task. Although reference types in C# are similar to pointers in C/C++, they are much easier to use in C# because all the hard work of keeping track of memory allocation and de-allocation is automatically carried out by the IDE's garbage collector.

Unfortunately, both the value and reference types have their own advantages and disadvantages. Memory allocation on the stack is faster than that on the heap. Hence for small amounts of data, value type variables are recommended over reference type variables. However, reference type variables are more efficient in handling large amounts of data because they pass only the reference, not the entire data value as is the case with value types which can then lead to a lot of extra overhead in memory. By using a reference type instead, only the reference is copied rather than the entire data object. With reference types it is also possible for two or more variables to reference the same object and so it is possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data and so it is not possible for operations on one variable to affect the other. C# reference types can be further subdivided into class types, interface types, array types and delegate types as shown in Table 1.5.

1.6.4 Type-Parameter Types

Type-parameter types were introduced as part of a relatively new C# language feature called generics which came about as a practical way to reduce the need to rewrite algorithms for each data type. Now programmers can create generic classes, delegates, interfaces and methods, postponing the declaration of data types until runtime. This way more generalized code can be written making C# programs even more compact and efficient. For example, consider a method called *swap* that can exchange the value between two integer variables. Prior to the introduction of generics, the programmer would need to write additional *swap* methods for any other data types that might also be needed in an application. With generics, the programmer now needs only to write one generic swap method containing one or more type-parameters, usually denoted by `<T>`, that act like a placeholder for a real data type until the method is called for use in the program. Thus, a generic swap method might look like this:

```
static void swap<T>(ref T var1, ref T var2)
{
   T temp;
   temp= var1;
   var1=var2;
   var=temp;
}
```

To swap a pair of integers previously declared as variables `i` and `j`, one would just substitute the data value identifier `int` for `T` and write:

```
swap<int>(ref i, ref j);
```

Likewise, to swap a pair of strings previously declared as variables `x` and `y`, one would just substitute the data value identifier string for `T` and write:

```
swap<int>(ref i, ref j);
```

In the first instance, the type parameter `T` is replaced with a type `int` and in the second instance it is replaced with a type string and in so doing, one method did the work of two.

1.6.5 Pointer Types

As for pointer types it may come as a complete shock or wonderful news that, by default, C# does not directly support pointer data types in order to maintain data type safety and security. However, by using the unsafe reserved keyword, it is possible to define a context in which pointers can be used. The pointer data type is rather unique because instead of containing data, a pointer contains the memory address of data. In the Common Language Runtime (CLR) layer, unsafe code is referred to as unverifiable code. That is to say, unsafe code in C# is not necessarily dangerous; it is simply code whose safety cannot be verified by the CLR. The CLR will therefore only execute unsafe code if it is within a fully trusted assembly. Therefore, if you use unsafe code in your C# programs, it is your responsibility to ensure that your code does not introduce security risks or pointer errors. Consequently, pointer types are very seldom used in C#.

1.6.6 Variable Declaration

Before you can use a variable, you must first declare and also initialize it to a specific value. The variable declaration defines the variable, gives the variable a name, associates a data type with it and also allows the compiler to allocate memory for it. The syntax for declaring and initializing a value type variable in C# is as follows:

```
type variableName;
variableName = value;
```

However, you can also declare and initialize a variable in a single step like this:

```
type variableName = value;
```

Since variables can also be thought of as objects that are instantiations of the data type class to which they belong, if you declare a variable but fail to initialize it to some specific value, a default initial value will automatically be assigned to the variable by the type's constructor when the variable is declared. A constructor is a method that creates an instance of a class and the keyword new is used to call up the default constructor. For example, the following three integer declarations of the integer variable j are all equivalent:

```
int j;
int j = 0;
int j = new int();
```

1.6.7 Constant Declaration

Constants can be declared and initialized as follows:

```
const type ConstantName = value;
```

1.6.8 Nullable Types

A nullable type is a data value type variable that can store a null value and is usually used to indicate that the value of the variable is unknown. Since reference types can store null values by default, only data value types can be declared nullable types. Nullable types are declared by including a question mark (?) immediately after the keyword for the value type like this: `myValueType? myVariable;` For example, for integers you have: `int? i;` and this means that variable `i` can accept a all the values that can be assigned to an `int` value type in addition to null. All the variables that contain null are displayed as blanks.

1.6.9 Scope

The scope of an item such as a variable or a constant is the section of the program where the item may be accessible by its name and is determined by where you declare it in your code. If you attempt to refer to the item outside its scope, you will get a build error message when you try to compile your code and you will not be able to successfully run your program until you fix this problem. For example, variables declared within a class, can be referred only by all the methods within the class whereas variables declared within a method, can be referred only within the method.

1.6.10 Characters

Characters in C# are declared using the char type. Internally, a character occupies 2 bytes and is stored as a number using the 16-bit Unicode character encoding system which allows one to represent characters of any language. In particular, the Unicode character codes ranging from 0-127 also correspond with the ASCII character encoding [8] scheme. For example, the letter A has a ASCII code of 65 (decimal) = 41 in hexadecimal = \u0041 in Unicode. Therefore a char variable to hold the character 'A' can be declared in any one of the following equivalent ways:

```
char c = 'A';
char c = '\x0041';
char c = '\u0041';
char c = (char)65;
```

1.6.11 Strings

A string consists of a sequence of any characters from the Unicode character set [9] including letters, numbers and even special characters such as `*`, `#` or `&`. Strings are reference types and as such, a string variable holds a reference to a string object. The keyword string is used to declare a string variable, and enclosing the text in double quotes specifies the value of the string. For example,

```
//declares a string variable
string myString;

//declares and initializes a string variable
string myString= Hello ;
```

You can assign an empty string to a string meaning that the value of the string is known but the string does not contain any characters: `string myString="";`. However, if the value of the string is unknown, then the string is called a null string and is declared like this: `string myString=null;` You can also concatenate or append strings using the `+` sign as shown here: `string myString="Hello"+"there!";`

When you declare and initialize a string, you are actually creating a string object from the String class. As a result you can use the properties and methods of the String class to manipulate string objects. Alternatively, you can also use the more versatile `StringBuilder` objects from the `StringBuilder` class so that you can then use the methods and properties from that class to work with strings.

When declaring a string variable certain characters sometimes cannot, for various reasons, be included in the usual way. C# supports two different solutions to this problem. The first approach is to use verbatim string literals. These are defined by enclosing the required string within the characters `\@"..."`. For example, in order to declare a string variable named `fn` and assign it the text: `C:\myfile.txt\` one could write: `string fn = @"C:\myfile.txt\";`.

The second approach is to use something called an *escape sequence* which is the technical term for special characters, alone or within a string that cannot be expressed or interpreted literally. An escape sequence is characterized by a backslash followed by a character having a special meaning. For example, the character for a new line is given by `'\n'` and the character for a backslash is written as `'\\'`. Consequently, in order to declare a string variable named `fn` and assign it the text: `C:\myfile.txt\` one could also now write: `string fn = \"C:\\myfile.txt\\\"`. A complete list of commonly used character escape sequences is given in Table 1.6.

1.6.12 Formatting of Output Data

Proper formatting of output data is very important in order to produce elegant and readable results particularly when the output data is numeric. Output to the console window on the screen is achieved by using the class `System.Console` which has two output methods:

```
Console.Write(x);
Console.WriteLine(x);
```

TABLE 1.6
The Most Common Escape Sequences

\’	-	single quote, needed for character literals
\”	-	double quote, needed for string literals
\\	-	backslash
\0	-	Unicode character 0
\a	-	Alert (character 7)
\b	-	Backspace (character 8)
\f	-	Form feed (character 12)
\n	-	New line (character 10)
\r	-	Carriage return (character 13)
\t	-	Horizontal tab (character 9)
\v	-	Vertical quote (character 11)
\uxxxx	-	Unicode escape sequence for character with hex value xxxx
\xn[n][n][n]	-	Unicode escape sequence for character with hex value nnnn (variable length version of \uxxxx)

The first method writes the value of x to the console window. The second method also writes the value of x to the console window but then advances the cursor to the next line. Both of these methods allow for formatted output of values and for this purpose, a format string with placeholders for a variable number of argument values are passed as parameters. The general format of a placeholder for strings in C# is as follows:

```
{index[,width]:[format[precision]]}
```

where items that are enclosed by the square brackets `[...]` are optional, `index` = argument number (beginning with 0) that specifies which value is to be formatted, and `width` = field width whose absolute value gives the minimum number of characters in the resulting string. If `width` $>$ 0, then string is right aligned (left padded). If `width` $<$ 0 then string is left aligned (right padded). `format` = formatting code (see Table 1.7) and `precision` = number of decimal places (sometimes number of digits). In addition, C# also allows for customized number formatting as shown in Table 1.8.

1.6.13 Type Conversion

Sometimes during the course of writing C# programs, it becomes necessary to convert data from one data type to another. One way to achieve this objective is through a process called *casting*. A *cast* is simply a way to force a value to a different data type. C# provides two types of casting. Implicit casts are performed automatically to convert from a less precise to a more precise data type. For example, a declaration like `float x = 2;` implicitly converts the integer 2 to a float type so that it can be properly assigned to the float variable x. Explicit casts are used to cast data from a more precise to a less precise data type. The new data type is specified between closed parentheses before the old variable to be converted into the new variable as

TABLE 1.7
The Most Common Number Formating Codes

Code	Description	Example
d,D	Decimal format	-xxxxx
f,F	Fixed point format	-xxxxx.xx
n,N	Number format	-xx,xxx.xx
e,E	Floating-point format	-x.xxxE+xxx
x,X	Hexadecimal format	xxxx
c,C	Currency format	$xx,xxx.xx
p,P	Percentage format	x.xx%
g,G	General format (default format)	

TABLE 1.8
The Most Common Custom Number Formating Codes

Specifier	Type	Format	Output Example given input = 1234.56
0	zero placeholder	0:00.000	1234.560
#	digit placeholder	0:#.##	1234.56
.	decimal point placeholder	0:0.0	1234.6
,	thousand separator	0:0,0	1,235
%	percentage	0:0%	123456%

shown below:

```
new_variable = (DataType) old_variable;
```

Note that in casting from a more precise to a less precise data type some data information may be lost if the less precise data type is not large enough to accommodate the data type of the original value being casted. Consequently, the resulting value is truncated rather than rounded. In addition, your program may also throw an exception at runtime if the range of allowed values in the new and less precise data type variable does not fall within the bounds of the allowed data values from the original expression. For example,

```
float x = 2.3;
short s = (short)x;
```

explicitly converts a float to a `short` data type truncating the `3` to yield a final value of `s=2`. However, the following cast

```
int i = 32768;
short s = (short)i;
```

will cause the C# compiler to throw an exception because the data value 32768 exceeds the maximum allowed size of the short data type which is 32767. Finally, it's also important to remember that when applied to arithmetic expressions, the casting is done before any other arithmetic operations are carried out.

The as operator works similarly to a cast. The as keyword is used to cast an object to a different data type. However, the type being cast to must be compatible with the original type. The general format of using the as operator is as follows:

```
new_variable as DataType
```

Although using the as keyword is similar to a cast, it is not the same. If you use a cast and there is a some kind of problem, an exception is thrown. With as, if there is an error in changing the variable expression to the desired DataType, the variable expression is set to the value of null and converted to the DataType anyway and no exception is thrown. This feature makes using the as keyword safer than doing a cast.

To check and see if a certain variable object is of a specified type, C# uses the is keyword. The general format of using the is keyword is as follows:

```
(expression is DataType)
```

If the expression is compatible with the DataType, this returns true, otherwise it returns false.

Another way to convert data from one data type to another is through the use of the various static methods provided by the sealed System.Convert class. The general format for using the Convert class to convert from one data type to another is:

```
Convert.method(value);
```

where method is the name of the conversion method you want to use and value is the original data value you want to convert. The results of the conversion may vary depending on the type of conversion being sought and, in some cases where C# may not be able to perform the requested conversion, a runtime error exception may be thrown. For example, say you have a string variable x declared and assigned the value of 5: string x="5";. Then to convert x to some integer variable, say i, one could write: int i = Convert.ToIn32(x);. The most commonly used static methods of the Convert class are: ToDecimal(value), ToDouble(value), ToInt32(value), ToChar(value), ToBool(value) and ToString(value) where the value is the item that is converted to the specified data type.

The ToString([format]) method is a particularly useful since it allows you to convert any value to a string and at the same time also format the resulting output values using the codes described in tables 1-7 and 1-8. For example,

```
double price = 29.95;

//Make implicit call to ToString method
string msg1 = Price:   + price;

//Displays output as: Price: 29.95
Console.WriteLine({0},msg1);

//Makes explicit call to ToString method
string msg2 = price.ToString( c );

//Displays output as: Price: 29.95
Console.WriteLine( Price: {0} ,msg2);
```

You can also format numbers using the Format method of the `String` class. Since this is a static method, you can access it directly from the `String` class without first having to create an instance of that class. However, you must provide two arguments: the first argument is a string literal containing the format specification for the value to be formatted and the second argument is the value to be formatted. Using data from the code snippet just described one can also write:

```
//Explicit use of the Format method of the String class
String msg3 = String.Format( {0:c} ,price);

//Displays output as: Price: $29.95
Console.WriteLine( Price: {0} ,msg3);
```

Conversely, the `Parse()` method allows you to convert a specified string value to a specified data type value. For example,

```
float newprice = float.Parse(str2);
```

converts the contents of the string variable `str2` to a float variable called newprice having the value of 29.95.

Since the value of any data type can ultimately be treated as an object, it is also possible to convert a value type to a reference type, and back again to a value type, by using a process called boxing and unboxing, respectively. For example, in the following code snippet, an int value type is converted to object type and back again to an `int` value type.

```
int i=799;
object obj = I;    //boxing
int j = (int) obj; //unboxing
```

Boxing and unboxing provides a way to convert between value types and reference types by permitting any value of a value type to be converted to and from type object and so value types can become objects on demand.

1.6.14 Reading Keyboard Input Data

To read keyboard input, use the .NET method `ReadLine();` which always returns the input value as a string object. If numeric data was entered, then it must first be converted from the input string to the desired numeric value as shown in the following example:

```
Console.Write("Type in an integer:");
string s = Console.ReadLine();
int i = Convert.ToInt32(s);
Console.WriteLine("You entered: {0}", i);
```

Alternatively, the code above can also be written as follows:

```
Console.Write("Type in an integer:");
int i = int.Parse(Console.ReadLine());
Console.WriteLine("You entered: {0}", i);
```

TABLE 1.9
The Basic Arithmetic Operators

Type	Operator Name	Does what?	Example	
binary	`+`	addition	adds two operands	`x + y;`
binary	`-`	subtraction	subtracts two operands	`x - y;`
binary	`*`	multiplication	multiplies two operands	`x * y;`
binary	`/`	division	divides two operands	`x / y;`
binary	`%`	modulus	returns remainder obtained from dividing two operands	`x % y;`
unary	`+`	positive sign	returns value of operand	`+x;`
unary	`-`	negative sign	changes sign of operand	`-x;`
unary	`++`	increment	adds one to the operand	`x++;` or `x = x + 1;`
unary	`--`	decrement	subtracts one from the operand	`x--;` or `x = x - 1;`

1.6.15 Basic Expressions and Operators

Expressions are constructed from operands and operators. The operator of an expression indicates which operation to apply to the operands. The general format for writing an expression is as follows:

```
assignmentVariable = operand1 operator operand2;
```

There are three kinds of operators that are widely used and they are: unary operators, binary operators and conversion operators. Unary operators operate on one operand whereas binary operators operated on two operands. A conversion operator converts from a source type as indicated by the parameter type of the conversion operator, to a target type, as indicated by the return type of the conversion operator. Type casting, for example, provides a good illustration of using a conversion operator for practical applications. Unary and binary operators, however, require some additional discussion and are listed in Table 1.9.

Arithmetic expressions are coded using arithmetic operators to indicate what operations are to be performed on the operands in an expression. Operands can be either a literal or a variable. Increment and decrement operators can either precede or follow a variable depending on whether you want the variable updated before (prefix) or after (postfix) the expression is evaluated. The general behavior of increment and decrement operators is summarized in Table 1.10 and is illustrated in the following examples:

```
int i=0;
Console.WriteLine(i++); //Outputs 0 and i is now 1

int i=0;
Console.WriteLine(++i); //Outputs 1 and i is now 1
```

The assignment operator, `=`, is used for assigning a value, expression or another variable to a variable. The simplest example of using the assignment operator is as follows: `variableName = expression;`. The assignment operator can also be used

TABLE 1.10
Increment and Decrement Operators

Expression	Operation	Interpretation
`x = ++i;`	Preincrement	`i = i + 1;` `x = i;`
`x = i++;`	Postincrement	`x = i;` `i = i + 1;`
`x = --i;`	Predecrement	`i = i - 1;` `x = i;`
`x = i--;`	Postdecrement	`x = i;` `i = i - 1;`

TABLE 1.11
Assignment Operators

Operator	Name	Example	Equivalent to
`=`	assignment	`x = value;`	`x = value;`
`+=`	compound addition	`x += value;`	`x = x + value;`
`-=`	compound subtraction	`x -= value;`	`x = x - value;`
`*=`	compound multiplication	`x *= value;`	`x = x * value;`
`/=`	compound division	`x /= value;`	`x = x / value;`
`%=`	compound modulus	`x %= value;`	`x = x % value;`

in conjunction with the standard basic arithmetic expressions, including the modulus operator, to write shorter and more compact expressions as described in Table 1.11. To avoid ambiguity in calculations involving multiple arithmetic expressions, a specific order of precedence has been established as indicated in Table 1.12. Relational or comparison operators are used to create Boolean expressions to compare two operands and return a Boolean value. Table 1-13 lists the relational operators available in C#.

Logical operators are used to perform both bit manipulation and to combine relational operators in order to build a more elaborate logic or Boolean expression whose final output is either true or false. Table 1-14 lists the logical operators that are available in C#.

TABLE 1.12
Operator Order of Precedence

Type	Operators	Direction of Operation
unary	`+ - ++ --`	right to left
multiplicative	`* / %`	left to right
additive	`+ -`	left to right
assignment	`= *= /= %= += -=`	right to left

TABLE 1.13
Relational Operators Available in C#

Operator	Description
`>`	Greater than
`>=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to
`==`	Equal to
`!=`	Not equal to

TABLE 1.14
Logical Operators Available in C#

Operator	Name	Description
`&&`	Conditional-AND	Returns a true value only if both expressions are true. Only evaluates second expression if necessary.
`||`	Conditional-OR	Returns a true value only if either expression is true. Only evaluates second expression if necessary.
`&`	AND	Returns a true value if both expressions are true and it always evaluates second expression.
`|`	OR	Returns a true value if either expression is true and it always evaluates second expression.
`!`	NOT	Reverses the value of the expression.

TABLE 1.15
Unambiguous Numeric Suffixes in C#

C# Type	Example
`float`	`float x = 2.0f;`
`double`	`double y = 4.0d;`
`decimal`	`decimal z = 7.0m;`
`uint`	`uint I = 8u;`
`long`	`ulong j = 9ul;`

TABLE 1.16
Special Value Constants in C#

Special Value	Double Constant	Float Constant
NaN	`double.NaN`	`float.NaN`
$+\infty$	`double.PositiveInfinity`	`float.PositiveInfinity`
$-\infty$	`double.NegativeInfinity`	`float.NegativeInfinity`
-0	`-0.0`	`-0.0f`

By default, the C# compiler infers a numeric literal to be either a double or an integral type and so numeric suffixes sometimes must be added to explicitly define the type of literal that is actually wanted. A list of the available numeric suffixes is shown in Table 1.15. In addition, floats and doubles also have special values that arise in certain operations. These special values are `NaN` (Not A Number), $+\infty$, $-\infty$, and -0 and are summarized in Table 1.16.

1.6.16 Program Flow Mechanisms

C# has the following mechanisms to conditionally control the flow of program execution:

- Selection statements (`if`, `switch`)
- Loop sequences (`for`, `while`, `do-while`, `foreach`)
- Conditional Operator (`? :`)

To sum up, selection statements are used to select one of a number of possible statements for execution based on the value of some expression. Example: `if` and `switch` statements. Loop sequences are used to repeatedly execute a statement. Examples include the `while`, `do`, `for` and `foreach` statements. Conditional operators provide a short way to write a simple `if-else` structure. In view of their usefulness, all these program flow mechanisms are important enough to warrant a more detailed discussion.

Selection Statements

The `if-else` statement allows you to select different actions based on results of Boolean expressions. It is used to build conditional statements and takes on the general format as shown below:

```
if (booleanExpression) {statements}
[else if (booleanExpression) {statements}]
...
[else {statements}]
```

The square brackets `[]` indicate that a clause is optional whereas the ellipsis (...) indicate that the preceding element can be repeated as many times as needed. If more than one statement is used then you must enclose those statements within a block using the brackets: `{}`.

The `Switch` statements allow you to select only one of the available choices from among multiple choices. The general switch construct takes on the following format:

```
switch (expression)
{
case result1:
        statement(s)
        break;

[case resultN:
        statement(s)
        break;]

...

        [default:
        statement(s)
        break;]
}
```

where expression evaluates a result value that corresponds to one of the possible listed case labels. The switch statement then transfers control to the corresponding case label and a statement or a block of statements are processed provided the corresponding value `resultN` is evaluated to be true. If control is not transferred to one of the case labels, the code following the optional default label is executed. The break statement exits the switch statement and must be included at the end of every case block. If a case does not contain any statements, code execution will fall through to the next label and default is an optional case to deal with any other case that is not included in the list.

Loop Sequences

Loop sequences are used to repeat one or more statements a specific number of times, until a specific condition is satisfied or go on indefinitely.

The `for` loop repeats an operation for as long as a specified Boolean condition is satisfied and has the general form:

```
for ([initialization];[Boolean expression];[counter update];)
{
   statement(s);
}
```

where `initialization` is the counter initialization statement, `Boolean expression` is the condition to be satisfied during the processing of the loop, `counter update` is the counter increment or decrement statement, and `statement(s)` is the statement or block of statements to be repeated. `for` loops can be nested inside each other but the counter variable must be unique to the loop to which it is assigned. `for` loops are useful when you need to increment or decrement a counter that determines how many times the loop is executed. The enclosing brackets `[ ... ]` indicate that these features are optional.

The `while` loop is used to process a block of statement(s) for as long as a specified Boolean condition is satisfied. In addition, the Boolean expression is tested before the `while` loop is executed. The `while` loop construct has the following format:

```
while (Boolean expression)
{
   statement(s);
}
```

The `do-while` loop is used to process a block of statement(s) for as long as a specified Boolean condition is satisfied. However, the Boolean expression is tested after the do-while loop is executed and so the statement block is executed at least once regardless of the result obtained from the Boolean expression. The `do-while` loop construct has the following format:

```
do
{
   statement(s);
}
while (Boolean expression);
```

The `foreach` statement iterates over each element in an enumerable object and has the following general syntax:

```
foreach (type elementName in arrayName)
{
   statement(s);
}
```

Fortunately, most of the types in C# and the .NET Framework that represent a set or list of elements are enumerable. For example, to enumerate over the characters in a string, one could write:

```
string s = H e l l o ;
foreach(char c in s)
    Console.Write( {0}  ,c);
```

As with everything, the `foreach` loops have a few restrictions. First, you cannot access individual array elements but only the entire array. In addition, the `foreach` loop allows read access only and so we cannot modify the array. To access and/or modify individual array elements we must use the for loop as described earlier.

Conditional Operator

The expression "`booleanExpression ? Statement_1: Statement_2;`" provides a short way to write a simple `if-else` construct where if `booleanExpression` is true then do `Statement_1` else do `Statement_2`. For example, the code below can be used to calculate the `sinc` function.

```
static double sinc(double x)
{
  return x != 0.0 ? Math.Sin(x)/x : 1.0;
}
```

1.6.17 Jump Statements

Jump statements are used to transfer control. The jump statements in C# are: `break`, `continue`, `goto`, `return` and `throw`.

The `break` statement is used to completely stop the execution of the remaining items in the body of a `while` loop, a `for` loop, or a `switch` statement. Program control is then transferred to the statement that follows the loop.

The `continue` statement is used to jump back to the start of a loop thus forgoing any additional remaining statements in the loop. Note that by using the `continue` statement you can cause the counter to skip a certain value.

The `goto` statement transfers execution to another label within the statement block and the general form is given by: `goto statement-label;` A label statement is just a placeholder in a code block and is written with a colon suffix.

The `return` statement exits the method and must return an expression of the method's return type if the method is not void. The `return` statement can appear anywhere in a method.

The `throw` statement throws an exception to indicate some kind of error has occurred and will be discussed in more detail later in this chapter.

1.6.18 Arrays

An array is a fixed size, indexed collection of data elements of one specific data type. Arrays are reference types and can be one or multi-dimensional. Its elements are distinguished from the others by an index starting at 0 and going up to $n - 1$ where n is called the size or length of the array. Arrays are declared in two steps. First, the type of array is declared and a reference variable is created. Second, space is allocated and fixed for the specified number of elements using the new operator. The new operator automatically initializes the elements of an array to their default value, which for example, is zero for all numeric types and null for all reference types.

The general syntax for declaring a one-dimensional array is:

```
arrayType[] arrayName;
arrayName = new arrayType[arrayLength];
```

These two steps are often combined into one step:

```
arrayType[] arrayName = new type[arrayLength];
```

And you refer to an element of an array by coding the array name followed by its index in brackets as shown: `arrayName[index];`

In C#, arrays are objects (*i.e.* instances of a class named `System.Array`) and so if you are unsure what type of data an array will contain, you can always declare an array of object types so each element of the array can then contain data of any type. In addition, since arrays are instances of the `System.Array` class, you can also use the properties and methods of this class to work with your arrays. For example, the length of an array can be easily found by looking at the value obtained from `arrayName.Length;` Likewise, you can also sort your array and even search for a specific element in your array using the methods provided by the .NET Framework's Array Class.

It is also possible to create an array and assign values to it in one statement as shown:

```
type[] arrayName = [new type[length]] {value1 [,value2]...};
```

The following shows three different ways to declare the same integer array and assign three data elements to it:

```
int myInt = new[]{10,15,25};

int[] myInt = {10,15,25};

int[] myInt = new int[3];
myInt[0] = 10;
myInt[1] = 15;
myInt[2] = 25;
```

There are two ways to traverse through each and every element of an array. One way is to use a for loop as shown in the following example:

```
int[] x = new int[10];
for (int i=0; i<x.Length; i++)
 Console.WriteLine( x [ +i.ToString()+ ]={0} ,x[i].ToString());
```

The other way is to use the foreach loop where the previous example can be written as:

```
int[] x = new int[10];
foreach (int i in x)
{
 Console.WriteLine( x [ +i.ToString()+ ]={0} ,x[i].ToString());
}
```

There are times, however, when you do not want to traverse through each element of an array but instead only through a selected number of elements. In that case you cannot use the `foreach` loop but instead must use the `for` loop.

Arrays can also have 2 dimensions in which case they are called rectangular or two-dimensional arrays. The syntax for declaring two-dimensional arrays are as follows:

```
type[,] arrayName = new type[rowCount,columnCount];
```

and the syntax for referring to an element of a two-dimensional array is:

```
arrayName[rowIndex, columnIndex];
```

To extract the number of rows or columns in a two-dimensional array you need to use the `GetLength()` method as shown below:

```
arrayName.GetLength(dimensionIndex);
```

where `dimensionIndex = 0` for rows and `dimensionIndex = 1` for columns.

C# also supports higher multi-dimensional arrays following the same kind of constructs. The number of dimensions of an array type, also known as the rank, is one plus the number of commas written between the square brackets of the array type. An element type of an array can be any type, including an array type. An array with elements of an array type is sometimes called a jagged array because the lengths of the element arrays do not all have to be the same.

1.6.19 Enumerations

An `enum` type is a distinct user defined value type with a set of named constants known as members of the enumeration. By default, an enumeration uses the `int` type and sets the first constant to `0`, the second constant to `1` and so on. Alternatively, you can also specify your own data type and member values. The general syntax for declaring an enumeration is:

```
enum EnumerationName [:type]
{
  ConstantName1 [=value]
 [,ConstantName2 [=value]]...
}
```

where the items enclosed by [...] are optional.

For example, `enum Color {red, white, blue};` means that the complier maps `red=0`, `white=1` and `blue=2` by default. However, the values of the enumeration constants can also be specified explicitly as in the declaration:

```
enum Color:int {red=2, blue=5};
```

1.6.20 Structures

`Struct` types are data structures similar to classes in the sense that they can also contain constructors, constants, fields, methods, properties, indexers, operators, events and nested types. However, unlike classes which are reference types, structs are value types and therefore require no heap allocation. `Struct` constructors are invoked with the `new` operator but instead of dynamically allocating an object and returning a reference to it, a struct constructor simply returns the struct value itself, typically in a temporary location on the stack, and this value is then copied as often as necessary. When you declare a variable with a `struct` type, an instance of that structure is created but values are not automatically assigned. Instead, a default constructor must always be provided to initialize the structure members to their default values. Unlike classes, structs cannot be inherited by other structs or by other classes. In addition, structs cannot have an empty constructor.

The general syntax for declaring a structure is this:

```
[attributes] [modifiers] struct identifier [:intefaces]
{
   structure members;
}
```

where the square brackets `[]`, indicate that the listed item is optional. Attributes provide additional declarative information. Modifiers may be new, public, protected, internal or private. The identifier is the variable name of the `struct`. Interfaces provide a comma-separated list of the interfaces implemented by the `struct`.

Structs are particularly useful for small data structures that have value semantics. The use of structs rather than classes for small data structures can make a significant difference in the number of memory allocations an application performs. As for limitations, copying an entire `struct` is typically less efficient than simply copying an object reference. Therefore, assignment and parameter passing with structs can be more memory expensive than with reference types. Also, except for using ref and out parameters and since structs are value types, it is impossible to create references to structs, and this potentially useful feature eliminates them from serious consideration for use in a number of additional applications.

1.6.21 Exceptions

Exceptions are essentially runtime errors, such as attempting to divide by zero, which can suddenly stop program execution unless they are handled properly. The handler is a block of code that catches the exception, does something with it and then attempts to continue with the program execution. If your code is not setup to catch exceptions and accommodate handlers, the default handler is used and the program crashes. In C#, an exception is an object of the class `System.Exception` which represents an error during program execution. Since an exception is an object, it has properties and methods and you can use these features to extract information with regards to the kind of runtime error your code experienced. In its most general format, the try-catch exception handler looks like this:

```
try
{
   statement(s);
}
[catch(MostSpecificException [exceptionName])
 {statement(s);}] ...
 catch([LeastSpecificException [exceptionName]])
 {statement(s);}
[finally {statement(s)}]
```

A catch block may be coded for each type of exception that may occur in the try block. If more than one catch block is coded, you must code the catch blocks for the most specific types of exceptions first. Since all exceptions are subclasses of the Exception class, a catch block for the Exception class will also catch all types of exceptions. A finally block can be added after all the catch blocks and the code in this block is always executed whether or not an exception occurs. Consider the following example:

```
try
{code statements;}
catch(OverFlowException) //a specific exception
{code statements;}
catch(Exception ex)      //all other exceptions
{
   //Display error message and error type
   //or do some other error handling here
   Console.WriteLine(ex.Message +  \n\n  +
        ex.GetType().ToString() +  \n\n  +
        ex.StackTrace().ToString());
}
finally //this code runs whether or not an exception occurs
{code statements;}
```

The syntax for throwing an existing exception is: "`throw exceptionName;`". For example,

```
if (x == 0) throw ArithmeticException;
```

You can also throw customized exceptions specific to your own code. The syntax for throwing a new exception is:

```
throw new ExceptionClass([message]);
```

For example,

```
if (x == 0) throw new Exception( x  cannot have a value of 0 );
```

1.6.22 Classes

A *class* is a fundamental data structure whose role is very much like that of a blueprint in that it contains all the necessary information and detailed instructions for dynamically creating and manipulating any number of instances of the class itself, also known as objects. An *object* is said to be an instance of the class from which it is created and the process of creating an object is called *instantiation*.

Classes are created using class declarations. A class declaration defines the characteristics and members of a new class. It does not create an instance of the class but creates the template from which class instances can be created. A class declaration starts with a header that specifies the attributes and modifiers of the class, the keyword class, the name of the class, the base class (if any) and the interfaces implemented by the class (if any). The header is followed by the class body, which consists of a list of member declarations written between the delimiters { and }. The general syntax of a class declaration is given by:

```
[attributes] [modifier] class className [:base-list]
{
    class members;
}
```

Attributes may provide additional declarative information. For example, class members are either static members or instance members. Static members are associated with classes and instance members, which are the default, are associated with objects (instances of classes).

Optional access modifiers are used to specify the degree of accessibility of the members declared by a class to other regions of the program. There are five possible forms of accessibility and these are summarized in Table 1.17.

In addition, class members consist of data members and function members. Data members store data associated with the class whereas function members execute

TABLE 1.17
Class Accessibility Options in C#

Accessibility	Description
public	Members are accessible to all other classes.
private	Members are accessible only to the class in which they have been defined.
protected	Members are accessible only to the class in which they have been defined and any derived classes.
internal	Members are accessible by other classes in the same assembly, but not by classes in other assemblies.
protected internal	Members are accessible only to the current class, derived classes, or classes in the current assembly.

code. Data members include fields and constants. Fields are variables declared within a class. Constants contain values that never change throughout the life of the program. Function members include methods, properties, constructors, destructors, operators, indexers, events and delegates. Methods are simply operations that can be performed by an object. Properties refer to data values that are associated with the creation or instantiation of particular objects. Constructors consist of a special type of method that is always executed whenever an object is instantiated. Likewise, Destructors consist of a special type of method that is always executed whenever an object is destroyed. Operators are a special type of method that is executed for another C# operator. Indexers allow you to store data in objects and later refer to them by an index just like arrays. Events are signals to notify other objects that something significant has taken place. Delegates are special types of objects that are used to wire an event to a method. Indexers, delegates and events will be discussed later in this chapter. Of all these, the most commonly used class members are just properties, methods and constructors.

To create a user-defined class, you first need to add a class file to your project and this is accomplished in the Visual Studio IDE by going to the menu bar on top of the screen and clicking Project followed by Add Class. In the dialog box that pops up, enter the name you want to call your new class and click the Add button. This action will add a class file having the name of your choice to the Solution Explorer window. The IDE will add the `namespace` and class blocks automatically to the class file you just created.

Declaring a class does *not* create an object. A class is just an abstract description of an object if it is ever instantiated by the class. Classes are reference types and so variables of the class type require memory for both the reference to the data and for the actual data itself. Consequently, instantiating an object is actually a two step process. First, you declare a variable of the class type in order to allocate memory to hold a reference to the object. Then you create the actual object by using the `new` operator to invoke a constructor to initialize the instance and return a reference to the instance of the class. Therefore, the syntax for creating an object in C# is as follows:

```
ClassName ObjectVariable;
ObjectVariable = new ClassName([parameter list]);
```

Alternatively, you can also instantiate an object from a class in one line:

```
ClassName ObjectVariable = new ClassName([parameter list]);
```

The concept of *inheritance* allows you to create a new class based on an existing class. The new class can use all the features of the original class marked with protected or greater access (except for constructors), it can override existing features, it can extend existing features, or it can add its own features. The original class is called the *base* or *parent* class. The new class, which was created by inheriting features from the base class, is called the *derived* or the *child* class. To inherit from a class, the following format is used:

```
class derived_class : base_class;
```

Calling a base method contained in a base class by a derived method contained in a derived class is a very common programming practice. However, which method gets called when both the base and the derived methods have the same name can be a source of considerable confusion for both the programmer and the compiler. Fortunately, virtual methods have been developed to resolve this ambiguity. A *virtual method* enables you to call the method associated with the actual assigned type instead of the one contained in the base class. A method is declared as *virtual* within the base class and a deriving class must then indicate when the method is to be overridden. This is done by using the `override` keyword when declaring the new method. You can force a class to override a method by declaring the method in the base class to be *abstract*. An *abstract method* is not given a body. Instead, derived classes are expected to supply the body. Whenever a method is declared as abstract, so must the class be declared as abstract. Abstract classes are created with the expectation that other classes will be derived from them. To prevent inheritance from a class, you need to use the `sealed` keyword when defining the class as sealed classes cannot be inherited.

In some other object-oriented programming languages, such as C++, a class can inherit more than one class and this feature is called *multiple inheritance*. In C#, however, a class can inherit only one class. Nevertheless, C# provides the functionality and benefits of multiple inheritance by enabling multiple interfaces to be implemented instead. Although a class can inherit from only one other class, it can implement multiple interfaces. In addition, structures cannot inherit from other structures or classes but they can, however, also implement different interfaces. One of the benefits of implementing interfaces instead of inheriting from a class is that you can implement more than one interface at a time and this feature allows you to do a much cleaner version of multiple-inheritance than those available in other object oriented programming languages.

An *abstract class* is a class that can be inherited by other classes but it cannot be used to instantiate an object. An *abstract method* is a method that must be overridden when inherited. An *interface* is like a pure abstract class in the sense that abstract classes are classes that generally contain at least one abstract method. However, an interface differs from an abstract class in a number of ways. First, an interface does not provide any implementation of code. Instead, a class that implements an interface is also required to provide the implementation code. Second, within an abstract class, some methods can be abstract, while others need not be. Within an interface, however, all methods must be abstract. An interface also differs from a class in that all of the members of an interface are assumed to be public. Therefore, an interface is said to provide guidelines but not specific code implementation details. In addition, interface members can only consist of virtual methods, properties, events and indexers but not data members, constructors, destructors or static members. The general declaration of an interface is:

```
[public] Interface IName
{
   members;
}
```

where IName is the name of the interface and can be any name you chose. Traditionally, however, interface names start with an I to indicate that this data structure is an interface. To implement multiple interfaces, you just separate each interface with a comma as shown:

```
Class myClass: IName1, IName2, ...
{
   ...
}
```

The general format for declaring a property within an interface is as follows:

```
[public] dataType name
{
   get; // do not show additional code here
   set; // do not show additional code here
}
```

and, like all members of an interface, is also assumed to be public. Finally, as with classes, an interface can be derived from another interface and this inheritance of interfaces is done in a similar manner to inheriting classes.

Constructors and Destructors

Whenever an object is instantiated, a constructor is called up to initialize the data that makes up the object and this data defines the object's state. Should a class not provide its own constructor, an automatic internal default constructor is used instead. A *constructor* is a special method within a class that gets automatically called up whenever an instance of the class is created. Constructors usually perform initialization or setup operations to help construct an object. If you do not provide a user-defined constructor, C# automatically uses a default constructor that assigns default values to all the variables in the newly instantiated object. Constructors can be overloaded and you can write as many constructors for a class as you want as long as their parameter lists are unique so as to not create ambiguity for the C# compiler.

To create a constructor, you must declare a public method with the same name as the class name and it must not contain a return type. Within the constructor you initialize the instance variables and include any additional statements you want to be executed when an object is instantiated from the class. To code a constructor that has parameters, code a data type followed by the parameter name for each parameter within the parentheses that follow the class name. The method header for a constructor has the following syntax:

```
public ClassName([parameter list])
{
   Statement(s);
}
```

By contrast, a *destructor* is a special method within a class that gets automatically called up whenever an instance of a class is destroyed. Most often, an instance of a class is destroyed when it is no longer in use and the process is then automatically handled by the IDE's garbage collector. As with constructors, if you do not explicitly create a destructor for a class, C# automatically provides one. To explicitly declare a destructor, use an identifier that consists of a tilde (∼) followed by the class name as shown here: ∼ `ClassName(){ ... }`. Destructors cannot have accessibility modifiers, be invoked explicitly, or take any parameters and they must have an empty argument list. Consequently, destructors cannot be overloaded and a class can have at most one destructor and like a constructor, a destructor has no return type.

Properties

A property is a class member that provides the means with which programmers can interact with otherwise private data members of the same class. A property consists of a named set of two matching methods called accessors. The set accessor is used for assigning a value to the property and the get accessor is used to retrieve a value from the property. A property does not allocate memory for data storage, it executes code. By tradition, a property identifier has the same name as the field it manipulates, except that the first letter is capitalized. Properties containing both a get and a set accessor are called read/write properties. A property that only has a get accessor is called a read-only property and a property that only has a set accessor is called a write-only property. The general syntax for coding a property is:

```
modifier returnType propertyName
{
   get { return something; }
   set { something = value; }
}
```

Properties are often associated with fields. A common practice is to encapsulate a field in a class by declaring it private and declaring a public property to give controlled access to the field from outside the class. This technique ensures that data will be used and changed only in the ways provided by your accessors.

Methods

A method is an encapsulated series of statements that carry out a specific computation or action. The signature of a method consists of the name of the method and the

TABLE 1.18
Method Accessibility Options in C#

Accessibility	Description
public	Method accessibility is unlimited.
private	Method is accessible only to the class in which they have been defined.
protected	Method is accessible only to the class in which they have been defined and any derived classes.
internal	Method is accessible by other classes in the same assembly, but not by classes in other assemblies.
protected internal	Method is accessible only to the current class, derived classes, or classes in the current assembly.

number, modifiers and types of its parameters. The signature of a method must be unique in the class in which the method is declared. The general syntax for coding a method is

```
[attributes] [access modifier] returnType MethodName([parameterList])
{
    statement(s);
}
```

As in the case with classes, method attributes may provide additional declarative information. For example, methods are either static or non-static. Methods are non-static by default and a non-static method is accessible only through objects instantiated by the associated class. Static methods, however, can be called directly from the corresponding class itself.

Optional access modifiers are used to specify the degree of accessibility of the methods declared by a class to other regions of the program. There are five possible forms of accessibility and these are summarized in Table 1.18.

A method can return at most one value to the method that calls it. The return statement, which is the last statement within the method, causes a value to be sent back to the calling method. To code a method that does not return data use the `void` keyword. To code a method that returns data, code a `return` type in the method declaration and code a return statement in the body of the method. The general syntax for calling methods is as follows:

```
[this.]MethodName([parameterList])
```

The `this` keyword is optional and serves to indicate that the method being called is in the current class. Items inside the parenthesis of a method heading are called parameters or arguments. Some programmers prefer to make a significant distinction between parameters and arguments with parameters referring to items appearing in the heading of a method and arguments referring to items sent into the method through a method call.

Within the parentheses of a method, you can also code an optional parameter list that contains one or more parameters, separated by commas, that allows data to be passed into and out of the method. Individual parameters must include both a data type followed by an identifier variable name. The general syntax for individual parameters in a parameter list is as follows:

```
[modifier] dataType variableName
```

There are four kinds of parameters: *value* parameters, *reference* parameters, *output* parameters and *parameter arrays*. An optional modifier may be used to instruct the compiler how to pass the particular parameter. Method parameter values can be passed by *value* or by *reference*. By default, parameters are passed by value without the need to specify any modifier.

If you pass a parameter by value, then only a copy of the parameter's value is passed into the calling method, not the actual variable itself. As a result, if a parameter variable is changed while inside a method, it has no effect on the corresponding values of the variable outside the method. Therefore, passing parameters by value also means that the original value of the parameter variables cannot be permanently changed by the calling method. Consider the following example below illustrating the effect of passing a parameter by value:

```
public static void Main()
{
  int x = 10;
  Console.WriteLine("Before x = {0}", x);
  ChangeVarByValue(x);
  Console.WriteLine("After  x = {0}", x);
}

public static void ChangeVarByValue(int x)
{
  x = 0;
}
OUTPUT:
Before x = 10
After  x = 10
```

If instead you pass a parameter by reference, the passed parameter provides a memory reference that points to the variable in the calling method and so the actual parameter, not its copy, is effectively passed into the calling method. As a result, if the called method changes the value of the parameter that was passed by reference, then the value of the variable in the calling method is also changed. Reference parameters are declared like regular variables except for the `ref` modifier placed in front of them as shown in the example below that illustrates the effect of passing a parameter by reference:

```
public static void Main()
{
   int x = 10;
   Console.WriteLine("Before x = {0}", x);
   ChangeVarByReference(ref x);
   Console.WriteLine("After  x = {0}", x);
}
```

```
public static void ChangeVarByReference(ref int x)
{
  x = 0;
}
OUTPUT:
Before x = 10
After  x = 0
```

Output parameters are used to extract multiple return values back from a method. Output parameters are declared with the `out` modifier and, like the variables preceded with the `ref` modifier, are also passed by reference. However, output parameters do not need to be assigned before going into a method but must be assigned before it comes out of the method. Consider that following example that illustrates how output parameters are extracted from a method:

```
public static void Main()
{
  x=5;
  int tot=0;
  Console.WriteLine("The original number is {0}", x);
  DoSum(x, out tot);
  Console.WriteLine("The final sum is {0}", tot);
}

public static void DoSum(int x, out int total)
{
  Console.WriteLine("Adding 7 to the number {0}", x);
  total = x + 7;
  return;
}
```

This produces the following output:

```
The original number is 5
Adding 7 to the number 5
The final sum is 12
```

Parameter arrays may be used if the number of parameters sent to a method is not known in advance. You can declare a parameter array using the keyword `params`, within the method header and then the method will be able to accept any number of parameters. However, only one `params` keyword is permitted in a method declaration and no additional parameters are permitted after the `params` keyword in a method declaration. Since arrays are passed by reference and we have an array of parameters, these too are passed by reference. For example, in the following method the passed parameters could be any type such as strings or integers:

```
public static void Display(params Object[] things)
{
   foreach(Object obj in things)
       Console.WriteLine( {0} , obj);
}
```

Table 1-19 summarizes all the various modifier options that are available for passing parameters into and out of methods.

TABLE 1.19
Summary of Optional Parameter Modifiers in C#

Optional Parameter Modifier	Passed By	Variable must be assigned
none	value	going IN
ref	reference	going IN
out	reference	going OUT
params	reference	going IN

1.6.23 Indexers

An indexer enables you to use an index to set a value to or get a value from an object. As with properties, the keywords get and set are used when defining an indexer. Unlike properties, however, you are not retrieving a particular data member. Instead, you are retrieving a value from the object itself. In addition, instead of creating a name as you do with properties, the this keyword is used to refer to the object instance and thus the object name itself is used later in the code. The general format for defining an indexer is as follows:

```
public dataType this[ int index ]
{
   get
   {
      // do whatever you want
      return aValue; //that is of same type as dataType
   }
   set
   {
      // do whatever you want with a value of dataType but
      // in general you should set a value within the class
      // based on the index and the value they assign.
   }
}
```

1.6.24 Overloading Methods, Constructors and Operators

Overloading is an important feature of object oriented programming because it provides alternative ways to perform the same kind of task. In addition, overloading is also an excellent example of *polymorphism* because of its ability to act appropriately depending on context.

Methods, constructors, and operators are all uniquely identified and characterized by their signatures. The signature of a method, for example, consists of the name of the method along with the type and kind (value, reference or output) of each of its parameters. A method is said to be overloaded when multiple methods in the same class have the same name but different signatures. Since constructors are special methods within a class, they too can be overloaded just like you can overload methods. In addition, most operators can also be overloaded so that they are then able to perform customized functions on objects created from the class that defines them. However, you should overload only those operators that make sense for the class.

Complex number operations provide excellent examples of all aspects of overloading and will be discussed in greater detail later in this book. For now, overloading methods can be illustrated in the simple example shown below where two methods have the same identifier names but contain different signatures:

```
public int Add(int a, int b)
{
    int sum = a + b;
    return sum;
}

public string Add(string s1, string s2)
{
    string msg = s1 + s2;
    return msg;
}
```

Note that although both of these methods are called Add, they are used in different context. The first method adds two integers whereas the second method adds (*i.e.* concatenates) two strings by overloading the addition operator `+`.

1.6.25 Delegates

A *delegate* is an object that can hold a reference (*i.e.* memory address) to a method. Methods that are referenced by a delegate may be either an instance method associated with an object or a static method associated with a class and, in addition, may also originate from a `struct`. All that is required is that the return type and signature of the method matches that of the delegate. However, before a method can be called through a delegate, the delegate must first be declared and then properly referred to the desired method. Consequently, the same delegate can be used to dynamically invoke different methods at runtime by simply changing the memory address of the method to which the delegate refers. In addition, delegates also support *multicasting* which is the ability to create what is called an invocation list of different methods that will be automatically called and processed in its entirety whenever the corresponding delegate is invoked. Multicasting is accomplished by first instantiating a delegate and then using the `+` or the `+=` operator to add methods to its invocation list as the examples below will illustrate. To remove a method from the invocation list, the `-` or the `-=` operators may be used instead.

In essence, delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. In all fairness, a delegate can be used to call a method even though the method can be called directly. However, delegates are very useful because sometimes you don't always know in advance which methods might be called into action at runtime. By granting your delegates the authority to run the correct method(s) for you, one delegate can be used to call several different methods. This is why delegates work so well in event-driven programs, where you don't always know in advance which event will occur first.

When you define a delegate type, you identify what type of method the delegate represents. To associate a delegate with a method, a delegate instance is defined using the method name as the argument inside the parenthesis. The constructor for a delegate always takes just one parameter because you are sending the name of one method for the constructor to reference. When a delegate is used, it passes a method, instead of data, as an argument. The general syntax for declaring a delegate is as follows:

```
[modifier] delegate returnType DelegateName([parameter]);
```

There are three steps in defining and using delegates: *Declaration*, *Instantiation* and *Invocation*. The following example illustrates these steps along with different ways to work with delegate objects.

```
namespace DelegateExample
{
   // Declaration
   public delegate void myDelegate(string s);

   class myClass
   {
      public void myMethod1(string s)
      {
        Console.WriteLine(s + " from instance method1\n");
      }

      public static void myMethod2(string s)
      {
        Console.WriteLine(s + " from static method2\n");
      }
   }

   class Program
   {
      static void Main(string[] args)
      {
        //Example illustrating different ways to work with
        //delegate objects

        //(1) Working with instance methods.

        //Instantiate a null delegate object called delObj1.
        myDelegate delObj1 = null; //Delegate Instantiation

        //Instantiate an object called myObj from myClass
        //class.
        myClass myObj = new myClass();

        //Assign instance method myMethod1 to delegate
        //object delObj1.
        delObj1 = myObj.myMethod1;

        //Use the delegate object delObj1 to pass some data
        //to myMethod1.
        delObj1("Hello");  //Invocation
```

```
//(2) Working with static methods.

//Instantiate a null delegate object called delObj2.
myDelegate delObj2 = null; //Delegate Instantiation

//Assign static method myMethod2 to delegate object
//delObj2. Note that because myMethod2 is a static
//method, there is no need to first instantiate an
//object from myClass in order to then assign
//myMethod2 to delegage object delObj2.
delObj2 = myClass.myMethod2;

//Use the delegate object delObj2 to pass some
//data to myMethod2
delObj2("Greetings");  //Invocation

//(3) Working with instance and static methods to
//build an invocation list

//Create two delegate objects and directly assign
//them to their respective instance and static methods.

//Delegate instantiation
myDelegate delObj3=new myDelegate(myObj.myMethod1);
//and invocation
delObj3("Building invocation list, element 1");

//Delegate instantiation
myDelegate delObj4=new myDelegate(myClass.myMethod2);
//and invocation
delObj4("Building invocation list, element 2");

//Build an invocation list from these two
//delegate objects
myDelegate delObjTotal = delObj3 + delObj4;

//Process the invocation list with some data
delObjTotal("Processing entire invocation list.");

//Remove a method from invocation list
delObjTotal -= delObj4;

//Process remaining methods in invocation list
delObjTotal("Last message is");

//Alternate way to build invocation list
myDelegate delObj5 = null;
delObj5 += new myDelegate(myObj.myMethod1);
delObj5 += new myDelegate(myClass.myMethod2);

//Process total invocation list with some data
delObj5("bye...");

//Remove a method from invocation list
delObj5 -= myClass.myMethod2;
```

```
            //Process remaining methods in invocation list
            delObj5("Final message is");

            //Pause until user hits enter key
            Console.ReadLine();
        }
    }
}
```

1.6.26 Events

Events are mechanisms that cause specific code to execute when some particular action occurs in an application. One of the most common ways for events to be raised in C# is from user interaction with control objects in a Windows Form. For example, the simple action of clicking a button with the mouse can raise an event to notify a Windows Form that one of its buttons was clicked. An underlying *event handler* then responds to this event by performing some specific task. Objects can also raise their own events. For example, the Timer object can be configured to raise its own Timer event after a certain amount of time has elapsed. Events may also be raised by the Windows operating system while in the process of running an application. For example, whenever a section of an existing window gets obscured by another window, the Windows operating system will raise an event. Then when the area of the previously obscured window gets re-exposed, another event is raised to notify Windows to repaint that particular area of the console screen. Finally, programmers may also write their own custom events which may be raised directly from within the C# application itself and be applied to a variety of purposes.

Event functionality in the .NET Framework is provided by three interrelated elements: a class that raises the event, an event delegate that connects the event with its handler, and a class that captures the event and responds to it. The class that raises the event is called the *publisher* or *sender*. Classes that capture the event and respond to it are called the *subscribers* or *receivers*. In other words, the publisher determines when an event is raised and the subscribers determine what action is taken in response to the event. An event can have multiple subscribers and a subscriber can handle multiple events from multiple publishers. Events that have no subscribers are never raised.

When an event is raised, the publisher of an event does not know in advance which subscriber will handle the event and so it is the responsibility of subscribers to register or unregister themselves with the publisher of an event. An intermediary mechanism is therefore needed to interact between the code that raises the event in the publisher and the code that executes a response to the event in the subscriber. Because of how they work, delegates provide the most ideally suited mechanism for this kind of desired functionality. A delegate is a class that can hold a reference to a method. Unlike other classes, a delegate class has a signature, and it can hold references only to methods that match its signature. Although delegates have other uses, the material contained in this section will primarily focus on their role in programming event driven applications.

To sum up, events therefore essentially work like this: any subscriber that has an interest in a particular event registers a special method, called an event handler, with the underlying delegate of the corresponding event publisher. Event handlers contain code specifying what actions to take when the event occurs. By registering for a particular event, the event handler simply gets added to the invocation list of the associated event delegate. Later, when the event is raised, the associated event delegate is invoked and sequentially calls all the methods and their associated event handlers that were previously added to its invocation list. Those readers who are familiar with design patterns in object-oriented programming may notice that event coding in C# follow the *observer* design pattern, also known as the *publisher-subscriber* pattern [10]. This pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Successful implementation of a custom event in C# is a multi-step process that requires some careful planning and a considerable amount of attention to detail. However, by following a prescribed list of steps and well established guidelines as described in greater detail below, implementing custom events in C# can actually be a very straight forward pleasant endeavor instead of some harrowing or daunting ordeal.

Step 1: EventArgs - Derive a class from `System.EventArgs` *to hold the event-related data.*

In order to follow event publishing standards, any data that is sent by the publisher of an event to all of its subscribers during the raising of that event is encapsulated as properties of a derived class that is traditionally named `EventNameEventArgs` since it inherits from the `System.EventArgs` base class. Actually, the `EventNameEventArgs` name can be any legal identifier you want. However, using an identifier that begins with your own event name and ends with `EventArgs` improves code readability and makes it easier to remember the purpose for creating this class. Presumably this data is also relevant to the occurrence of the event and subsequent event handling methods can read these event arguments in order to learn more details about the event.

For completeness it should be pointed out that using the `System.EventArgs` class or any subclass thereof, is not a strict technical requirement. Instead, the event delegate signature could very well just specify each parameter type and name. The problem with this approach, however, is that such a signature ties the event publisher with all of its subscribers and if you should ever need to modify any of these parameters in the future, then all the subscribers would have to be modified as well. Consequently, it is highly recommended to encapsulate all event data into a single derived class of the `System.EventArgs` base class since doing so significantly reduces the amount of additional work needed to make any future changes to the data values that are passed to the event subscribers.

The general format of the derived class, `EventNameEventArgs` is as follows:

```
public class EventNameEventArgs : System.EventArgs
{
  //Provide code to handle data for the event arguments.
}
```

Step 2: Event Handler Define a delegate type that specifies the signature of the event handler.

For each `EventArgs` subclass you created in Step 1 you must also create a matching delegate event handler. You can create your own delegate event handler or use one of the `System.EventHandler` delegates provided by the .NET Framework. If you use the default `System.EventArgs` class instead of a subclass of it, there is no need to declare your own delegate since you can then use the corresponding `System.EventHandler` delegate. On the other hand, if you use a derived `EventNameEventArgs` subclass then you need to create your own delegate and its declaration supplies the signature of the delegate event handler. As before with `EventNameEventArgs`, your delegate identifier can have any name you want. However, using a delegate identifier that begins with the event name prefix `EventName` and ends with the suffix `EventHandler` improves code readability and also makes it easier to remember the purpose of creating this delegate. In addition, all delegate event handlers must be of type void in order for them to also be suitable for multicasting so that they can then hold references to more than one event handling method. Delegates can therefore be thought of as an event dispatcher for the class that raises the event by maintaining a list of registered event handlers for the event. The general format of a delegate event handler declaration is:

```
public delegate void EventNameEventHandler(object sender,
    EventNameEventArgs e);
```

It should be noted that by convention event delegates in the .NET Framework have two parameters, the source that raised the event [object sender] and the data for the event `[EventNameEventArgs e]`. An instance of the `EventNameEventHandler` delegate can now bind to any method that matches its signature.

Step 3: Define an event based on the delegate type declared in Step 2.

The event keyword is used to formally declare each event. There are two valid event declaration syntax alternatives: the *field-like* syntax and the *property-like* syntax. The field-like syntax is more commonly used in custom event implementations where the number of events is small. The property-like syntax is recommended for use in situations that involve a large number of events where perhaps only a few of which are expected to be subscribed to at any given time. Using the property-like syntax enables you to exert greater control over the registration and unregistration of subscribers with their event handler delegates. In addition, the property-like syntax follows good object-oriented coding practices by allowing one to encapsulate events just like one can encapsulate private class members. However, the encapsulation of events is not a critical requirement and so creating events using the field-syntax is both acceptable and easier to use.

Using the field-like syntax, the general format for declaring an event, `EventName`, based on the delegate declared in Step 2, `EventNameEventHandler`, is as follows:

```
public event EventNameEventHandler EventName;
```

Here, the delegate for the event `EventName` is `EventNameEventHandler`, as declared and specified earlier in Step 2. Using the property-like syntax, the general format for declaring an event, named `EventName`, based on the delegate declared in Step 2: `EventNameEventHandler` and using the accessors `add` and `remove`, is as follows:

```
private event EventNameEventHandler privateEventName;
public event EventNameEventHandler EventName
{
  add
  {
    //May add extra code here.
    privateEventName += value;
  }
  remove
  {
    //May add extra code here.
    privateEventName -= value;
  }
}
```

The property-like syntax appears very similar to a typical property declaration with the exception that the set of `get` and `set` blocks have been replaced with a set of `add` and `remove` blocks. Instead of retrieving or setting the value of a private member variable, the `add` and `remove` blocks add and remove incoming delegate instances to or from the underlying event handler.

With either the field-like or the property-like syntax, the `EventName` is the name of the event being declared and `EventNameEventHandler` is the name of the delegate that was created for this event. Together, this line of code uses the event keyword to create an event instance named `EventName` that is a delegate of type `EventNameEventHandler`.

Step 4: Create a protected virtual method in the publisher class that raises the event declared in Step 3.

For each non static event in unsealed classes, the publisher class should include a protected virtual method that is responsible for raising the event. After an event has been defined, as it was done in Step 3, the publisher needs to *raise* the event. Raising the event is generally a two step process. The first step is to check for the existence of any subscribers. The second step is to raise the event only if there is at least one subscriber in the invocation list of the event delegate. If there are no subscribers, then the delegate will test to null. Since any unhandled exceptions raised in event handling methods found in subscribers will be propagated back to the event publisher, the raising of events should be attempted only within the context of a `try/catch` block. Prior to raising the event, you will need to have an instance of your `EventNameEventArgs` subclass populated with event-specific data. If the event contains or passes no data, then you should assign the event-related data to be `EventArgs.Empty`.

```
protected virtual void OnEventName()
{
  try
  {
    if (EventName != null)
    {
      EventNameEventArgs e = new EventNameEventArgs();
      EventName(this, e);
    }
  }
  catch
  {
    // Handle exceptions here
  }
}
```

Step 5: Define a public method in the publisher class to run the protected, virtual method declared in Step 4 when the event occurs.

One needs a method, such as `OnEventName()` discussed in step 4, that will be called by the publisher when it raises the event. However, since this method is encapsulated with a protected modifier in order to comply with good object-oriented programming practices, one needs a way to publicly interact with this protected method. The simplest way to do this is to have a publicly accessible method that calls on the protected `OnEventName()` method such as:

```
public void RunOnEvent(()
{ OnEventName(); }
```

Step 6: In each subscriber class implement the corresponding event handler method with the same signature as the publishers delegate defined in Step 2.

In order to subscribe to an event, one needs to build a subscriber class that is distinct and independent from the publisher class. In each subscriber class that wants to subscribe to an event in the publisher class, create an event handler method with the same signature as the publishers event delegate. The general format for this event handler method is like this:

```
public void EventNameEventHandler(object sender,EventNameEventArgs e)
{
  //Add your code to process an event for a particular
  //subscriber here.
}
```

Step 7: Instantiate the publisher object. For non-static event handlers in subscriber classes, also instantiate a subscriber object.

In order to subscribe to an event, the subscriber needs a reference to the object publishing the event of interest. Therefore, the object publishing the event of interest needs to be instantiated from the publisher class. Likewise, the instance event handlers in the subscriber classes need to be referenced by the event delegate in the publishing class. Therefore any object subscribing to the event of interest needs to be instantiated from the subscriber class. For subscriber classes containing static event handler methods, there is no need to first instantiate the subscriber object since you can call the event method handler directly. The general format of carrying out this step consists of introducing code that will look something like this:

```
publisherClass publisherObject = new publisherClass ();
```

for the publisher class and

```
subscriberClass subscriberObject = new subscriberClass();
```

for each subscriber class.

Step 8: Create a delegate object to the event and attach the event handler method to the event.

Create an instance of the event handler of interest, passing the name of the event handling method. Using the `new` keyword, instantiate the delegate in the same line in which the delegate is attached to the event. The subscriber then registers its event handler (delegate) instance with the publisher, like this:

```
publisherObject.EventName +=
   new EventNameEventHandler(subscriberObject.
        subscriberEventHandlerMethod);
```

or like this:

```
publisherObject.EventName += subscriberObject.
    subscriberEventHandlerMethod;
```

where

- `publisherObject` is the reference to the object that will raise the event of interest.
- The `+=` operator is used to add the delegate instance to the invocation list of the event handler in the publisher. Remember, multiple subscribers may register with the event. Use the `+=` operator to append the current subscriber to the underlying delegate's invocation list.
- `EventNameEventHandler` is a reference to the particular event handler delegate.

- Finally a call to `subscriberObject.subscriberEventHandlerMethod` supplies the name of the method in the subscribing class that is to be called upon the raising of the event.

Step 9: Unregister the subscriber event handling method from the event. (Optional)

When the subscriber no longer receives event notifications from the publisher, you can unregister the subscriber from the event. However, this step is optional since subscribers are automatically unregistered from publishers when the subscriber is disposed. The general format to unregister an event is just like the ones shown in step 8 but using `-=` instead of `+=` .

```
publisherObject.EventName -= subscriberObject.
    subscriberEventHandlerMethod;
```

A very simple code example illustrating how to implement events in C# following these suggested guidelines is given below.

```
namespace NewEventExample
{
 class Program
 {
  //STEP 1:
  //Derive a class from EventArgs to hold event related data
  public class myEventArgs : EventArgs
  {
    //Create some data to pass around in the event arguments
    //eventCounter counts the number of events called.
    public int eventCounter;

    //eventHandlerCounter counts the number of event
    //handlers called.
    public static int eventHandlerCounter = 0;
  }


 //STEP 2:
 //Define a delegate type that specifies the signature of
 //the event handler.
 public delegate void myEventHandler(object senderObj,myEventArgs e);

  //Create an event publisher class
  public class myPublisher
  {
    //Create a static event counter
    static int count = 0;

    //STEP 3:
    //Define an event based on the delegate type
    //"myEventHandler" declared in step 2
    //(field-like version)
    public event myEventHandler myEvent;
    //Define an event based on the delegate type
    //"myEventHandler" declared in step 2
    //(property-like version)
    /*
    private event myEventHandler myPrivateEvent;
    public event myEventHandler myEvent
    {
      add { myPrivateEvent += value; }
      remove { myPrivateEvent -= value; }
    }
    */

    //STEP 4:
    //Define a protected virtual method that raises
    //the event declared in step 3.
    protected virtual void OnEvent()
    {
      //Before raising the event, make sure there is
      //at least one registered subscriber for the event
      try
      {
        if (myEvent != null)
        {
          //Populate the derived argument class with event
          //specific data.
          myEventArgs arg = new myEventArgs();

          //Update the event counter and
          arg.eventCounter = ++count;
          Console.WriteLine("EVENT: {0}", arg.eventCounter);

          //raise the event by calling the associated
          //delegate to multicast every registered
          //subscriber event handler method in its
          //invocation list.
          myEvent(this, arg);
        }
      }
      catch
      {
        //Handle exceptions here
      }
    }

    //STEP 5:
    //Define a public method to run the protected, virtual
    //method declared in Step 4 for when the event occurs.
    public void RunOnEvent()
    {
      OnEvent();
    }
  }

  //SubscriberA contains an instance event handler as well
  //as a way to internally register and unregister for the
  //event along with an event handler.
```

```
public class SubscriberA
{
  private myPublisher currPub;

  public SubscriberA(myPublisher p)
  {
    currPub = p;
  }

  //STEP 6:
  //In each subscriber class, create its own event
  //handling method with the same signature as the
  //publisher’s delegate.
  public void SubscriberA_EventHandlerA(object senderObj,
  myEventArgs e)
  {
    Console.WriteLine("Processing event handler in
    SubscriberA (instance method)");
  }

  //STEP 8:
  //Register the event
  public void register()
  {
    currPub.myEvent += this.SubscriberA_EventHandlerA;

    //Update event handler counter and display message on
    //the screen.
    myEventArgs.eventHandlerCounter++;
    Console.WriteLine("\nRegistering SubscriberA. Event
    handlers registered:{0}",myEventArgs.eventHandlerCounter);
  }

  //STEP 9:
  //Unregister the event
  public void unregister()
  {
    currPub.myEvent -= this.SubscriberA_EventHandlerA;

    //Update event handler counter and display message on
    //the screen.
    myEventArgs.eventHandlerCounter--;
    Console.WriteLine("\nUnregistering SubscriberA. Event
    handlers registered:{0}",myEventArgs.eventHandlerCounter);
  }
}

//SubscriberB contains an instance event handler.
//Registering and unregistering for the event must be
//done externally.
public class SubscriberB
{
  //STEP 6:
  //In each subscriber class, create its own event
  //handling method with the same signature as the
  //publisher’s delegate.
```

```
    public void SubscriberB_EventHandlerB(object senderObj,
    myEventArgs e)
    {
      Console.WriteLine("Processing event handler in
      SubscriberB (instance method)");
    }
  }

  //SubscriberC contains a static handler for the event.
  public class SubscriberC
  {
    //STEP 6:
    //In each subscriber class, create its own event
    //handling method with the same signature as the
    //publisher's delegate.
    public static void SubscriberC_EventHandlerC(object
    senderObj, myEventArgs e)
    {
      Console.WriteLine("Processing event handler in
      SubscriberC (static method)");
    }
  }

  public class MyClass
  {
    public static void Main()
    {
      //STEP 7:
      //Instantiate the publisher object
      myPublisher myPublisherObj = new myPublisher();

      //STEP 7:
      //Instantiate a subscriber object named subA
      SubscriberA subA = new SubscriberA(myPublisherObj);

      //STEP 7:
      //Instantiate another subscriber object named subB
      SubscriberB subB = new SubscriberB();

      //STEP 8:
      //Register subscriber object subA for the event
      subA.register();

      //STEP 8:
      //Register subscriber object subB for the event either
      //this way:
      myPublisherObj.myEvent += subB.SubscriberB_EventHandlerB;
      //or this way:
      //myPublisherObj.myEvent += new
      myEventHandler(subB.SubscriberB_EventHandlerB);

      //Update event handler counter and display message on screen.
      myEventArgs.eventHandlerCounter++;
      Console.WriteLine("Registering SubscriberB. Event
      handlers registered: {0}", myEventArgs.eventHandlerCounter);
      //STEP 8:
      //Register event handler for subscriber class SubscriberC.
      myPublisherObj.myEvent +=
      SubscriberC.SubscriberC_EventHandlerC;

      //Update event handler counter and display message on screen.
      myEventArgs.eventHandlerCounter++;
      Console.WriteLine("Registering SubscriberC. Event
      handlers registered: {0}", myEventArgs.eventHandlerCounter);

      //Raise an event for all the currently registered
      //event handlers
      Console.WriteLine("\nRaise event for all {0} currently
      registered subscribers.\n", myEventArgs.eventHandlerCounter);
      myPublisherObj.RunOnEvent();

      //STEP 9:
      //Unregister event handler for subscriber object subA
      subA.unregister();

      //STEP 9:
      //Unregister event handler for SubscriberB object subB
      myPublisherObj.myEvent -= subB.SubscriberB_EventHandlerB;

      //Update event handler counter and display message on screen.
      myEventArgs.eventHandlerCounter--;
      Console.WriteLine("Unregistering SubscriberB. Event handlers
      registered: {0}", myEventArgs.eventHandlerCounter);

      //Raise an event for all currently registered event handlers
      Console.WriteLine("\nRaise event for all {0} currently
      registered subscribers.\n", myEventArgs.eventHandlerCounter);
      myPublisherObj.RunOnEvent();
    }
  }
}
```

The resulting output of the code given above is as follows:

```
Registering SubscriberA. Event handlers registered: 1
Registering SubscriberB. Event handlers registered: 2
Registering SubscriberC. Event handlers registered: 3

Raise event for all 3 currently registered subscribers.

EVENT: 1
Processing event handler in SubscriberA (instance method)
Processing event handler in SubscriberB (instance method)
Processing event handler in SubscriberC (static method)

Unregistering SubscriberA. Event handlers registered: 2
Unregistering SubscriberB. Event handlers registered: 1

Raise event for all 1 currently registered subscribers.

EVENT: 2
Processing event handler in SubscriberC (static method)
```

1.6.27 Collections

A collection is a set of similarly typed objects that are grouped together. Often, closely related data can be handled more efficiently when grouped together into a collection. Instead of writing separate code to handle each individual object, you can use the same code to process all the elements of a collection at once. The .NET Framework provides specialized collection classes for data storage and retrieval which are defined as part of the `System.Collections` and the `System.Collections.Generics` namespace. The latter contains contains interfaces and classes that define generic collections which allow users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections such as those found in the `System.Collections`. With typed collections you specify the data type to be used in the collection that you instantiate and the resulting collection objects can then only store elements of the specified data type. By contrast, untyped collections allow you to create collection objects without specifying the data type of its elements. There are at least two drawbacks to this approach. First, sloppy management of data types in the elements of a collection can sometimes lead to runtime errors. Second, handling elements in untyped collections may require a lot of type casting which can slow down program performance.

The `System.Collections` classes can usually be categorized into three types:

- Commonly used collections come in both generic and non-generic versions and consist of various well known data structures such as hash tables, queues, stacks, dictionaries and lists.
- Bit collections are collections whose elements are bit flags. This topic will be discussed later in this book in another chapter dedicated solely to bit manipulation.
- Specialized collections which are collections for highly specific purposes.

Most collection classes derive from the following interfaces:

ICollection	IComparer	IDictionary
IEnumerable	IList	IDictionaryEnumerator

and their generic equivalents. In addition, some collection classes have sorting capabilities and most are indexed. All collections that directly or indirectly implement either the `ICollection` interface or the `ICollection` generic interface share several useful features in addition to methods that add, remove, or search elements. For example, memory management is carried out automatically so that a collection can dynamically expand in size as needed. This feature is very useful when the programmer does not know in advance how much memory space to set aside for data allocation at the start of an application. Collection classes can also generate their own enumerator which makes it easy to iterate through their elements. The enumerator is an object that iterates through its associated collection. The lower bound of a collection is the index of its first element. All indexed collections in the `System.Collections` namespace have a lower bound of zero. The capacity of a collection is the number of elements it can contain. The count of a collection is the number of elements it actually contains.

Objects of any type can be grouped into a single collection of the type Object to take advantage of constructs that are inherent in the language. However, in a collection of type Object, additional processing is done on the elements individually, such as boxing and unboxing or conversions, which affect the performance of the collection. Boxing and unboxing typically occur if storing or retrieving a value type in a collection of type Object.

Choosing the best or most efficient collection class to use in the course of developing an application is a very important decision that most programmers need to make since each collection class has its own functionality and its own limitations. Fortunately, there are some useful well established guidelines that are very helpful in making such decisions. These guidelines are briefly summarized below.

Problem:
You need a sequential list where the element is typically discarded after its value is retrieved.

Suggested Solution:

- Use the Queue class or the `Queue(T)` generic class if you need first-in-first-out (FIFO) behavior.
- Use the Stack class or the `Stack(T)` generic class if you need last-in-first-out (LIFO) behavior.

Problem: You need to access the elements in a certain order, such as FIFO, LIFO or random.

Suggested Solution:

- The Queue class and the `Queue(T)` generic class offer FIFO access.
- The Stack class and the `Stack(T)` generic class offer LIFO access.
- The `LinkedList(T)` generic class allows sequential access either from the head to the tail or from the tail to the head.
- The rest of the collections offer random access.

Problem: You need to access each element by index.

Suggested Solution:

- The `ArrayList` and `tringCollection` classes and the `List(T)` generic class offer access to their elements by the zero-based index of the element.

- The `Hashtable`, `SortedList`, `isDictionary`, and `stringDictionary` classes, along with both the `Dictionary(TKey,TValue)` and `SortedDictionary(TKey ,TValue)` generic classes offer access to their elements by the key of the element.

- The `NameObjectCollectionBase` and `NameValueCollection` classes, as well as the `KeyedCollection(TKey,TItem)` and `SortedList(TKey,TValue)` generic classes offer access to their elements by either the zero-based index or the key of the element.

Problem: You need each element to contain one value, a combination of one key and one value, or a combination of one key and multiple values.

Suggested Solution:

- One value: Use any of the collections based on the `List` interface or the `IList (T)` generic interface.

- One key and one value: Use any of the collections based on the `IDictionary` interface or the `IDictionary(TKey,TValue)` generic interface.

- One value with embedded key: Use the `KeyedCollection(TKey,TItem)` generic class.

- One key and multiple values: Use the `ameValueCollection` class.

Problem: You need to sort the elements differently from how they were entered.

Suggested Solution:

- The `Hashtable` class sorts its elements by their hash codes.

- The `SortedList` class, the `ortedDictionary(TKey,TValue)` and `SortedList (TKey,TValue)` generic classes sort their elements by the key, based on implementations of the `IComparer` interface and the `IComparer(T)` generic interface.

- `ArrayList` provides a Sort method that takes an `IComparer` implementation as a parameter. Its generic counterpart, the `List(T)` generic class, provides a Sort method that takes an implementation of the `IComparer(T)` generic interface as a parameter.

Problem: You need fast searches and retrieval of information.

Suggested Solution:

- The `ListDictionary` is faster than `Hastable` for small collections.

- The `Dictionary(TKey,TValue)` generic class provide faster lookup than the `SortedDictionary(TKey,TValue)` generic class.

Problem: You need collections that accept only strings.

Suggested Solution:

- The `StringCollection` and `StringDictionary` classes are in the `System.Collections.Specialized` namespace. In addition, you can use any of the generic collection classes in the `System.Collections.Generic` namespace as strongly typed string collections by specifying the String class for their generic type arguments.

Additional and more detailed information regarding the coding and implementation of the Collections and the `Collections.Generics` classes can be found in [11] and [12] .

1.6.28 File Input/Output

Computer programs written for scientific and engineering applications often require their output data to be written to a file for later processing or perhaps for plotting results on a graph. Sometimes scientific and engineering applications also need input data to be read from a file. Whatever the case may be, the `System.IO` namespace provides several useful classes for managing both binary and text files. The abbreviation `IO` is often used to denote input/output file operations. To handle `IO` file operations, the .NET Framework uses the concept of streams. A stream is just a sequence of bytes. Since a byte is 8 bits and a bit is either a 0 or a 1, a stream can be thought of as just a sequence of zeros and ones arranged in some specific order. Streams involve the following three operations:

- Streams can be read from. Reading is the transfer of data from a stream into a data structure, such as an array of bytes.
- Streams can be written to. Writing is the transfer of data from a data structure into a stream.
- Streams can support seeking. Seeking is the query and modifying of the current position within a stream.

Consequently, to read and write text files you use text input or output streams and to read and write binary files, you use input or output binary streams. In a text file, data is stored as text characters or strings which can be very easily accessed and manipulated by other programs. In binary files, data can be stored in a variety of data types and, as a result, it can sometimes be difficult to access and manipulate unless you have some prior detailed knowledge of how the data was originally written to the file. Accordingly, for the purposes of the numerical applications contained in this book, discussion of `IO` file operations will henceforth be directed only to text files which can be more easily accessed by any computer regardless of any internal data type specifics.

In order to write output to a file we first have to create an output stream of type `FileStream` and attach a `StreamWriter` to it which, like `Console`, provides the methods `Write` and `WriteLine`. Likewise, in order to read input from a file we first have to open a `FileStream` and attach a `StreamReader` to it which, like `Console`, provides the methods `Read` and `ReadLine`. Perhaps the best way to illustrate how all these mechanisms work is through a simple file input/output manipulation example as shown below. First, a public delegate is declared to store references to one or more arbitrary functions of your choice. Since the number of data points is not known in advance and in order to allow greater flexibility in choosing a data type, all data is handled using a generic `List` collection structure instead of an array. The application then simply generates a table of $(x, f(x))$ data values, writes these values to a file and then reads these values from the file and then displays all of them out on the screen again.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace FileIODemo
{
  class Program
  {
    //Delegate to store references to some arbitrary
    //function
    public delegate double Function(double x);

    public class ConstructFunctionTable
    {
      public void MakeXYTable(Function f, double startValue,
                double endValue, double increment,
                ref List<double> xv, ref List<double> yv)
      {
        //Print table headings
        Console.WriteLine("x\tf(x)");
        Console.WriteLine("-------------");

        //Loop from the start to the end values of the
        //(x,f(x)) table incrementing x with your value
        //of choice. Calculate the corresponding function
        //f(x) for each x value. Display results on the
        //screen for reference.
        for (double x=startValue; x<=endValue; x+=increment)
        {
          xv.Add(Convert.ToDouble(x));
          yv.Add(Convert.ToDouble(f(x)));
          Console.WriteLine(String.Format("{0:f}\t{1:f}",
                                                   x,f(x)));
        }
        Console.WriteLine("-------------");
      }

      public void SaveXYTable(string pathNfilename,
                  ref List<double> xv, ref List<double> yv)
      {
        //Create a FileStream object and a StreamWriter object
        //to write to a file specified by the variable
        //pathNfilename.
        FileStream fsWrite = new FileStream(pathNfilename,
                           FileMode.Create, FileAccess.Write);
        StreamWriter sWriter = new StreamWriter(fsWrite);

        //Specify the available data delimeter characters
        char[] fileDelim = { ',', '|', '\t' };

        //Find out how many (x,y) data points you have.
        int xyCount = xv.Count;

        //Loop through all the data points you have writing
        //each of them to the text file referenced to by the
        //StreamWriter object.
        for (int i = 0; i < xyCount; i++)
        {
          sWriter.WriteLine(String.Format("{0:f}" +
             fileDelim[2] + "{1:f}", xv[i], yv[i]));
        }
        //Close the file StreamWriter object.
        sWriter.Close();
      }

      public void ReadXYTable(string pathNfilename,
                   ref List<double> xv, ref List<double> yv)
      {
        //Create a FileStream object and a StreamReader object
        //to read from a file specified by the variable
        //pathNfilename.
        FileStream fsRead = new FileStream(pathNfilename,
                              FileMode.Open, FileAccess.Read);
        StreamReader sReader = new StreamReader(fsRead);

        //Specify the available data delimeter characters
        char[] fileDelim = { ',', '|', '\t' };

        //Declare a string array to hold incoming data read
        //from file.
        string[] fields;

        //Read the first line of the file.
        string xyDataline = sReader.ReadLine();

        //If the first line of the file is not null, then
        //process data otherwise close the file StreamReader
        //object.
        while (xyDataline != null)
        {
          //Using the delimeter of choice, split the incoming
          //data into two fields: one for the x values and the
          //other for the y values.
          fields = xyDataline.Split(fileDelim[2]);
```

```
      //Store each value in its corresponding array.
      xv.Add(Convert.ToDouble(fields[0]));
      yv.Add(Convert.ToDouble(fields[1]));

      //Read another line of data from the file.
      xyDataline = sReader.ReadLine();
    }

    //Close the file StreamReader object.
    sReader.Close();
  }

  static void Main(string[] args)
  {
    //Give a name to some arbitrary text file to hold the
    //(x,f(x)) data
    string fileName = "TestData.txt";

    //and assign it to the path of the current default
    //directory
    string filePathNName =
           Directory.GetCurrentDirectory().ToString()
           + "\\" + fileName;
    //Note: The filename and path chosen here is arbitrary
    //and these were selected only to illustrate the
    //functionality of this example.

    //Declare a couple of generic collections of doubles.
    //Collections were chosen instead of arrays because we
    //do not know in advance how many data points we have.
    //Unlike arrays, collections do not have a fixed size
    //but can expand in size dynamically. Generics were
    //chosen to allow a flexible choice of data types.
    List<double> xvalue = new List<double>();
    List<double> yvalue = new List<double>();

    //Create a function table object to run this example.
    ConstructFunctionTable fcntable =
            new ConstructFunctionTable();

    //Create an (x,y=f(x)) table of sines running from 0
    //to 2, incrementing by 0.25 and return result inside two
    //collections xvalue, and yvalue.
    Console.WriteLine("The original (x,y) data table:\n");
    fcntable.MakeXYTable(Math.Sin, 0.0, 2.0, 0.25,
                            ref xvalue, ref yvalue);

    //Write the data contained in collections xvalue and
    //yvalue to the data text file described earlier.
    Console.Write("\nSaving data to file " + fileName +
                 " please wait. ");
    fcntable.SaveXYTable(filePathNName, ref xvalue,
                          ref yvalue);
    Console.WriteLine("Done!");
```

```
        //Clear out the collections xvalue and yvalue to
        //receive fresh data.
        xvalue.Clear();   yvalue.Clear();

        //Read the data contained in collections xvalue and
        //yvalue from the data text file described earlier.
        Console.Write("\nReading data from file " + fileName +
                      " please wait. ");
        fcntable.ReadXYTable(filePathNName, ref xvalue,
                             ref yvalue);
        Console.WriteLine("Done!");

        //Display the data results just read on the screen
        Console.WriteLine("\nData just read from file " +
                          fileName + "\n");
        Console.WriteLine("x\tf(x)");
        Console.WriteLine("-------------");
        int xydataCount = xvalue.Count;
        for (int i = 0; i < xydataCount; i++)
        {
          Console.WriteLine(String.Format("{0:f}\t{1:f}",
                                  xvalue[i], yvalue[i]));
        }

        //Pause until user hits the ENTER key
        Console.Write("\nPlease hit the ENTER key to terminate
                      this program");
        Console.ReadLine();
      }
    }
  }
}
```

The resulting output of the code given above is as follows:

```
The original (x,f(x)) data table:
x       f(x)
------------
0.00    0.00
0.25    0.25
0.50    0.48
0.75    0.68
1.00    0.84
1.25    0.95
1.50    1.00
1.75    0.98
2.00    0.91

Saving data to file TestData.txt please wait. Done!
Reading data from file TestData.txt please wait. Done!
Data just read from file TestData.txt

x       f(x)
------------
0.00    0.00
0.25    0.25
0.50    0.48
```

```
0.75    0.68
1.00    0.84
1.25    0.95
1.50    1.00
1.75    0.98
2.00    0.91
```

1.6.29 Output Reliability, Accuracy and Precision

In any numeric calculation, one should always take into consideration the reliability, accuracy and precision of the numerical methods used, the numerical values involved and, of course, the results that are obtained. The concept of *significant figures*, or *digits*, was developed years ago to formally evaluate the reliability of a numerical value. The significant digits of a number are said to be those that can be used with confidence and correspond to a certain number of digits plus one estimated digit. Accuracy refers to how closely a calculated or measured value agrees with its actual true value. Precision is the degree to which further measurements or calculations yield the same or similar results. Thus, accuracy is the degree of veracity while precision is the degree of reproducibility of either a set of measurements or a set of calculated values. The results of a calculation or a measurement can be accurate and precise, neither accurate nor precise, accurate but not precise, or precise but not accurate. However, a measurement or a computation is said to be valid only if it is both accurate *and* precise. Understandably, reliability, accuracy and precision are particularly important issues to consider when using computers to perform numerical calculations.

At the most rudimentary level, computers are limited to interpreting only two states, 0 or 1, and this feature relates to the fact that the primary logic units that make up digital computers are its on/off electronic components. As a result, computers use the binary number system to store and perform numerical operations. In particular, a 0 or 1 is called a *bit*, a set of 4 bits is called a *nibble* and a set of 8 bits consist of a unit called a *byte*. Another unit called *word* consists of a fixed-sized group of bits whose size depends on the particular computer's internal hardware design. The exact number of bits in a word is known as the *word size* or *word length*. Variables that have been assigned to store numerical values are controlled by the programmer under the options provided by the programming language being used. A variable can be declared as an integer, in which case the binary point is fixed at the end of the word length, or as a real, in which case the binary point is said to float across the word length. The floating-point representation allows one to describe a broad range of real numbers. For example, single precision, also known as floating-point numbers, employ 32 bit (4 bytes) word lengths and are therefore capable of storing values ranging anywhere from about $-3.4\text{x}10^{38}$ to $3.4\text{x}10^{38}$. On the other hand, doubles, also known as double precision numbers, employ 64 bit (8 bytes) word lengths and are therefore capable of storing values anywhere from about $-1.80\text{x}10^{308}$ to $1.8\text{x}10^{308}$. Decimals take up 128 bits (16 bytes) and are therefore able to store values ranging from about $-7.9\text{x}10^{28}$ to $7.9\text{x}10^{28}$. The actual range of allowed values for variables of each numerical data type has already been displayed in Table 1.4 earlier in this chapter and can all be actually directly computed using the following code:

```
Console.WriteLine("C# DATA TYPES");
Console.WriteLine("\n{0} \nsize = {1} byte,
range: [{2},{3}]", typeof(byte).ToString(),
sizeof(byte), byte.MinValue, byte.MaxValue);

Console.WriteLine("\n{0} \nsize = {1} byte,
range: [{2},{3}]", typeof(sbyte).ToString(),
sizeof(sbyte), sbyte.MinValue, sbyte.MaxValue);

Console.WriteLine("\n{0} \nsize = {1} bytes,
range: [{2},{3}]", typeof(Int16).ToString(),
sizeof(Int16), Int16.MinValue, Int16.MaxValue);

Console.WriteLine("\n{0} \nsize = {1} bytes,
range: [{2},{3}]", typeof(UInt16).ToString(),
sizeof(UInt16), UInt16.MinValue, UInt16.MaxValue);

Console.WriteLine("\n{0} \nsize = {1} bytes,
range: [{2},{3}]", typeof(Int32).ToString(),
sizeof(Int32), Int32.MinValue, Int32.MaxValue);

Console.WriteLine("\n{0} \nsize = {1} bytes,
range: [{2},{3}]", typeof(UInt32).ToString(),
sizeof(UInt32), UInt32.MinValue, UInt32.MaxValue);

Console.WriteLine("\n{0} \nsize = {1} bytes,
range: [{2},{3}]", typeof(Int64).ToString(),
sizeof(Int64), Int64.MinValue, Int64.MaxValue);

Console.WriteLine("\n{0} \nsize = {1} bytes,
range: [{2},{3}]", typeof(UInt64).ToString(),
sizeof(UInt64), UInt64.MinValue, UInt64.MaxValue);

Console.WriteLine("\n{0} \nsize = {1} bytes,
range: [{2:E},{3:E}]", typeof(Single).ToString(),
sizeof(Single), Single.MinValue, Single.MaxValue);

Console.WriteLine("\n{0} \nsize = {1} bytes,
range: [{2:E},{3:E}]",  typeof(Double).ToString(),
sizeof(Double), Double.MinValue, Double.MaxValue);

Console.WriteLine("\n{0} \nsize = {1} bytes,
range: [{2:E},{3:E}]", typeof(Decimal).ToString(),
sizeof(Decimal), Decimal.MinValue, Decimal.MaxValue);

Console.WriteLine("\n{0} \nsize = {1} bytes,
range: [0x{2,4:X4},0x{3,4:X4}]",
typeof(Char).ToString(), sizeof(Char),
(int)Char.MinValue, (int)Char.MaxValue);

Console.WriteLine("\n{0} \nsize = {1} bytes,
range: [{2},{3}]", typeof(Char).ToString(),
sizeof(Char), (int)Char.MinValue,(int)Char.MaxValue);
```

In general, floating-point calculations performed by a computer are particularly prone to three major kinds of problems:

- An operation may be mathematically illegal, such as attempting to divide a number by zero.
- An operation may be legal in principle but is not contextually supported by the current programming format used to actually perform the desired calculation. A good example of this kind of problem occurs when an attempt is made to calculate $\sqrt{-1}$ using the internal method `Math.Sqrt(-1)`, provided by the .NET Framework, which was originally designed to only calculate the square root of positive real numbers.
- An operation may be legal in principle, but the result may be impossible to represent in the specified format because of an overflow or an underflow event. A good example of this kind of problem arises when the numerical output is so large that it is expressed either as `NaN` (*i.e.* not-a-number) or as `infinity`. In either case, the actual numerical output may not be necessarily infinite or a `NaN` but instead may be just too large for the computer to be able to physically represent.

Although the relative errors involved in multiplication and division are often small, there are situations where such errors may also be significant enough to have some impact on the accuracy and precision of the results that are obtained. Other factors, such as trying to access floating-point numbers beyond the range of their allowed values, may lead to unwanted underflow and/or overflow problems for which there are no quick fixes. When such underflow and/or overflow problems occur, floating-point operations may return $\pm\infty$ or `NaN` as a way to indicate that the result obtained was beyond the range of allowed values for variables of that specific data type. Unfortunately, once a series of operations generates a `NaN`, then everything else becomes a `NaN` and so it's better to try and find a way to prevent this problem from happening in the first place.

One approach is to apply some mathematical trick to slightly modify the original formula sufficiently enough in order for this problem to be completely eliminated or at least have its impact be substantially minimized so that the calculation can then proceed forward without too much additional worry or fuss. For example, in the process of calculating some probabilities one often has to evaluate ratios of factorials which can very easily and quickly become quite large. As a result, dividing two very large numbers, such as U/V, in a careless way may create all kinds of overflow and/or underflow problems. Instead, it may be more computationally prudent to first take their individual logarithms, subtract them from each other and then exponentiate the result as shown below:

$$U/V = \exp(\ln U - \ln V)$$

Another important point to keep in mind when doing computer based floating-point arithmetic is how the IEEE specifications for both the 32-bit float and the 64-bit double actually work. Although a float type is 32 bits wide, only 24 of these bits are assigned to the mantissa and the rest to the exponent. Likewise, a double type is 64 bits wide, but only 53 of these bits are assigned to the mantissa while the rest are assigned to the exponent. As a result, at times some precision may be lost when attempting to carry out certain arithmetic operations near the upper or lower regions of the allowed range of floating-point data types.

Because of the physical limitations imposed by hardware, computers cannot correctly store irrational numbers, such as π or $\sqrt{2}$, or non-terminating rational numbers, such as $1/6$, in floating-point format. In addition, the number of digits (or bits) of precision also puts a limit on the number of rational numbers that can be represented exactly. Consequently, computers handle most numerical values only through approximation schemes and, as a result, a certain amount of truncation and/or round-off errors is often inadvertently introduced to some extent into numerical calculations [13].

Truncation errors arise as a result of using approximate instead of exact numerical values. Round-off errors show up when numbers consisting of limited significant figures are used to represent exact numerical values. Unfortunately, even small errors introduced by floating-point calculations can sometimes eventually grow significantly large, particularly when mathematical algorithms are required to repeatedly perform certain arithmetic operations. As a result, programs that require a lot of number crunching can sometimes produce bugs that are very hard to find along with misleading or even erroneous results. Because of these issues, naive use of floating-point arithmetic can lead to many unwanted problems. For example, you can look at a piece of code all day and it will seem completely correct because it would be correct if the numbers in the computer were stored exactly as they are entered. Even worse, you can add lots of `Debug.WriteLine(...)` statements throughout your program to examine both final and intermediate values more closely only to be mislead into thinking that everything is correct when in reality the computer may be interpreting things internally to be completely different. In order to better illustrate how much trouble these seemingly harmless features can cause, consider the following numerical examples consisting of some simple floating-point arithmetic operations.

```
float tenth = 0.1f;
float one = 1f;
float ten = 10f;

//Goal: Compute 1 - 0.1*10
Console.WriteLine("1 - (1/10)*10 = {0}",one-tenth*ten);
//Expected output: 0
//Actual output: 1.490116E-08 on 32-bit machines
//Actual output: 0 on 64-bit machines

//Goal: Compute 1.0 - 0.9
Console.WriteLine("1 - 0.9 = {0}",one-0.9f);
//Expected output: 0.1
//Actual output: 0.0999999999999996 on 32-bit machines
//Actual output: 0.1 on 64-bit machines
```

From just these simple numerical examples, it seems evident that the precision of the numerical output depends on whether you use a 32-bit or a 64-bit machine. However, since most personal computers today are 32-bit machines, great care must be exercised when working with floating point numbers. Using floating point numbers to control a `for` loop, may result in the loop stopping earlier or later than expected. In addition, using an equality or inequality test to stop a `while` loop, may result in the loop stopping earlier or later than expected. To avoid such problems, developers are strongly advised to use integers for equality or inequality testing and to control code structures involving loops.

From the examples we have just seen, one of the most troublesome and hard-to-find bugs seems to appear in code that does equality testing. If you compare two floating point numbers for equality, the binary representation of the numbers may not be exactly equal even though you know, that by the numbers used, they should be. For example, consider the following equality test that uses the numerical data of the example just discussed.

```
if ( one - tenth*ten) == 0.0)
{...} //will always output false
```

Although the computed value for `ten * onetenth - one =` 1.490116×10^{-08} seems close enough to 0.0 for most practical purposes, the computer will always give a false test result when asked to compare whether 1.490116 x 10^{-08} is equal to 0.0. And while there may be situations in which testing for equality using floating-point numbers is useful or even desirable and necessary, developers are strongly encouraged to first double check their algorithms to see if they can be redesigned to avoid any equality testing of floating-point numbers altogether. If that approach is not feasible, then there is a considerable number of alternative options available to resolve or at least lessen the effect of this floating-point problem on numerical calculations. Goldberg [13], for example, wrote an excellent, long and very detailed account of what every programmer should know about floating-point arithmetic, particularly as it applies to IEEE standards. Dawson [14] wrote a much shorter but very excellent article on this topic and offered some valuable practical advice for addressing and overcoming these issues. Although Dawson's article and proposed solution was directed primarily towards C/C++ programmers, Ruegg [15] expanded on Dawson's ideas and also included a C# version of his code for doing floating-point equality testing. For the convenience and benefit of my readers, I will just make a brief outline of the highlights of these excellent articles.

When doing numerical comparisons, there are always two variables of interest: the `result` along with its close companion, the `expectedResult`. Most programmers will agree that writing code to do comparison of float-point numbers like this:

```
float result, expectedResult;
if (result == expectedResult) {...}
```

is considered bad practice that can lead to false and unwanted erroneous results. The most popular solution commonly found in the literature regarding this problem is to introduce a small tolerance factor in order to allow the comparison to take place within a user specified range of values instead of just one specific number:

```
float result, expectedResult, tolerance = ...;
if (Math.Abs(result - expectedResult) < tolerance) {...}
```

where `Math.Abs()` is an internal C# function that calculates the absolute value and is described in more detail in the next chapter.

Dawson [14] points out that there are at least two problems with this approach. First, a calculation of the absolute error `Math.Abs(result-expectedResult)` is not very meaningful. For example, a calculated absolute error of 1.0 tells you very little. If the actual result is 1000.0 then an error of 1.0 is wonderful. However, if the actual result is 1.0 then an error of 0.1 is terrible. Second, since floating numbers have fixed precision in computers, the calculated absolute error may turn out to be too small to be accurately compared with the proposed tolerance factor. For example, consider a calculation that has an expected answer of $10,000$. Because floating point arithmetic is imperfect, your calculated answer may be off by one or two least significant bits. If you are using 4-byte floats and you are off by one in the least significant bit of your result then instead of $10,000$ you'll get $10,000.000977$. The difference between the expected and actual result is therefore given by 0.000977. If you arbitrarily pick a small tolerance factor of, say 0.00001, then doing a numerical comparison will always give a false result even though the numbers are adjacent floats.

That is not to say that absolute error comparisons have no value at all. If the range of the expected result is known, then checking for absolute error is simple and effective. However, one needs to make sure that the absolute error value is larger than the minimum representable difference for the range and type of float that is being used. Because of these two major problems, it is often more informative and useful to calculate the relative instead of the absolute error and to specify the relative error as a percentage. Thus,

```
relativeError = Math.Abs((result-expectedResult)/expectedResult));
```

Sometimes, however, we do not have an expected result but instead just have two numbers that we want to compare and see if they are almost equal. Using the concept of relative error just discussed, one way to implement this calculation in C# would be to write something like this:

```
public static bool AlmostEqualRelative(float x,
                  float y, float maxRelativeError)
{
   if (x == y)
      return true;
   float relativeError=Math.Abs((x-y)/y));
   if (relativeError <= maxRelativeError)
      return true;
   return false;
}
```

where the maxRelativeError parameter specifies what relative error we are willing to tolerate. For example, if we want 99.999% accuracy then we should pass a maxRelativeError of 0.00001.

Unfortunately, the function `AlmostEqualRelative` shown above has some problems that needs to be addressed. If `x` and `y` are both equal to zero then the step with the term `relativeError` given above will calculate 0.0/0.0. Since zero divided by zero is undefined, 0.0/0.0 will give a `NAN` result. In turn, a `NAN` will never return true on a $\leq$ comparison, and so this function will always return false if `x` and `y` are both zero. As a result, the function `AlmostEqualRelative` is not a good reliable choice for doing floating-point comparisons.

Another major source of trouble is that the function `AlmostEqualRelative` always uses the second paramater as the divisor. As a result, the function call

```
AlmostEqualRelative(x,y,tolerance)
```

will very likely not give the same result as function call

```
AlmostEqualRelative(y,x,tolerance).
```

An improved version of the original function `AlmostEqualRelative` would always calculate the relative error by dividing the smaller by the larger number. One way to implement this additional functionality in C# would be to write something like this:

```
public static bool AlmostEqualRelative2(float x,
                   float y, float maxRelativeError)
{
    if (x == Double.NaN || y == Double.NaN)
        return false;     // per IEEE spec
    if (x == y)
        return true;
    float relativeError;
    if (Math.Abs(y) > Math.Abs(x))
        relativeError = Math.Abs((x - y) / y);
    else
        relativeError = Math.Abs((x - y) / x);
    if (relativeError <= maxRelativeError)
        return true;
    return false;
}
```

Unfortunately, this new and supposedly improved comparison function has been found to behave poorly for numbers around zero. For example, the positive number closest to zero and the negative number closest to zero are extremely close to each other, yet this function will correctly calculate that they have a rather large relative error. As a result, one needs to add an additional check for the maximum absolute error in order to correctly account for numbers that are near zero but have opposite signs. Then the new function would return `true` if either the absolute error *or* the relative error were smaller than the input maximum values. One way to implement this additional functionality in C# would be to write something like this:

```
public static bool AlmostEqualRelativeOrAbsolute(float x,
float y, float maxRelativeError, float maxAbsoluteError)
{
    if (x == Double.NaN || y == Double.NaN)
        return false;     // per IEEE spec
    if (Math.Abs(x - y) < maxAbsoluteError)
        return true;
    float relativeError=0.0;
    if (Math.Abs(y) > Math.Abs(x))
```

```
        relativeError = Math.Abs((x - y) / y);
    else
        relativeError = Math.Abs((x - y) / x);
    if (relativeError <= maxRelativeError)
        return true;
    return false;
}
```

However, this newer and supposedly even better function still has some flaws and limitations. Fortunately there is an alternate and far superior technique for comparing floating point numbers [14]. Most hardware implementations today use the IEEE 754 standard [16] in which two ordered floating point numbers remain ordered when interpreted as sign-magnitude integers. Therefore, floating-point numbers can be compared, at least in principle, by casting them to integers and doing an integer comparison. Using this method one can also obtain the next representable floating-point number by converting to an integer and incrementing. However, when using this method for doing comparison of floating-point numbers several technical details including special values such as `infinity` and `NaN`, subnormal numbers, twos-complement integers, handling of $+0$ and -0, have to be considered. In addition, this technique is less portable because it depends on soley IEEE-754 hardware specifications. Nevertheless, it is fast and reliable thus making it perhaps the best method for doing floating-point comparisons available today.

To summarize, Dawson's [14] approach, later expanded by Ruegg [15], was to make floating-point comparisons on the bit level instead of on an actual numerical level. Instead of choosing an arbitrary tolerance factor to make numerical comparisons, their technique centers around comparing values at the bit level which, in addition to being more accurate, also makes good use of the reality of the limitations imposed by hardware.

2

The .NET Framework Math Class Library

2.1 Introduction

The .NET Framework Math Class Library [17] provides constants and static methods for computing common mathematical functions consisting of real input and output values. Therefore, it seems logical to begin this book of numerical methods in C# with a brief review of all the available mathematical routines that are already part of this library and also include some practical examples to better illustrate how these routines may be used in actual applications. Similar methods for mathematical functions capable of handling complex numbers will be discussed in a later chapter.

An important characteristic of the .NET Framework Math Class Library is that all of its classes and data members are static. This means that one cannot instantiate an object variable of the Math Class type. Instead, the members of a static class can only be accessed by directly using the class name itself. In addition, the Math Class Library is sealed and so it cannot be used for inheritance. Additional supplementary material will be introduced as needed in order to both complement and expand on the existing mathematical routines of this library.

2.2 The .NET Framework Math Class - Fields

2.2.1 The `Math.PI` and `Math.E` Fields

The `Math.PI` and `Math.E` fields are declared internally as:

```
public const double PI;
public const double E;
```

and represent the commonly used mathematical constants π and e respectively. The constant $\pi = 3.141592653\ldots$ is the ratio of the circumference of a circle to its diameter and the constant $e = 2.718281828\ldots$ represents the natural logarithmic base. The following code snippet illustrates how these constants may be accessed.

```
Console.WriteLine("PI = {0}", Math.PI);
Console.WriteLine("E  = {0}", Math.E);
```

2.3 The .NET Framework Math Class - Methods

2.3.1 The Minimum and Maximum Methods

The .NET Framework Math Class Library provides two methods for comparing the relative size of numbers. The `Math.Min` method returns the smaller of two numbers whereas the `Math.Max` method returns the larger of two numbers. These methods are declared internally as:

```
public static [type] Min([type] x, [type] y);
public static [type] Max([type] x, [type] y);
```

Both input parameters, `x` and `y`, along with the corresponding output are of the same data type as specified by the label `[type]` which indicates that these methods can both be overloaded with the following data types: `byte, sbyte, int16, int32, int64, decimal, double, single, uint16, uint32` and `uint64`. The following code snippet demonstrates how to use both the `Min` and `Max` methods to return an output and display the smaller or greater of two variables as indicated.

```
public static void MinMax_Example()
{
   double  x = 5.0, y = 10.0;
   Console.WriteLine("The smaller of {0} and {1} is {2}.",
                     x,y,Math.Min(x,y));
   Console.WriteLine("The greater of {0} and {1} is {2}.",
                     x,y,Math.Max(x,y));
}
```

2.3.2 The Power, Exponential and Logarithmic Methods

The general exponential function with a fixed real number base $b > 1$ and a real number power x is the function expressed by the formula $f(x) = b^x$. The number x is called the exponent and the expression b^x is known formally as the exponentiation of b by x or the exponential of x with base b. It is also more commonly expressed as "the xth power of b", "b to the xth power" or "b to the power x". The most commonly used bases are the natural base e and the base 10. If the base equals the Euler number e, then the exponential function is called the *natural* exponential function and is expressed by $f(x) = e^x = exp(x)$.

The inverse of any exponential function, when it is well-defined, is called the logarithmic function with base b and is denoted by $\log_b$. Thus $\log_b b^x = x$. The logarithm of a number to a given positive real number base is the power or exponent to which the base must be raised in order to produce the number. By definition, the logarithm of x to a base b is written as $\log_b(x)$ or, if the base is implicit, as $\log(x)$. Hence, for a number x, a base b and an exponent y, if $x = b^y$ then $y = \log_b(x)$.

The .NET Framework Math Class Library provides a complete set of methods for calculating powers, exponentials and logarithms of real numbers. Similar methods capable of handling complex numbers will be discussed in a later chapter.

The Power method, `Math.Pow(b,x)`, is used to calculate b^x where b and x are both real numbers. Both of these input variables along with the resulting output are of type double. This method is declared internally as:

```
public static double Pow(double b, double x);
```

In particular, when the exponent $x = 1/2 = 0.5$ then $b^x = b^{1/2} = \sqrt{b}$. This special case of the Power method occurs often enough in applications that the .NET Framework Math Class Library provides its own internal method, `Math.Sqrt(x)`, for taking the square root of a real number specified by the input parameter `x`. This method is declared internally as:

```
public static double Sqrt(double x);
```

Both the input parameter, `x`, and the value returned by this method are of type double. If $x \geq 0$, this method returns the positive square root of `x`. If $x < 0$, then this method returns `NaN` without throwing an exception. If `x = NaN` or if `x = PositiveInfinity` then that value is returned instead.

When the base of interest, b, is the natural base e, then the expression e^x is called the exponential function. The method, `Math.Exp(x)`, is used to raise e to a specific power given by the input parameter `x`. This method is declared internally as:

```
public static double Exp(double x);
```

Note that both the input parameter `x` and the resulting output given by `Math.Exp(x)` are of type double. If the input parameter `x` equals `NaN` or `PositiveInfinity` then that value is returned instead without throwing an exception. However, if `x` equals `NegativeInfinity` then 0.0 is returned as expected.

The inverse of the exponential function e^x is the natural logarithm, or logarithm to base e and is commonly written as $\ln(x)$ or $\log(x)$. The method `Math.Log(x)`, is used for calculating the natural base e logarithm of a number specified by the input parameter `x` and is declared internally as:

```
public static double Log(double x);
```

If $x > 0$, then `Math.Log(x)` returns the natural logarithm of `x`. That is, $\ln(x)$ or $\log_e(x)$. If $x = 0$ then `Math.Log(x)` returns `NegativeInfinity`. If $x < 0$ then `Math.Log(x)` returns `NaN`.

The method `Math.Log(x)` can also be overloaded to return the logarithm of a number in another base as specified by the input parameters `x` and `newBase`, respectively. In this case, the method is declared internally as:

```
public static double Log(double x, double newBase);
```

If $x > 0$, and `newBase` ≥ 0 then `Log(x, newBase)` returns the logarithm of `x` in the base `newBase`. That is, $\log_{\text{newBase}} x$. If `newBase` < 0 then `Log(x, newBase)` returns `NaN`. If $x = 0$ then `Math.Log(x, newBase)` returns `NegativeInfinity`. If $x < 0$ then `Math.Log(x, newBase)` returns `NaN`.

Base 10 logarithmic calculations occurs frequently enough in applications that the .NET Framework Math Class Library provides its own internal method for taking the base 10 logarithm of a real number. The method `Math.Log10(x)`, is used for calculating the base 10 logarithm of a number as specified by the input parameter `x`. This method is declared internally as:

```
public static double Log10(double x);
```

If $x > 0$, then `Math.Log10(x)` returns the base 10 logarithm of `x`. That is, $\log_{10}(x)$. If $x = 0$ then `Math.Log10(x)` returns `NegativeInfinity`. If $x < 0$ then the method `Math.Log10(x)` returns `NaN`.

The following code snippet illustrates the use of the power, exponential and logarithmic methods that were just discussed.

```
public static void PowerExpLog_Example()
{
  double b = 2.75, x = 3.25, newBase = 8.0;

  Console.WriteLine("Exp({0}) = {1}",x,Math.Exp(x));
  Console.WriteLine("Log({0}) = {1}\n",Math.Exp(x),
                            Math.Log(Math.Exp(x)));

  Console.WriteLine("Pow({0},{1}) = {2}",b,x,Math.Pow(b, x));
  Console.WriteLine("Log10({0})/Log10({1}) = {2}\n",
   Math.Pow(b,x),b,Math.Log10(Math.Pow(b,x))/Math.Log10(b));

  Console.WriteLine("Log{0}({1}) = {2}",newBase,x,
   Math.Log(x,newBase));
  Console.WriteLine("Pow({0},{1}) = {2}",newBase,
   Math.Log(x,newBase),Math.Pow(newBase,Math.Log(x,newBase)));
}
```

2.3.3 Special Multiplication, Division and Remainder Methods

Sometimes when two 32-bit integers are multiplied together the final product will be larger than the maximum allowed value that the 32-bit integer data type can hold. Although one could go back and switch the variable declarations to that of another data type that would be capable of accepting larger numbers, there may be one or more compelling reasons for the variables being multiplied to retain their original integer data type. As a result, the .NET Framework Math Class Library provides a special method called `Math.BigMul(x,y)` that can be used for calculating the product of two 32-bit integers and producing an output that is then expressed as a 64-bit integer. This method is declared internally as:

```
public static long BigMul(int x, int y);
```

Similarly, the .NET Framework Math Class Library provides a special method, `Math.DivRem`, for calculating the quotient of two integers returning any remainder in an output parameter. The `DivRem` method can be overloaded and used for both 32-bit and 64-bit integers and is declared internally as:

```
public static  int DivRem(int x,int y,out int remainder);
public static long DivRem(long x,long y,out long remainder);
```

Finally, the method, `Math.IEEERemainder(x,y)`, returns only the remainder obtained from the division of two specified numbers of type double. The return value is also of type double. This method is declared internally as:

```
public static double IEEERemainder(double x,double y);
```

where the parameters `x` is the dividend and `y` is the divisor. This method actually computes the expression $x - yQ$ where $Q = x/y$ is rounded to the nearest integer. If Q falls halfway between two integers, then the even integer is returned. If the expression $x - yQ$ is zero, then the value $+0$ is returned if $x > 0$ otherwise the value -0 is returned if $x < 0$. If $y = 0$, then `NaN` is returned.

The following example illustrates the use of `BigMul`, `DivRem` and `IEERemainder` methods:

```
public static void MultDivRem_Example()
{
  int int1 = Int32.MaxValue;
  int int2 = Int32.MaxValue;
  int intResult;
  long longResult;
  double divisor, doubleResult;

  longResult=Math.BigMul(int1,int2);
  Console.WriteLine("{0}*{1}={2}\n",int1,int2,longResult);

  intResult=Math.DivRem(int1,2,out int2);
  Console.WriteLine("{0}/{1}={2}, with a remainder of {3}.",int1,2,
      intResult,int2);

  String str="The IEEE remainder of {0:e}/{1:f} is {2:e}";
  divisor=2.0;
  doubleResult=Math.IEEERemainder(Double.MaxValue,divisor);
  Console.WriteLine(str,Double.MaxValue,divisor,doubleResult);

  divisor=3.0;
  doubleResult=Math.IEEERemainder(Double.MaxValue,divisor);
  Console.WriteLine(str,Double.MaxValue,divisor,doubleResult);
}
```

2.3.4 The Absolute Value Method

For any real number x the absolute value or modulus of x is denoted by $|x|$ and is defined as

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

As can be seen from the above definition, the absolute value of x is always either positive or zero, but never negative. The .NET Framework Math Class Library provides the method `Math.Abs(x)` that returns the absolute value of a number as specified by the input parameter `x`. This method is declared internally as:

```
public static [type] Abs([type] x);
```

The input parameter, `x`, and the return value are of the same data type as given by the unspecified label `[type]` which indicates that this method can be overloaded with the following data types: `decimal, double, int16, int32, int64, sbyte, single`. If the input parameter `x` is equal to `NegativeInfinity` or `PositiveInfinity`, the return value is `PositiveInfinity`. If the input parameter `x` is equal to `NaN`, the return value is `NaN`. The following code snippet illustrates the use of the method `Math.Abs(x)`.

```
double x = -2.0;
Console.WriteLine("Before:{0,-5} After:{1,-5}",x,Math.Abs(x));
```

which will display the following output on the monitor screen:

```
Before: -2.0    After: 2.0
```

2.3.5 The Sign Method

The sign function is a mathematical function that extracts the sign of a real number. To avoid confusion with the trigonometric sine function, this function is often called the signum function after the Latin form of the word "sign". The signum function of a real number x is defined as follows:

$$sng(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ +1 & \text{if } x > 0 \end{cases}$$

The method, `Math.Sign(x)`, returns a value indicating the sign of a number specified by the input parameter `x`. This method is declared internally as:

```
public static int Sign([type] x);
```

The input parameter, `x`, is of a data type given by the label `[type]` which indicates that this method can be overloaded with the following data types: `int16, int32, int64, sbyte, single, decimal, double`. The following code snippet illustrates the use of the `Math.Sign(x)` method:

```
string str = "The sign of {0} is {1}";
double x = 5.0, y = 0.0, z = -5.0;
Console.WriteLine(str, x, Math.Sign(x));
Console.WriteLine(str, y, Math.Sign(y));
Console.WriteLine(str, z, Math.Sign(z));

Output:
The sign of 5 is +1
The sign of 0 is 0
The sign of -5 is -1
```

2.3.6 Angular Units of Measurement

In order to provide a more thorough coverage of both the trigonometric and the hyperbolic functions, it is important to first review the various angular units of measurement that are available for use and, in addition, to also provide routines in C# for converting values back and forth between them.

In elementary plane geometry, an angle is formally defined by two rays that intersect at the same endpoint. The point where the two rays intersect is called the vertex of the angle and the two rays themselves are called the sides of the angle. An arbitrary angle, say θ, is then measured by first drawing a circular arc of length, say s, that is centered at the vertex of the angle such that it intersects both of its sides. Then the length of the arc s is divided by the radius r of the corresponding circle so that the angle $\theta = s/r$. Since they are defined as the ratio of lengths, angles are considered dimensionless. Nevertheless, there are several units used to measure angles, the most common of which are the the *radian* and the *degree*.

The angle subtended at the center of a circle by an arc that is equal in length to the radius of the circle is defined to be one radian. The degree, denoted by a small superscript circle ($^\circ$) is $1/360$ of a full circle. Therefore, one full circle is 360° or 2π radians, and one radian is $180^\circ/\pi$ degrees, or about 57.2958°. The radian is the preferred unit of angular measurement in the metric system and is abbreviated *rad*. However, this symbol is often omitted in the literature because the radian is assumed to be the default unit for angle measurement unless specifically stated otherwise. All the trigonometric methods in the .NET Framework Math Class Library use the radian as their default unit for angle measurement. The mathematical relationship between radians and degrees is:

$$\text{radians} = (\pi/180) * \text{degrees} \qquad \text{and} \qquad \text{degrees} = (180/\pi) * \text{radians}$$

The corresponding conversion routines between degrees and radians is given by:

```
public static double ConvertDegreesToRadians(double degrees)
{
   double radians = (Math.PI / 180.0) * degrees;
   return (radians);
}

public static double ConvertRadiansToDegrees(double radians)
{
   double degrees = (180.0 / Math.PI) * radians;
   return (degrees);
}
```

A sign convention that has been universally adopted in mathematics is that angles are positive if measured anti-clockwise, and negative if measured clockwise, from a given reference line. If no line is specified, the reference line can be assumed to be the x-axis in the Cartesian plane. Fractions of a degree may be written in normal decimal notation, such as 2.523°, or in the *degree-minute-second* unit system. The minute of arc, also known as *MOA*, *arcminute*, or just *minute*, is $1/60$ of a degree and is denoted by a single prime ($'$) or the letter "M". The second of arc, also known as *arcsecond*, or *second* is $1/60$ of a minute of arc or $1/3600$ of a degree. It is denoted by a double prime ($''$) or the letter "S". The following routine illustrates how to convert an angle expressed in the DMS (degree-minute-second) format to its corresponding value in the DD (Degree-Decimal) format.

```
public static double DMStoDD(double DMSDeg, double DMSMin, double
    DMSSec)
{
   double DD = DMSDeg + (DMSMin/60.0) + (DMSSec/3600.0);
   return DD;
}
```

Similarly, the following routine illustrates how to convert an angle expressed in the DD (Degree-Decimal) format to its corresponding value in the DMS (Degree-Minute-Second) format. Two potential outputs are given: one in the degree/minute/second format and another in the traditional format of xx°xx′xx″.

```
public static void DDtoDMS(double DD, out double d,
   out double m, out double s, out string strDMS)
{
   //Extract the degree component
   double deg = Math.Floor(DD);
   DD -= deg;
   DD *= 60;
   //Extract the minute component
   double min = Math.Floor(DD);
   DD -= min;
   DD *= 60;
   //Extract the second component
   double sec = Math.Round(DD);

   d = deg;
   m = min;
   s = sec;

   //Create padding character
   char pad;
   char.TryParse("0", out pad);

   //Create degree/minute/second strings
   string str_deg = deg.ToString();
   string str_min = min.ToString().PadLeft(2, pad);
   string str_sec = sec.ToString().PadLeft(2, pad);

   //Append degree/minute/second strings together
   strDMS = string.Format("{0}\xb0 {1}' {2}\"", str_deg, str_min,
       str_sec);
}
```

The grad is another unit of angular measurement in the metric system. However, it is not widely used. The international standard symbol for this unit today is *gon*. Other symbols used in the past include ”gr”, ”grd”, and ”g”, the latter sometimes written as a superscript such as in 50^g. The grad is a unit of plane angle, equivalent to $1/400$ of a full circle and so one full circle is the equivalent of 400 grads. Therefore, one grad equals $9/10$ of a degree or $\pi/200$ of a radian. The following code snippet contains routines in C# to convert between grads and degrees along with grads and radians.

```
public static double GradsToRadians(double grads)
{
   double radians = (grads / 200.0) * Math.PI;
   return (radians);
}

public static double RadiansToGrads(double radians)
{
   double grads = (radians / Math.PI) * 200.0;
   return (grads);
}

public static double DegreesToGrads(double degrees)
{
   double grads = (degrees / 9.0) * 10.0;
   return (grads);
}

public static double GradsToDegrees(double grads)
{
   double degrees = (grads / 10.0) * 9.0;
   return (degrees);
}
```

2.3.7 The Trigonometric Functions

The .NET Framework Math Class Library provides three methods for calculating the basic three trigonometric functions: $\cos x$, $\sin x$ and $\tan x$. These methods are declared internally as:

```
public static double Cos(double x);
public static double Sin(double x);
public static double Tan(double x);
```

The input parameter `x` must be in radians and is of type double. The output value returned is the result calculated by the particular trigonometric function and is also of type double. Similar methods for doing corresponding calculations using complex numbers will be discussed in a later chapter. The other three remaining trigonometric functions, $\sec x$, $\csc x$ and $\cot x$, can be easily calculated from their standard definitions as shown below:

$$\sec x = \frac{1}{\cos x} \qquad \csc x = \frac{1}{\sin x} \qquad \cot x = \frac{1}{\tan x}$$

The domain and range of these six trigonometric functions are summarized in Table 2-1. Note that if `x = NaN, NegativeInfinity or PositiveInfinity` then these methods will return `NaN` instead of throwing an exception. The following code snippet illustrates the use of all six trigonometric functions:

```
public static double Sec(double x)
{
   return (1.0 / Math.Cos(x));
}
```

TABLE 2.1
Domain and Range of the Trigonometric Functions

Function	Domain	Range
cos x	$-\infty < x < +\infty$	$-1 \leq \cos x \leq +1$
sin x	$-\infty < x < +\infty$	$-1 \leq \sin x \leq +1$
tan x	$-\infty < x < +\infty$ except	
	$x \neq \pm\pi/2, \pm 3\pi/2 \ldots$	$-\infty < \tan x < +\infty$
sec x	$-\infty < x < +\infty$ except	$-\infty < \sec x \leq -1$ and
	$x \neq \pm\pi/2, \pm 3\pi/2 \ldots$	$1 \leq \sec x < +\infty$
csc x	$-\infty < x < +\infty$ except	$-\infty < \csc x \leq -1$ and
	$x \neq 0, \pm\pi, \pm 2\pi \ldots$	$1 \leq \csc x < +\infty$
cot x	$-\infty < x < +\infty$ except	
	$x \neq 0, \pm\pi, \pm 2\pi \ldots$	$-\infty < \cot x < +\infty$

```
public static double Csc(double x)
{
   return (1.0 / Math.Sin(x));
}

public static double Cot(double x)
{
   return (Math.Cos(x) / Math.Sin(x));
}

public static void TrigFunctions_Example()
{
   for (double angleDEG=0.0; angleDEG<=360.0; angleDEG +=45.0)
   {
      double angleRAD = DegreesToRadians(angleDEG);
      Console.WriteLine("Angle = {0}\xb0", angleDEG);
      Console.WriteLine("cos({0}\xb0) = {1}",
                        angleDEG, Math.Cos(angleRAD));
      Console.WriteLine("sin({0}\xb0) = {1}",
                        angleDEG, Math.Sin(angleRAD));
      Console.WriteLine("tan({0}\xb0) = {1}",
                        angleDEG, Math.Tan(angleRAD));
      Console.WriteLine("sec({0}\xb0) = {1}",
                        angleDEG, Sec(angleRAD));
      Console.WriteLine("csc({0}\xb0) = {1}",
                        angleDEG, Csc(angleRAD));
      Console.WriteLine("cot({0}\xb0) = {1}",
                        angleDEG, Cot(angleRAD));
   }
}
```

2.3.8 The Inverse Trigonometric Functions

The .NET Framework Math Class Library provides three methods, ACos x, ASin x, and ATan x, for calculating the corresponding inverses of the three basic trigonometric functions that were just discussed. These methods are declared internally as:

```
public static double ACos(double x);
public static double ASin(double x);
public static double ATan(double x);
```

The other three remaining inverse trigonometric functions, ASec x, ACsc x and ACot x, can be easily calculated from the ones provided by the .NET Framework Math Class Library by taking advantage of some well known trigonometric identities [18] as shown below:

$$\text{ASec}\, x = \text{ACos}\,(\frac{1}{x}) \qquad \text{ACsc}\, x = \text{ASin}\,(\frac{1}{x}) \qquad \text{ACot}\, x = \text{ATan}\,(\frac{1}{x})$$

The input parameter, `x`, and the values returned by these methods are all of type double. However, because of the nature of these inverse functions, their domain and range values have changed from those given in Table 2-1 and are now summarized below.

Method: `Math.ACos(x)`
Input Parameter: A number x of type double such that $-1 \leq x \leq 1$ which represents the cosine of an angle.
Return Value: An angle θ, measured in radians, such that $0 \leq \theta \leq \pi$ and whose cosine is the specified number `x`. If $x < -1$ or $x > 1$ then this method returns `NaN` instead of throwing an exception.

Method: `Math.ASin(x)`
Input Parameter: A number x of type double such that $-1 \leq x \leq 1$ which represents the sine of an angle.
Return Value: An angle θ, measured in radians, such that $-\pi/2 \leq \theta \leq \pi/2$ and whose sine is the specified number `x`. If $x < -1$ or $x > 1$ then this method returns `NaN` instead of throwing an exception.

Method: `Math.ATan(x)`
Input Parameter: A number x of type double such that $-\infty < x < +\infty$ which represents the tangent of an angle.
Return Value: An angle θ, measured in radians, such that $-\pi/2 \leq \theta \leq \pi/2$ and whose tangent is the specified number `x`. If $x =$ `NaN` then this method returns a `NaN` instead of throwing an exception. If $x = -\pi/2$, rounded to double precision, then this method returns a `NegativeInfinity`. If $x = +\pi/2$, rounded to double precision, then this method returns a `PositiveInfinity`.

Unfortunately, the one-argument arctangent function, `ATan(x)`, does not distinguish between diametrically opposite directions thus making the actual angle that it finds rather ambiguous. As a result, the .NET Framework Math Class Library also provides a special two-argument method, called `ATan2(y,x)`, for calculating the arctangent function which takes into account the sign of the coordinate point (x,y)

relative to the origin and places the angle in the correct quadrant. This means that `ATan2(y,x)` effectively calculates the counterclockwise angle in radians between the x-axis and the point (x,y) in a 2-dimensional Cartesian plane. The positive sign is for counter-clockwise angles (upper half-plane, $y > 0$), and negative sign is for clockwise angles (lower half-plane, $y < 0$). Mathematically, the `ATan2(y,x)` method can be derived from the `ATan(x)` by the following formula:

$$\mathrm{ATan2(y,x)} = \begin{cases} \arctan(\frac{y}{x}) & x > 0 \\ \arctan(\frac{y}{x}) + \pi & y \geq 0, x < 0 \\ \arctan(\frac{y}{x}) - \pi & y < 0, x < 0 \\ \frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases}.$$

The `ATan2(y,x)` method provided by the .NET Framework Math Class Library is declared internally as:

```
public static double ATan2(double y, double x);
```

Alternatively, one can also directly program the mathematical formula for $\mathrm{ATan2(y,x)}$ that was just described calling it, say $\mathrm{ATan2A(y,x)}$, as follows:

```
public static double ATan2A(double y, double x)
{
   if (x == 0.0)
   {
      if (y == 0.0) return double.NaN;
      else return (Math.Sign(y) * (Math.PI/2.0));
   }
   else if (x < 0.0)
   {
      if (y < 0.0) return (-Math.PI + Math.Atan(y/x));
      else return (Math.PI + Math.Atan(y/x));
   }
   else return (Math.Atan(y / x));
}
```

Regardless of the choice made, the domain and range of the `ATan2(y,x)` method is summarized below.

Method: `Math.ATan2(y,x)` or `ATan2A(y,x)`
Input Parameter: The (x,y) coordinates of a point relative to the origin of a Cartesian plane.
Return Value: An angle θ, measured in radians, such that $-\pi \leq \theta \leq \pi$, and $\tan\theta = y/x$, where (x,y) is a point in the Cartesian plane. That is, the return value is the angle in the Cartesian plane formed by the x-axis, and a vector starting from the origin, (0,0), and terminating at the point, (x,y) on the plane. Also,

- For (x,y) in quadrant 1, $0 < \theta < \pi/2$.

- For (x,y) in quadrant 2, $\pi/2 < \theta < \pi$.
- For (x,y) in quadrant 3, $-\pi < \theta < -\pi/2$.
- For (x,y) in quadrant 4, $-\pi/2 < \theta < 0$.

For points on the boundaries of the quadrants, the return value is the following:

- If $y = 0$ and $x > 0$, $\theta = 0$.
- If $y = 0$ and $x < 0$, $\theta = \pi$.
- If $y > 0$ and $x = 0$, $\theta = \pi/2$.
- If $y < 0$ and $x = 0$, $\theta = -\pi/2$.

As an aside observation, both the inverse functions $\mathrm{ASin}(x)$ and $\mathrm{ACos}(x)$ can also be alternately expressed and programmed in terms of the inverse tangent function by the trigonometric identities:

$$\mathrm{ASin2(x)} = \begin{cases} \frac{\pi}{2} & x = 1 \\ -\frac{\pi}{2} & x = -1 \\ \arctan(\frac{x}{\sqrt{1-x^2}}) & -1 < x < 1 \end{cases}$$

```
public static double ASin2(double x)
{
  if (x == 1.0) return (Math.PI/2.0);
  else if (x == -1.0) return (-Math.PI/2.0);
  else return (Math.Atan(x/Math.Sqrt(1.0-x*x)));
}
```

$$\mathrm{ACos2(x)} = \begin{cases} 0 & x = 1 \\ \pi & x = -1 \\ \frac{\pi}{2} - \arctan(\frac{x}{\sqrt{1-x^2}}) & -1 < x < 1 \end{cases}$$

```
public static double ACos2(double x)
{
  if (x == 1.0) return 0.0;
  else if (x == -1.0) return Math.PI;
  else return ((Math.PI/2.0)-Math.Atan(x/Math.Sqrt(1.0-x*x)));
}
```

Finally, the following code snippet shows one way to implement the remaining three inverse trigonometric functions that were not provided by the .NET Framework Math Class Library:

```
public static double ASec(double x)
{
   return Math.ACos(1.0 / x);
}

public static double ACsc(double x)
{
   return Math.ASin(1.0 / x);
}

public static double ACot(double x)
{
   return Math.ATan(1.0 / x);
}
```

2.3.9 The Hyperbolic Functions

The hyperbolic functions are related to the hyperbola in much the same way that trigonometric functions are related to the unit circle. For example, the trigonometric functions $\cos x$ and $\sin x$ are related to a circle of radius r because the circle

$$x^2 + y^2 = r^2$$

can be expressed in parametric form by the equations:

$$x = r\cos t, \quad y = r\sin t$$

Likewise, the hyperbolic functions $\sinh x$ and $\cosh x$ are named that way because the hyperbola

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1$$

can be expressed in parametric form by the equations:

$$x = a\cosh t, \quad y = b\sinh t$$

More formally, the basic fundamental hyperbolic functions $\sinh x$ and $\cosh x$ are defined as follows:

$$\cosh x = \frac{e^x + e^{-x}}{2} \qquad \sinh x = \frac{e^x - e^{-x}}{2}$$

where x is a real number. Methods for doing corresponding calculations using complex numbers will be discussed in a later chapter.

The .NET Framework Math Class Library provides three methods for calculating the hyperbolic functions: $\cosh x$, $\sinh x$ and $\tanh x$. These three methods are declared internally as:

```
public static double Cosh(double x);
public static double Sinh(double x);
public static double Tanh(double x);
```

The input parameter `x` must be in radians and is of type double. The output value returned is also of type double. For the hyperbolic cosine function, $\cosh x$, if the input value is equal to either `NegativeInfinity` or `PositiveInfinity`, then ∞ is returned without throwing an exception. If the input value is equal to `NaN`, then `NaN` is returned also without throwing an exception. For the hyperbolic sine function, $\sinh x$, if the input value is equal to `NegativeInfinity`, `PositiveInfinity`, or `NaN`, then that same value is returned without throwing an exception. For the hyperbolic tangent function, $\tanh x$, if the input value is equal to `NegativeInfinity`, then this method returns a -1. If output value is equal to `PositiveInfinity`, then this method returns a 1. If value is equal to `NaN`, then this method returns `NaN` without throwing an exception. The remaining hyperbolic functions are then derived from $\sinh x$ and $\cosh x$ as follows:

$$\tanh x = \frac{\sinh x}{\cosh x} \qquad \coth x = \frac{1}{\tanh x} \qquad \operatorname{sech} x = \frac{1}{\cosh x} \qquad \operatorname{csch} x = \frac{1}{\sinh x}$$

The following code snippet illustrates the use of all six hyperbolic functions that were just discussed.

```
public static double Sech(double x)
{
   return (1.0 / Math.Cosh(x));
}

public static double Csch(double x)
{
   return (1.0 / Math.Sinh(x));
}

public static double Coth(double x)
{
   return (Math.Cosh(x) / Math.Sinh(x));
}

public static void HyperbolicFunctions_Example()
{
  for (double deg=0.0; deg<=360.0; deg += 45.0)
  {
    double rad = DegreesToRadians(deg);
    Console.WriteLine("Cosh({0}\xb0)={1}",deg,Math.Cosh(rad));
    Console.WriteLine("Sinh({0}\xb0)={1}",deg,Math.Sinh(rad));
    Console.WriteLine("Tanh({0}\xb0)={1}",deg,Math.Tanh(rad));
    Console.WriteLine("Sech({0}\xb0)={1}",deg,Sech(rad));
    Console.WriteLine("Csch({0}\xb0)={1}",deg,Csch(rad));
    Console.WriteLine("Coth({0}\xb0)={1}",deg,Coth(rad));
  }
}
```

2.3.10 The Inverse Hyperbolic Functions

Unfortunately, the .NET Framework Math Class Library does not provide internal support for calculating any of the inverse hyperbolic functions:

$$\text{ACosh}\, x, \qquad \text{ASinh}\, x, \qquad \text{ATanh}\, x, \qquad \text{ASech}\, x, \qquad \text{ACsch}\, x, \qquad \text{ACoth}\, x$$

Instead, these functions must all be manually coded directly from their analytical expressions (*e.g.,*Abramowitz and Stegun [19]).

$$\begin{aligned} \text{Asinh}\, x &= \ln(x + \sqrt{x^2+1}) && \text{where } -\infty < x < +\infty \\ \text{Acosh}\, x &= \ln(x + \sqrt{x^2-1}) && \text{where } x \geq 1 \\ \text{Atanh}\, x &= \frac{1}{2}\ln\frac{1+x}{1-x} && \text{where } -1 < x < +1 \end{aligned}$$

Once these three inverse trigonometric functions have been obtained, the remaining others can be easily calculated by using the following identities:

$$\begin{aligned} \text{ACsch}\, x &= \text{ASinh}(\,\frac{1}{x}\,) && \text{where } x \neq 0 \\ \text{ASech}\, x &= \text{ACosh}(\,\frac{1}{x}\,) && \text{where } 0 < x \leq 1 \\ \text{ACoth}\, x &= \text{ATanh}(\,\frac{1}{x}\,) && \text{where } |x| > 1 \end{aligned}$$

The following code snippet illustrates the use of the six inverse hyperbolic functions.

```
public static double ASinh(double x)
{
   return (Math.Log(x + Math.Sqrt(x * x + 1.0)));
}

public static double ACosh(double x)
{
   return (Math.Log(x + Math.Sqrt((x * x) - 1.0)));
}

public static double ATanh(double x)
{
   return (Math.Log((1.0 + x) / (1.0 - x)) / 2.0);
}

public static double ACoth(double x)
{
   return (ATanh(1.0/x));
}
```

```
public static double ASech(double x)
{
   return (ACosh(1.0 / x));
}

public static double ACsch(double x)
{
   return (ASinh(1.0 / x));
}

public static void InverseHyperbolicFunctions_Example()
{
  for (double deg = 0.0; deg <= 360.0; deg += 45.0)
  {
    double rad = DegreesToRadians(deg);
    Console.WriteLine("Angle = {0}\xb0", deg);
    Console.WriteLine("ACosh({0})={1}\xb0",Math.Cosh(rad),
                RadiansToDegrees(ACosh(Math.Cosh(rad))));
    Console.WriteLine("ASinh({0}) = {1}\xb0",Math.Sinh(rad),
                RadiansToDegrees(ASinh(Math.Sinh(rad))));
    Console.WriteLine("ATanh({0}) = {1}\xb0",Math.Tanh(rad),
                RadiansToDegrees(ATanh(Math.Tanh(rad))));
    Console.WriteLine("ASech({0}) = {1}\xb0",Sech(rad),
                RadiansToDegrees(ASech(Sech(rad))));
    Console.WriteLine("ACsch({0}) = {1}\xb0",Csch(rad),
                RadiansToDegrees(ACsch(Csch(rad))));
    Console.WriteLine("ACoth({0}) = {1}\xb0",Coth(rad),
                RadiansToDegrees(ACoth(Coth(rad))));
   }
}
```

2.3.11 Rounding Off Numeric Data

The .NET Framework Math Class Library provides four internal methods for handling the rounding of numerical data. The *Ceiling* and *Floor* functions, for example, map real numbers to the next higher and next lower integers, respectively. The *Truncation* function limits the number of digits to the right of the decimal point by discarding the least significant ones. Finally, the *Round* function rounds a value to the nearest integer or specified number of decimal places. Let us now examine each of these functions more carefully.

The Ceiling Method

The Ceiling function of a real number x, denoted by $\lceil x \rceil$, or $ceil(x)$ or $ceiling(x)$, returns a value that is the smallest integer $\geq x$. More formally,

$$\lceil x \rceil = \min\{n \in \mathbb{Z} \mid n \geq x\}$$

For example, ceiling(2.3) = 3, ceiling(2) = 2 and ceiling(-2.3) = -2.

The method `Math.Ceiling([type] x)` is provided by the .NET Framework Math Class Library and may be overloaded to accommodate both `decimal` and `double` types as input and output values, respectively. The method then returns the largest integer that is $\geq x$. This method is declared internally as:

```
public static [type] Ceiling([type] x);
```

Note that in actuality this method returns a `decimal` or `double` type rather than an expected integer type. If the input parameter x is equal to `NaN`, `NegativeInfinity` or `PositiveInfinity`, then that value is returned instead without first throwing an exception.

The Floor Method

The Floor function of a real number x, denoted by $\lfloor x \rfloor$, or $floor(x)$ or $int(x)$, returns a value that is the largest integer $\leq x$. More formally,

$$\lfloor x \rfloor = \max\{n \in \mathbb{Z} \mid n \leq x\}$$

For example, floor(2.9) = 2, floor(-2) = -2 and floor(-2.4) = -3.

The method `Math.Floor([type] x)` is provided by the .NET Framework Math Class Library and may be overloaded to accommodate both `decimal` and `double` types as input and output values, respectively. The method then returns the largest integer that is $\leq x$. This method is declared internally as:

```
public static [type] Floor([type] x);
```

Note that in actuality this method returns a `decimal` or `double` type rather than an expected integer type. If the input parameter x is equal to `NaN`, `NegativeInfinity` or `PositiveInfinity`, then that value is returned instead without first throwing an exception. In addition, the function $x - \lfloor x \rfloor$, which can also be written as $x \mod 1$, is called the fractional part of x. If $x > 0$, then the $floor(x)$ function is also known as the integral part or integral value of x and is denoted as $int(x)$.

The Truncation Method

The Truncation function simply retains the integral part of a number and discards any remaining fractional digits. Whereas the Floor function rounds down and the Ceiling function rounds up, the Truncation function rounds toward zero. Thus, the Truncation function is like the Floor function for positive numbers, and like Ceiling function for negative numbers.

The method `Math.Truncate([type] x)` is provided by the .NET Framework Math Class Library and may be overloaded to accommodate both `decimal` and `double` types as input and output values, respectively. The method then returns the integral part of a number specified by the input parameter x. This method is declared internally as:

```
public static [type] Truncate([type] x);
```

The following code snippet illustrates how the Ceiling, Floor and Truncation methods may be used in an application.

```
double[] values = {7.03,7.64,0.12,-0.12,-7.1,-7.6 };
Console.WriteLine("Value   Ceiling   Floor   Truncate\n");
foreach (double value in values)
  Console.WriteLine("{0,5} {1,5} {2,8} {3,8}", value,
  Math.Ceiling(value),Math.Floor(value),Math.Truncate(value));

Output:
Value      Ceiling     Floor    Truncate
 7.03         8          7         7
 7.64         8          7         7
 0.12         1          0         0
-0.12         0         -1         0
-7.1         -7         -8        -7
-7.6         -7         -8        -7
```

The Round Method

Because of the physical limitations imposed by hardware, computers cannot correctly store irrational numbers, such as π or $\sqrt{2}$, or non-terminating rational numbers in floating-point format. In addition, the number of digits (or bits) of precision also limits the amount of rational numbers that can be represented exactly. Instead, all such numbers must at some point be approximated and adjusted to a *rounded value*. Unfortunately, such small errors inadvertently introduced into floating-point arithmetic can sometimes grow significantly large, particularly when mathematical algorithms are required to repeatedly perform certain operations, and this can sometimes produce misleading or even erroneous results. Because of these issues, naive use of floating-point arithmetic can lead to many unwanted problems and the creation of robust floating-point software can be quite a complicated undertaking. However, one important step towards properly handling floating-point values is to first determine how they will be rounded off.

The basic idea behind the concept of rounding off a numeric value is to somehow systematically reduce the number of significant digits that it contains. Since there are many ways of actually doing this, several different rounding algorithms, schemes or modes have been developed and cataloged over the years [20]. Perhaps the easiest of all these rounding algorithms is the one taught in elementary school which is more commonly known as the Symmetric Arithmetic Rounding or Round-Half-Up. It consists of the following steps:

- Decide which is the last digit to keep.
- Increase it by 1 if the next digit is 5 or more. This is called rounding up.
- Leave it the same if the next digit is 4 or less. This is called rounding down.

Unfortunately, in the process of standardizing the computer representation for binary floating-point numbers, the *IEEE* chose to use the controversial *Banker's* Rounding Algorithm as their default rounding method to be implemented by all *IEEE* compliant software compilers in the industry, including Visual Studio. With the *Banker's* Rounding Algorithm, the input value is rounded to the nearest even number and this can sometimes lead to unexpected results. For example, both 1.5 and 2.5 round to 2, and 3.5 and 4.5 both round to 4. As a result, programmers have had to write their own rounding methods, such as `RoundUp` and `RoundDown` shown below, if they wanted to use the more familiar Symmetric Arithmetic rounding algorithm just described.

```
public static double RoundUp(double x)
{
   return Math.Floor(x + 0.5);
}

public static double RoundDown(double x)
{
   double floorVal = Math.Floor(x);
   if ((x - floorVal) > 0.5)
   {
      return (floorVal + 1.0);
   }
   else
   {
      return (floorVal);
   }
}
```

This rather annoying issue was finally resolved with the release of the 2005 version of Visual Studio. Now the .NET Framework Math Library provides a method called `Math.Round` that can be overloaded in a number of different ways in order to provide various rounding schemes. Alternative rounding schemes are useful when the amount of error being introduced into a calculation must somehow be bounded. Such applications usually involve multi-precision, floating-point and interval arithmetic calculations. A comprehensive summary of the various rounding schemes that are available with the `Math.Round` method is given below. Whatever method is eventually chosen, note that the allowed input data type, specified by `[type]`, may be either a `decimal` or a `double`.

`Round(x)` - Rounds the input numeric value `x` to the nearest integral value. The method is declared internally as: `public static [type] Round([type] x);`. The output returns a numeric value nearest to the parameter `x`. If the fractional component of `x` is halfway between two integers, one of which is even and the other odd, then the even number is returned. Note that the method returns a `decimal` or a `double` rather than an integral type. This method throws an overflow exception if the output result resides outside the respective ranges of the data type specified by `type`. This rounding scheme minimizes rounding errors that result from consistently rounding a midpoint value in a single direction. To control the type of rounding used by the `Round(x)` method, use the method `Round(x, MidpointRounding)` overload described later in this same chapter.

`Round(x, n)` - Rounds the input numeric value `x` to a specified number of decimal places explicitly given by the other input parameter `n`. The method is declared internally as: `public static [type] Round([type] x, int n);`. The input parameter `n` specifies the number of decimal places to round off the return value and ranges from 0 to 28. If `n`<0 or `n`>28 then an argument-out-of-range exception is thrown.

If `n` is zero, then an integer is returned. If the value of the first digit in `x` to the right of the decimal position represented by the parameter `n` is 5, the digit in the decimals position is rounded up if it is odd, or left unchanged if it is even. If the precision of `x` < `n`, then `x` is returned unchanged. Note that the method returns a `decimal` or a `double` rather than an integral type. This method throws an overflow exception if the output result is outside the respective ranges of the data type specified by `[type]`. This rounding scheme minimizes rounding errors that result from consistently rounding a midpoint value in a single direction. To control the type of rounding used by the `Round(x,n)` method, use the method `Round(x,n, MidpointRounding)` overload described later in this same chapter.

`Round(x, mode)` - Rounds the input numeric value `x` to the nearest integral value. The other input parameter, `mode`, specifies how to round the value `x` if it is midway between two other numbers. The method is declared internally as:

```
public static [type] Round([type] x, MidpointRounding mode);
```

The output returns a numeric value nearest to the input parameter `x`. If `x` is halfway between two numbers, one of which is even and the other odd, then the parameter `mode` determines which of the two is returned. The `mode` parameter can have one of two enumerated values: `MidpointRounding.[ToEven]` or `.[AwayFromZero]`. If the `mode` parameter is set to `.[ToEven]`, then if one's digit is odd, it is changed to an even digit. Otherwise it is left unchanged. If the precision of `x` is less than `n`, then `x` is returned unchanged. If the mode parameter is set to `.[AwayFromZero]`, then one's digit is always rounded up to the next digit. This is the most familiar Symmetric Arithmetic rounding algorithm described earlier at the start of this section. Lastly, this method can throw two exceptions. The `Argument` exception is returned if the mode is not a valid value of either `.[ToEven]` or `.[AwayFromZero]`. An `Overflow` exception is returned if the output result is outside the range of the specified `[type]` data type.

`Round(x, n, mode)` - Rounds the input numeric value `x` to a specified number of decimal places explicitly given by the second input parameter `n`. The third input parameter, `mode`, specifies how to round the value `x` if it is midway between two other numbers. The method is declared internally as:

```
public static [type] Round([type] x, int n, MidpointRounding mode);
```

The output returns a numeric value nearest to the input parameter `x` to a specified precision given by the second input parameter `n`. If `x` is halfway between two numbers, one of which is even and the other odd, then the parameter `mode` determines which of the two is returned. The `mode` parameter can have one of two enumerated values: `MidpointRounding.[ToEven]` or `.[AwayFromZero]`. If the `mode` parameter is set to `ToEven`, then if one's digit is odd, it is changed to an even digit. Otherwise it is left unchanged. This kind of rounding minimizes rounding errors that result from consistently rounding a midpoint value in a single direction. If the mode parameter is set to `AwayFromZero`, then one's digit is always rounded up to the next digit.

This is the most familiar Symmetric Arithmetic rounding algorithm described earlier at the start of this section. Lastly, this method can also throw three exceptions. The `Argument-out-of-range` exception is thrown if $n < 0$ or if $n > 28$. The `Argument` exception is thrown if the mode is not a valid value of either `ToEven` or `AwayFromZero`. The `Overflow` exception is thrown if the output lies outside the range of the specified `[type]` data type. The following code snippet illustrates how the rounding methods discussed: `Round(x)`, `Round(x, n)`, `Round(x, mode)`, and `Round(x, n, mode)` might be used in an actual application.

```
Console.WriteLine("Example of Using Method: Round(x)");
Console.WriteLine("Round(4.4)={0}",Math.Round(4.4));
Console.WriteLine("Round(4.5)={0}",Math.Round(4.5));
Console.WriteLine("Round(4.6)={0}",Math.Round(4.6));
Console.WriteLine("Round(-4.4)={0}",Math.Round(-4.4));
Console.WriteLine("Round(-4.5)={0}",Math.Round(-4.5));
Console.WriteLine("Round(-4.6)={0}\n",Math.Round(-4.6));

Console.WriteLine("Example of Using Method: Round(x,n)");
Console.WriteLine("Round(4.44,1)={0}",Math.Round(4.44,1));
Console.WriteLine("Round(4.45,1)={0}",Math.Round(4.45,1));
Console.WriteLine("Round(4.46,1)={0}",Math.Round(4.46,1));
Console.WriteLine("Round(-4.44,1)={0}",Math.Round(-4.44,1));
Console.WriteLine("Round(-4.45,1)={0}",Math.Round(-4.45,1));
Console.WriteLine("Round(-4.46,1)={0}\n",Math.Round(-4.46,1));

double result = 0.0;
double posValue = 3.45;
double negValue = -3.45;

Console.WriteLine("Example of Using Method: Round(x,n)");
// By default, round a positive and a negative value
// to the nearest even number.
// The precision of the result is 1 decimal place.

result = Math.Round(posValue,1);
Console.WriteLine("Math.Round({1,5},1)={0,4}",
   result,posValue);
result = Math.Round(negValue,1);
Console.WriteLine("Math.Round({1,5},1)={0,4}\n",
   result,negValue);

Console.WriteLine("Example of Using Method:
   Round(x,n,mode)");
// Round a positive value to the nearest even number,
// then to the nearest number away from zero.
// The precision of the result is 1 decimal place.

result = Math.Round(posValue,1,MidpointRounding.ToEven);
Console.WriteLine("Math.Round({1,5},1,
   MidpointRounding.ToEven)={0,4}",result,posValue);
result = Math.Round(posValue, 1,
   MidpointRounding.AwayFromZero);
Console.WriteLine("Math.Round({1,5},1,
   MidpointRounding.AwayFromZero)={0,4}\n",result,posValue);
```

```
// Round a negative value to the nearest even number,
// then to the nearest number away from zero.
// The precision of the result is 1 decimal place.

result = Math.Round(negValue,1,
   MidpointRounding.ToEven);
Console.WriteLine("Math.Round({1,5},1,
   MidpointRounding.ToEven)={0,4}", result, negValue);
result = Math.Round(negValue,1,
   MidpointRounding.AwayFromZero);
Console.WriteLine("Math.Round({1,5},1,
   MidpointRounding.AwayFromZero)={0,4}", result, negValue);

result = Math.Round(1.5,MidpointRounding.AwayFromZero);
string s ="Math.Round(1.5,MidpointRounding.AwayFromZero)={0}"
Console.WriteLine(s, result);
result = Math.Round(2.5,MidpointRounding.AwayFromZero);
s = "Math.Round(2.5,MidpointRounding.AwayFromZero)={0}"
Console.WriteLine(s, result);

result = Math.Round(1.5, MidpointRounding.ToEven);
s = "Math.Round(1.5,MidpointRounding.ToEven)={0}";
Console.WriteLine(s, result);
result = Math.Round(2.5, MidpointRounding.ToEven);
s = "Math.Round(2.5,MidpointRounding.ToEven)={0}";
Console.WriteLine(s, result);

//Testing the RoundUp Method: output = 2
Console.WriteLine("RoundUp(1.5)={0}",RoundUp(1.5));
//Testing the RoundDown Method: output = 1
Console.WriteLine("RoundDown(1.5)={0}",RoundDown(1.5));

//Testing the RoundUp Method
Console.WriteLine("\n\nRoundUp(.4) = {0}", RoundUp(.4));
Console.WriteLine("RoundUp(.5) = {0}", RoundUp(.5));
Console.WriteLine("RoundUp(.6) = {0}", RoundUp(.6));
Console.WriteLine("RoundUp(1.4) = {0}", RoundUp(1.4));
Console.WriteLine("RoundUp(1.5) = {0}", RoundUp(1.5));
Console.WriteLine("RoundUp(1.6) = {0}", RoundUp(1.6));
Console.WriteLine("RoundUp(2.4) = {0}", RoundUp(2.4));
Console.WriteLine("RoundUp(2.5) = {0}", RoundUp(2.5));
Console.WriteLine("RoundUp(2.6) = {0}", RoundUp(2.6));

//Testing the RoundDown Method
Console.WriteLine("\nRoundDown(.4) = {0}", RoundDown(.4));
Console.WriteLine("RoundDown(.5) = {0}", RoundDown(.5));
Console.WriteLine("RoundDown(.6) = {0}", RoundDown(.6));
Console.WriteLine("RoundDown(1.4) = {0}", RoundDown(1.4));
Console.WriteLine("RoundDown(1.5) = {0}", RoundDown(1.5));
Console.WriteLine("RoundDown(1.6) = {0}", RoundDown(1.6));
Console.WriteLine("RoundDown(2.4) = {0}", RoundDown(2.4));
Console.WriteLine("RoundDown(2.5) = {0}", RoundDown(2.5));
Console.WriteLine("RoundDown(2.6) = {0}", RoundDown(2.6));
```

3

Vectors and Matrices

3.1 Introduction

Vectors are so fundamental in mathematics, including the natural sciences and engineering that this chapter hardly needs an introductory section on this topic [21]. However, in order to make a more complete presentation and also provide the requisite background material for this chapter, the topic of vectors will be briefly summarized before a suggested implementation of them in C# is given. Matrices are also closely associated with vectors in the sense that an n-element single column or single row matrix can be used to represent a vector in n-dimensional space. As a result of this close relationship, the focus of this chapter is on developing both a real number vector and a real number matrix library in C#. Their equivalent complex number counterparts will be addressed in a later chapter that exclusively covers the topic of complex numbers. To avoid issues inherent with floating number data types that already have been discussed in Chapter 1, this vector and matrix library will be developed using variables declared as double precision data types.

Vectors are used to represent any quantity that has both a magnitude and direction. Vectors can be added, subtracted, multiplied by a number, and flipped around so that their original direction is reversed. These operations obey the familiar algebraic laws of commutativity, associativity, and distributivity. The sum of two vectors with the same initial point can be found geometrically using the parallelogram law. Multiplication by a positive number, commonly called a scalar in this context, amounts to changing the magnitude of the vector in the sense of stretching or compressing it while maintaining its direction. Multiplication by negative numbers changes the magnitude and reverses the vector's direction. However, vector multiplication by another vector is not uniquely defined. Instead, a number of different types of products, such as the dot product, cross product, and tensor direct product can be defined for pairs of vectors.

In mathematics, a matrix (plural matrices) is just a rectangular array of numbers consisting of m rows and n columns [21].

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

The horizontal and vertical lines in a matrix are called *rows* and *columns*, respectively. The numbers in the matrix are called its *entries*. To specify the size of a matrix, a matrix with m rows and n columns is called an $m \times n$ matrix. Such a matrix is said to have an *order* of $m \times n$ where m and n are called its *dimensions*. A matrix where one of the dimensions equals one is also called a *vector*. Consequently, an $m \times 1$ matrix (one column and m rows) is called a *column matrix* or *column vector*

$$\vec{A} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix}$$

and a $1 \times n$ matrix (one row and n columns) is called a *row matrix* or *row vector*

$$\vec{A} = \begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix}$$

3.2 A Real Number Vector Library in C#

The first order of business in developing a vector library in C# is deciding which data value type to use for storing the vector information. The most natural compulsion for developers working with an object-oriented programming language, such as C#, is to think that just about everything should be described by a class only to be instantiated later as an object upon which many additional wonderful things can be made to happen. Therefore, it might come as a complete surprise to some readers that my choice of data type to implement vectors was a `struct` instead of a `class`. To answer this question, we should perhaps take a moment to review the key differences between these two very important data value types.

A `struct` is a value type created on the stack whereas a `class` is a reference type created on the heap. This means that a variable of a `struct` type directly contains the data whereas a variable of a `class` type, known as an object, contains only a reference to the data. Using a value type instead of a reference type will result in fewer objects on the managed heap, which in turn results in a lesser load for the garbage collector, less frequent garbage collector cycles, and consequently better performance. However, `struct` value types have their own drawbacks as well. Passing around a variable of type `struct` is definitely costlier than passing around a reference type and so a `struct` is therefore particularly useful only for small data structures that can be conveniently implemented using value semantics where data assignment copies the value instead of the reference. Microsoft recommends that a `struct` should be less than 16 bytes. As a result, a `struct` is more suitable for representing lightweight objects such as vectors.

The real vector `struct` will be called by `RVector`in order to distinguish it from its complex vector counterpart, `CVector`, which will be introduced in a later chapter that covers complex numbers exclusively. The two constructors should be able to create an initialized vector of a given length (or size) as well as a vector converted from a real double array.

```
public struct RVector : ICloneable
{
    private int ndim;
    private double[] vector;

    public RVector(int ndim)
    {
        this.ndim = ndim;
        this.vector = new double[ndim];
        for (int i = 0; i < ndim; i++)
        {
            vector[i] = 0.0;
        }
    }

    public RVector(double[] vector)
    {
        this.ndim = vector.Length;
        this.vector = vector;
    }
```

We are also going to need to define a number of operators and methods in the `RVector struct` to handle the most common vector operations. First, we need an indexing property in order to access the *n*-th element of a vector more easily just like the *n*-th element of an array.

```
    public double this[int i]
    {
        get
        {
            if (i < 0 || i > ndim)
            {
                throw new Exception("Requested vector index is out of
                     range!");
            }
            return vector[i];
        }
        set { vector[i] = value; }
    }
```

We also need a method to read the size or dimension of a vector which is stored as a private variable.

```
    public int GetVectorSize
    {
        get { return ndim; }
    }
```

We also need another method to clone this vector `struct`. This feature is used to make clone copies of a particular vector if the need ever arises. The utility `SwapVectorEntries` is for swapping vector entries if it is ever needed.

```
public RVector Clone()
{
    RVector v = new RVector(vector);
    v.vector = (double[])vector.Clone();
    return v;
}

object ICloneable.Clone()
{
    return Clone();
}

public RVector SwapVectorEntries(int m, int n)
{
    double temp = vector[m];
    vector[m] = vector[n];
    vector[n] = temp;
    return new RVector(vector);
}
```

In addition, it would be very useful to be able to display the contents of our vector `struct` in such a way that it can be more easily identified with the way vectors are commonly written down in standard textbooks. For that we need to override the default `ToString` method as shown in the following code.

```
public override string ToString()
{
    string str = "(";
    for (int i = 0; i < ndim - 1; i++)
    {
        str += vector[i].ToString() + ", ";
    }
    str += vector[ndim - 1].ToString() + ")";
    return str;
}
```

Sometimes when using conditional statements in code, vectors need to be compared with each other. The `System.Object` type provides a virtual method called `Equals` designed to return a boolean type variable to indicate whether or not two objects have the same *value*. Since an object's *value* is an abstract concept, we need to define explicitly what we mean for two vectors to be equal or not equal to each other. Two vectors are said to be equal if they have the same magnitude and direction. Mathematically, two vectors $\vec{A}$ and $\vec{B}$ are said to be equal if their coordinates are equal to each other. Therefore,

$$\vec{A} = \big(a_1, a_2, \cdots, a_n\big) \quad \text{and} \quad \vec{B} = \big(b_1, b_2, \cdots, b_n\big)$$

are said to be equal to each other if and only if $a_1 = b_1, a_2 = b_2, \ldots, a_n = b_n$. Checking for equality in vectors can therefore be implemented as follows.

```
public override bool Equals(object obj)
{
    return (obj is RVector) && this.Equals((RVector)obj);
}
```

```
    public bool Equals(RVector v)
    {
        return vector == v.vector;
    }

    public override int GetHashCode()
    {
        return vector.GetHashCode();
    }

    public static bool operator ==(RVector v1, RVector v2)
    {
        return v1.Equals(v2);
    }

    public static bool operator !=(RVector v1, RVector v2)
    {
        return !v1.Equals(v2);
    }
```

We are now ready to start developing additional methods for carrying out more explicit vector operations, such as vector addition and subtraction. Note that, because of their very nature, vectors can only be added to or subtracted from other vectors. In order to carry out vector addition and subtraction, it would be helpful to develop methods to override their default $\pm$ mathematical operators as shown below. The *sum* of vectors $\vec{A} = \left(a_1, a_2, \cdots, a_n\right)$ and $\vec{B} = \left(b_1, b_2, \cdots, b_n\right)$ is given by

$$\vec{A} + \vec{B} = \left(a_1 + b_1, a_2 + b_2, \cdots, a_n + b_n\right)$$

and can be implemented in by

```
    public static RVector operator +(RVector v)
    {
        return v;
    }

    public static RVector operator +(RVector v1, RVector v2)
    {
        RVector result = new RVector(v1.ndim);
        for (int i = 0; i < v1.ndim; i++)
        {
            result[i] = v1[i] + v2[i];
        }
        return result;
    }
```

The *difference* between vectors $\vec{A} = \left(a_1, a_2, \cdots, a_n\right)$ and $\vec{B} = \left(b_1, b_2, \cdots, b_n\right)$ is given by

$$\vec{A} - \vec{B} = \left(a_1 - b_1, a_2 - b_2, \cdots, a_n - b_n\right)$$

and can be implemented by

```
    public static RVector operator -(RVector v)
    {
        double[] result = new double[v.ndim];
        for (int i = 0; i < v.ndim; i++)
        {
            result[i] = -v[i];
        }
        return new RVector(result);
    }

    public static RVector operator -(RVector v1, RVector v2)
    {
        RVector result = new RVector(v1.ndim);
        for (int i = 0; i < v1.ndim; i++)
        {
            result[i] = v1[i] - v2[i];
        }
        return result;
    }
```

A vector may also be multiplied, or re-scaled, by a real number r. In the context of conventional vector algebra, these real numbers are often called scalars (from scale) to distinguish them from vectors which have a sense of direction in addition to a scalar magnitude. The operation of multiplying a vector by a scalar is called *scalar multiplication*. The resulting vector is

$$r\vec{A} = \left(ra_1, ra_2, \cdots, ra_n\right)$$

Geometrically, multiplying by a positive scalar r stretches a vector out by a factor of r. Dividing a vector by a factor r is equivalent to multiplying it by the amount of $1/r$, Physically, this action has the effect of compressing the vector by a factor of $1/r$. If the scalar is a negative number, then the direction of the vector is reversed from its original position. Although vectors can be divided by a scalar, the converse is not true and a scalar that is divided by a vector is undefined.

```
    public static RVector operator *(RVector v, double d)
    {
        RVector result = new RVector(v.ndim);
        for (int i = 0; i < v.ndim; i++)
        {
            result[i] = v[i] * d;
        }
        return result;
    }

    public static RVector operator *(double d, RVector v)
    {
        RVector result = new RVector(v.ndim);
        for (int i = 0; i < v.ndim; i++)
        {
            result[i] = d * v[i];
        }
        return result;
    }
```

```
public static RVector operator /(RVector v, double d)
{
    RVector result = new RVector(v.ndim);
    for (int i = 0; i < v.ndim; i++)
    {
        result[i] = v[i] / d;
    }
    return result;
}

public static RVector operator /(double d, RVector v)
{
    RVector result = new RVector(v.ndim);
    for (int i = 0; i < v.ndim; i++)
    {
        result[i] = v[i] / d;
    }
    return result;
}
```

Both vector multiplication and division by another vector requires more careful consideration because such operations are defined very differently from those involving scalar variables and are even given different names. For example, the *dot product* of two vectors $\vec{A} = (a_1, a_2, \ldots, a_n)$ and $\vec{B} = (b_1, b_2, \ldots, b_n)$ is defined by:

$$\vec{A} \cdot \vec{B} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

The code for the dot product of two n-dimensional vectors can be expressed as shown below.

```
public static double DotProduct(RVector v1, RVector v2)
{
    double result = 0.0;
    for (int i = 0; i < v1.ndim; i++)
    {
        result += v1[i] * v2[i];
    }
    return result;
}
```

The *length* or *magnitude* or *norm* of the vector $\vec{A}$ is denoted by $||\vec{A}||$ or, less commonly, by $|\vec{A}|$, which is not to be confused with the absolute value which is a scalar *norm*. The length of the vector $\vec{A} = \left(a_1, a_2, \cdots, a_n\right)$ can be computed with the Euclidean norm

$$||\vec{A}|| = \sqrt{a_1{}^2 + a_2{}^2 + a_3{}^2 + \ldots + a_n{}^2}$$

which happens to be equal to the square root of the dot product of the vector with itself:

$$||\vec{A}|| = \sqrt{\vec{A} \cdot \vec{A}}.$$

The corresponding code for calculating both the norm and norm square of a vector is shown below.

```
public double GetNorm()
{
    double result = 0.0;
    for (int i = 0; i < ndim; i++)
    {
        result += vector[i] * vector[i];
    }
    return Math.Sqrt(result);
}

public double GetNormSquare()
{
    double result = 0.0;
    for (int i = 0; i < ndim; i++)
    {
        result += vector[i] * vector[i];
    }
    return result;
}
```

In three-dimensional Euclidean geometry, the dot product of two vectors, say $\vec{A}$ and $\vec{B}$, can also be expressed by the following expression

$$\vec{A} \cdot \vec{B} = ||\vec{A}||\,||\vec{B}|| \cos\theta$$

where $||\vec{A}||$ and $||\vec{B}||$ denote the length of $\vec{A}$ and $\vec{B}$, respectively, and θ is the angle between them.

A *unit* vector is any vector with a length of one. Normally unit vectors are used to simply indicate direction and are often indicated with a little hat on top of the vector itself, such as $\hat{a}$. In the three-dimensional Cartesian coordinate system, the unit vectors are co-directional with the x, y, and z axes. In particular, the unit vectors $\hat{\imath}$, $\hat{\jmath}$, and $\hat{k}$ are said to form a standard orthonormal basis along the x, y, and z axes respectively and a general three-dimensional vector $\vec{A}$ can therefore be written as

$$\vec{A} = a_1\hat{\imath} + a_2\hat{\jmath} + a_3\hat{k} = a_1\hat{x} + a_2\hat{y} + a_3\hat{z} = (a_1, a_2, a_3)$$

where the unit vectors $\hat{\imath}$, $\hat{\jmath}$, and $\hat{k}$ may also be expressed as

$$\hat{\imath} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix},\ \hat{\jmath} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},\ \hat{k} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The additional notations ($\hat{x}$, $\hat{y}$, $\hat{z}$), ($\hat{x}_1$, $\hat{x}_2$, $\hat{x}_3$), ($\hat{e}_x$, $\hat{e}_y$, $\hat{e}_z$), or ($\hat{e}_1$, $\hat{e}_2$, $\hat{e}_3$), with or without hat/caret, are also used, particularly in contexts where $\hat{\imath}$, $\hat{\jmath}$, $\hat{k}$ might lead to confusion with another quantity, such as the index symbols i, j, k, used to identify an element of a set or an array or a sequence of variables.

A vector of arbitrary length can be divided by its own length to create a unit vector. This process is known as normalizing a vector and after a vector, say $\vec{A}$, gets normalized, it is usually rewritten with a little hat on top of it as shown here: $\hat{A}$. Mathematically, to normalize a vector $\vec{A} = (a_1, a_2, a_3, \ldots, a_n)$, scale the vector by the reciprocal of its length $||\vec{A}||$ thus obtaining the following expression.

$$\hat{A} = \frac{\vec{A}}{||A||}$$

The corresponding code for normalizing a vector is given by

```
public void Normalize()
{
    double norm = GetNorm();
    if (norm == 0)
    {
        throw new Exception("Tried to normalize a vector with
            norm of zero!");
    }
    for (int i = 0; i < ndim; i++)
    {
        vector[i] /= norm;
    }
}

public RVector GetUnitVector()
{
    RVector result = new RVector(vector);
    result.Normalize();
    return result;
}
```

The *cross product*, also called the *vector product* or *outer product*, is only meaningful in three dimensions. The cross product differs from the dot product primarily in that the result of the cross product of two vectors is also a vector. The cross product, denoted by $\vec{A} \times \vec{B}$, is a vector perpendicular to both $\vec{A}$ and $\vec{B}$ and is defined as

$$\vec{A} \times \vec{B} = ||\vec{A}||\,||\vec{B}|| \sin\theta\, \hat{n}$$

where θ is the measure of the angle between $\vec{A}$ and $\vec{B}$, and $\hat{n}$ is a unit vector perpendicular to both $\vec{A}$ and $\vec{B}$ which completes a right-handed system. The right-handedness constraint is necessary because there exists two unit vectors that are perpendicular to both $\vec{A}$ and $\vec{B}$, namely, $\hat{n}$ and $-\hat{n}$.

The cross product $\vec{A} \times \vec{B}$ is defined so that $\vec{A}$, $\vec{B}$, and $\vec{A} \times \vec{B}$ also becomes a right-handed system. However, note that $\vec{A}$ and $\vec{B}$ are not necessarily orthogonal.

The cross product can be written as

$$\vec{A} \times \vec{B} = (a_2b_3 - a_3b_2)\hat{\imath} + (a_3b_1 - a_1b_3)\hat{\jmath} + (a_1b_2 - a_2b_1)\hat{k}$$

The definition of the cross product can also be represented by the determinant of a matrix as shown below:

$$\vec{A} \times \vec{B} = \det \begin{bmatrix} \hat{\imath} & \hat{\jmath} & \hat{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{bmatrix} = (a_2b_3 - a_3b_2)\hat{\imath} + (a_3b_1 - a_1b_3)\hat{\jmath} + (a_1b_2 - a_2b_1)\hat{k}$$

The corresponding code for the cross product of two vectors is given by:

```
    public static RVector CrossProduct(RVector v1, RVector v2)
    {
        if (v1.ndim != 3)
        {
            throw new Exception("Vector v1 must be 3 dimensional!");
        }
        if (v2.ndim != 3)
        {
            throw new Exception("Vector v2 must be 3 dimensional!");
        }
        RVector result = new RVector(3);
        result[0] = v1[1] * v2[2] - v1[2] * v2[1];
        result[1] = v1[2] * v2[0] - v1[0] * v2[2];
        result[2] = v1[0] * v2[1] - v1[1] * v2[0];
        return result;
    }
}
```

3.3 A Real Number Matrix Library in C#

Matrices, and more generally, linear algebra have many applications in mathematics, the natural sciences and engineering [21]. Due to their widespread use, considerable effort has been made to develop efficient methods for computing matrices both efficiently and effectively, particularly if the matrices are big. To this end, there are several matrix decomposition methods, which express matrices as products of other matrices, whose inverses, products etc. are easier to compute. As a result of their usefulness in so many different disciplines, the remainder of this chapter will be focused on developing a set of tools for implementing a real number matrix in C#. The equivalent complex number matrix counterparts will be addressed in a later chapter that exclusively covers the topic of complex numbers. More advanced applications, such as the solution of linear systems of algebraic equations or calculating eigenvalues and eigenvectors, will be deferred to yet a later chapter.

As briefly described in the introduction of this chapter, a matrix is simply a rectangular array of numbers consisting of m rows and n columns [21].

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

The horizontal and vertical lines of a matrix are called *rows* and *columns*, respectively. The numbers in the matrix are called its *entries*. To specify the size of a matrix, a matrix with m rows and n columns is called an $m \times n$ matrix. Such a matrix is said to have an *order* of $m \times n$ where m and n are called its *dimensions*. For convenience, the entries of a matrix A are often denoted by $A_{i,j}$ where i and j represent the i-th row and j-th column. The matrix A itself can also be represented by $A = [a_{i,j}]_{m \times n}$.

A real matrix data structure can be constructed using a two-dimensional array, one for rows and another one for columns. Although many matrices have equal number of rows and columns which greatly simplifies calculations, this is not always the case and we should therefore allow for the general possibility of having to declare matrices accordingly. The matrix constructors below allow for the creation of a $m \times n$ matrix object using two-dimensional double precision array. The first constructor accepts integer values for `nRows` and `nRows` as input parameters and initializes all the matrix entries to zero. The second constructor creates a matrix that holds a specified two-dimensional array with the size of the matrix being the same as that of the dimensions of the array.

```
public struct RMatrix : ICloneable
{
    private int nRows;
    private int nCols;
    private double[,] matrix;

    public RMatrix(int nRows, int nRows)
    {
        this.nRows = nRows;
        this.nCols = nCols;
        this.matrix = new double[nRows, nCols];
        for (int i = 0; i < nRows; i++)
        {
            for (int j = 0; j < nCols; j++)
            {
                matrix[i, j] = 0.0;
            }
        }
    }

    public RMatrix(double[,] matrix)
    {
        this.nRows = matrix.GetLength(0);
        this.nCols = matrix.GetLength(1);
        this.matrix = matrix;
    }

    public RMatrix(RMatrix m)
    {
        nRows = m.GetnRows;
        nCols = m.GetnCols;
        matrix = m.matrix;
    }
```

A matrix whose entries are all equal to 1 is called the identity matrix and such matrices are of particular importance in matrix theory. The identity matrix can be implemented very easily as shown below.

```
public RMatrix IdentityMatrix()
{
    RMatrix m = new RMatrix(nRows, nCols);
    for (int i = 0; i < nRows; i++)
    {
        for (int j = 0; j < nCols; j++)
        {
            if (i == j)
            {
                m[i, j] = 1;
            }
        }
    }
    return m;
}
```

As in the case with vectors, we also need to define an indexing property in order to access the entries of the matrix object more easily by simply specifying the desired row and column of interest. For example, if M denotes a matrix object then $M[i, j]$ or $M_{i,j}$ denotes a matrix element consisting of the entry located at the i-th row and j-th column of matrix M. This type of notation is very much like those found in standard mathematical textbooks on this topic.

```
public double this[int m, int n]
{
    get
    {
        if (m < 0 || m > nRows)
        {
            throw new Exception("m-th row is out of range!");
        }
        if (n < 0 || n > nCols)
        {
            throw new Exception("n-th col is out of range!");
        }
        return matrix[m, n];
    }
    set { matrix[m, n] = value; }
}
```

We also need a couple of properties to read the dimensions of the RMatrix which are stored as a private variables.

```
public int GetnRows
{
    get { return nRows; }
}

public int GetnCols
{
    get { return nCols; }
}
```

We also need another method to clone this matrix `struct`. This feature is used to make clone copies of a particular matrix if the need ever arises.

```
public RMatrix Clone()
{
    RMatrix m = new RMatrix(matrix);
    m.matrix = (double[,])matrix.Clone();
    return m;
}

object ICloneable.Clone()
{
    return Clone();
}
```

As in the case with vectors, it would be nice to have a customized way to display matrices on the computer screen so that they may then look like their counterparts found in textbooks. This is accomplished by overriding the `ToString()` method.

```
public override string ToString()
{
    string strMatrix = "(";
    for (int i = 0; i < nRows; i++)
    {
        string str = "";
        for (int j = 0; j < nCols - 1; j++)
        {
            str += matrix[i, j].ToString() + ", ";
        }
        str += matrix[i, nCols - 1].ToString();
        if (i != nRows - 1 && i == 0)
            strMatrix += str + "\n";
        else if (i != nRows - 1 && i != 0)
            strMatrix += " " + str + "\n";
        else
            strMatrix += " " + str + ")";
    }
    return strMatrix;
}
```

Sometimes when using conditional statements in code, matrices need to be compared with each other. The `System.Object` type provides a virtual method called `Equals` designed to return a boolean type variable to indicate whether or not two objects have the same *value*. Since an object's *value* is an abstract concept, we need to define explicitly what we mean for two matrices to be equal or not equal to each other. Mathematically, Two matrices A and B are said to be equal to each other if and only if $A_{i,j} = B_{i,j}$ for all i, j. Checking for equality in matrices can therefore be implemented as follows.

```
public override bool Equals(object obj)
{
    return (obj is RMatrix) && this.Equals((RMatrix)obj);
}
public bool Equals(RMatrix m)
{
    return matrix == m.matrix;
}
```

```
public override int GetHashCode()
{
    return matrix.GetHashCode();
}

public static bool operator ==(RMatrix m1, RMatrix m2)
{
    return m1.Equals(m2);
}

public static bool operator !=(RMatrix m1, RMatrix m2)
{
    return !m1.Equals(m2);
}
```

We are now ready to start developing additional methods for carrying out more explicit matrix operations, such as matrix addition and subtraction. Note that, because of their very nature, matrices can only be added to or subtracted from other matrices provided both matrices have the same number of rows and columns. In order to carry out matrix addition and subtraction, it would therefore be helpful to develop methods to override the default $\pm$ mathematical operators as shown below.

The sum $A+B$ of two $m \times n$ matrices A and B is calculated as shown below.

$$C_{i,j} = (A+B)_{i,j} = A_{i,j} + B_{i,j} \quad \text{where} \quad 1 \le i \le m \quad \text{and} \quad 1 \le j \le n.$$

```
public static RMatrix operator +(RMatrix m)
{
    return m;
}

public static RMatrix operator +(RMatrix m1, RMatrix m2)
{
    if (!RMatrix.CompareDimension(m1, m2))
    {
        throw new Exception("The dimensions of two matrices must
            be the same!");
    }
    RMatrix result = new RMatrix(m1.GetnRows, m1.GetnCols);
    for (int i = 0; i < m1.GetnRows; i++)
    {
        for (int j = 0; j < m1.GetnCols; j++)
        {
            result[i, j] = m1[i, j] + m2[i, j];
        }
    }
    return result;
}
```

Similarly, the difference $A-B$ of two $m \times n$ matrices A and B is calculated as follows.

$$C_{i,j} = (A-B)_{i,j} = A_{i,j} - B_{i,j} \quad \text{where} \quad 1 \le i \le m \quad \text{and} \quad 1 \le j \le n.$$

```
public static RMatrix operator -(RMatrix m)
{
    for (int i = 0; i < m.GetnRows; i++)
    {
        for (int j = 0; j < m.GetnCols; j++)
        {
            m[i, j] = -m[i, j];
        }
    }
    return m;
}

public static RMatrix operator -(RMatrix m1, RMatrix m2)
{
    if (!RMatrix.CompareDimension(m1, m2))
    {
        throw new Exception("The dimensions of two matrices must
            be the same!");
    }
    RMatrix result = new RMatrix(m1.GetnRows, m1.GetnCols);
    for (int i = 0; i < m1.GetnRows; i++)
    {
        for (int j = 0; j < m1.GetnCols; j++)
        {
            result[i, j] = m1[i, j] - m2[i, j];
        }
    }
    return result;
}
```

Matrix multiplication comes in two forms. The scalar multiplication cA of a matrix A and a number c is given by multiplying every entry of A by c:

$$cA = cA_{i,j} = (cA)_{i,j}$$

Although matrix division is not defined, you can still multiply a matrix by a fractional scalar, such as $1/c$, that can affect and contract every entry in the matrix.

$$(1/c)A = (1/c)A_{i,j} = ((1/c)A)_{i,j}$$

```
public static RMatrix operator *(RMatrix m, double d)
{
    RMatrix result = new RMatrix(m.GetnRows, m.GetnCols);
    for (int i = 0; i < m.GetnRows; i++)
    {
        for (int j = 0; j < m.GetnCols; j++)
        {
            result[i, j] = m[i, j] * d;
        }
    }
    return result;
}
```

```
public static RMatrix operator *(double d, RMatrix m)
{
    RMatrix result = new RMatrix(m.GetnRows, m.GetnCols);
    for (int i = 0; i < m.GetnRows; i++)
    {
        for (int j = 0; j < m.GetnCols; j++)
        {
            result[i, j] = m[i, j] * d;
        }
    }
    return result;
}

public static RMatrix operator /(RMatrix m, double d)
{
    RMatrix result = new RMatrix(m.GetnRows, m.GetnCols);
    for (int i = 0; i < m.GetnRows; i++)
    {
        for (int j = 0; j < m.GetnCols; j++)
        {
            result[i, j] = m[i, j] / d;
        }
    }
    return result;
}

public static RMatrix operator /(double d, RMatrix m)
{
    RMatrix result = new RMatrix(m.GetnRows, m.GetnCols);
    for (int i = 0; i < m.GetnRows; i++)
    {
        for (int j = 0; j < m.GetnCols; j++)
        {
            result[i, j] = m[i, j] / d;
        }
    }
    return result;
}
```

The second form of matrix multiplication is from multiplying two matrices together. The matrix product of two matrices A and B is just another matrix, say C, where $C = AB$ and whose entries are formally defined as follows. For $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times p}$ then $(AB) \in \mathbb{R}^{m\times p}$ where

$$(C)_{i,j} = (AB)_{i,j} = \sum_{k=1}^{n} A_{i,k}B_{k,j}$$

for each pair i and j with $1 \le i \le m$ and $1 \le j \le p$. Because of this definition, matrix multiplication is not commutative which means $AB \neq BA$. Matrix multiplication can be implemented as follows.

```
public static RMatrix operator *(RMatrix m1, RMatrix m2)
{
    if (m1.GetnCols != m2.GetnRows)
    {
        throw new Exception("The numbers of columns of the" +
         " first matrix must be equal to the number of " +
         " rows of the second matrix!");
    }
    double tmp;
    RMatrix result = new RMatrix(m1.GetnRows, m2.GetnCols);
    for (int i = 0; i < m1.GetnRows; i++)
    {
        for (int j = 0; j < m2.GetnCols; j++)
        {
            tmp = result[i, j];
            for (int k = 0; k < result.GetnRows; k++)
            {
                tmp += m1[i, k] * m2[k, j];
            }
            result[i, j] = tmp;
        }
    }
    return result;
}
```

Using matrix multiplication and treating vectors as $n \times 1$ matrices, the dot product can also be written as:

$$\mathrm{A} \cdot \mathrm{B} = \mathrm{A}^T\mathrm{B}$$

where A^T denotes the *transpose* of the matrix A which is created by any one of the following equivalent actions:

- Write the rows of A as the columns of A^T
- Write the columns of A as the rows of A^T
- Reflect A by its main diagonal (which starts from the top left) to obtain A^T

In this specific case, since A is a column matrix, the transpose of A is a row matrix or row vector ($1 \times n$ matrix). More formally, the transpose of an $m \times n$ matrix A is the $n \times m$ matrix

$$\mathrm{A}^{\mathrm{T}}_{ij} = \mathrm{A}_{ji} \quad \text{for} \quad 1 \le i \le n, \; 1 \le j \le m$$

```
public RMatrix GetTranspose()
{
    RMatrix m = this;
    m.Transpose();
    return m;
}

public void Transpose()
{
    RMatrix m = new RMatrix(nCols, nRows);
    for (int i = 0; i < nRows; i++)
```

```
        {
            for (int j = 0; j < nCols; j++)
            {
                m[j, i] = matrix[i, j];
            }
        }
        this = m;
    }
```

The *trace* of an $n \times n$ square matrix A is defined to be the sum of the elements on the main diagonal (the diagonal from the upper left to the lower right) of A so that it can be expressed mathematically as

$$\mathrm{tr}(A) = a_{11} + a_{22} + \cdots + a_{nn} = \sum_{i=1}^{n} a_{ii}$$

where $a_{i,j}$ represents the entry on the i-th row and j-th column of A. Equivalently, the trace of a matrix is the sum of its eigenvalues, thus making it an invariant with respect to a change of basis. The following implementation can be used to define the trace of a linear operator in general.

```
    public double GetTrace()
    {
        double sum_of_diag = 0.0;
        for (int i = 0; i < nRows; i++)
        {
            if (i < nCols)
                sum_of_diag += matrix[i, i];
        }
        return sum_of_diag;
    }
```

In more advanced matrix calculations there are occasions where it may be necessary to extract a vector from a row or a column of a matrix. Other situations may require a swap of two rows or two columns of a matrix. The following methods demonstrate how such operations may be carried out along with a few miscellaneous matrix manipulation utilities that are self-explanatory.

```
    public bool IsSquared()
    {
        if (nRows == nCols)
            return true;
        else
            return false;
    }

    public static bool CompareDimension(RMatrix m1, RMatrix m2)
    {
        if (m1.GetnRows == m2.GetnRows && m1.GetnCols == m2.GetnCols)
            return true;
        else
            return false;
    }
    public RVector GetRowVector(int m)
    {
        if (m < 0 || m > nRows)
        {
            throw new Exception("m-th row is out of range!");
        }
        RVector RowVector = new RVector(nCols);
        for (int i = 0; i < nCols; i++)
        {
            RowVector[i] = matrix[m, i];
        }
        return RowVector;
    }

    public RVector GetColVector(int n)
    {
        if (n < 0 || n > nCols)
        {
            throw new Exception("n-th col is out of range!");
        }
        RVector ColVector = new RVector(nRows);
        for (int i = 0; i < nRows; i++)
        {
            ColVector[i] = matrix[i, n];
        }
        return ColVector;
    }

    public RMatrix ReplaceRow(RVector vec, int m)
    {
        if (m < 0 || m > nRows)
        {
            throw new Exception("m-th row is out of range!");
        }
        if (vec.GetVectorSize != nCols)
        {
            throw new Exception("Vector ndim is out of range!");
        }
        for (int i = 0; i < nCols; i++)
        {
            matrix[m, i] = vec[i];
        }
        return new RMatrix(matrix);
    }

    public RMatrix ReplaceCol(RVector vec, int n)
    {
        if (n < 0 || n > nCols)
        {
            throw new Exception("n-th col is out of range!");
        }
        if (vec.GetVectorSize != nRows)
        {
            throw new Exception("Vector ndim is out of range!");
        }
        for (int i = 0; i < nRows; i++)
        {
            matrix[i, n] = vec[i];
        }
        return new RMatrix(matrix);
    }

    public RMatrix SwapMatrixRow(int m, int n)
    {
        double temp = 0.0;
        for (int i = 0; i < nCols; i++)
        {
            temp = matrix[m, i];
            matrix[m, i] = matrix[n, i];
            matrix[n, i] = temp;
        }
        return new RMatrix(matrix);
    }

    public RMatrix SwapMatrixColumn(int m, int n)
    {
        double temp = 0.0;
        for (int i = 0; i < nRows; i++)
        {
            temp = matrix[i, m];
            matrix[i, m] = matrix[i, n];
            matrix[i, n] = temp;
        }
        return new RMatrix(matrix);
    }
```

In linear algebra, linear transformations can be represented by matrices. If T is a linear transformation mapping $\mathbb{R}^n$ to $\mathbb{R}^m$ and $\vec{x}$ is a column vector with n entries, then

$$T(\vec{x}) = \mathrm{A}\vec{x}$$

for some $m \times n$ matrix A, is called the transformation matrix of T. Matrices allow arbitrary linear transformations to be represented in a consistent format, suitable for computation. For example, if one has a linear transformation $T(x)$ in functional form, it is easy to determine the transformation matrix A by simply transforming each of the vectors of the standard basis by T and then inserting the results into the columns of matrix A. In other words,

$$\mathrm{A} = \begin{bmatrix} T(\hat{e}_1) \; T(\vec{e}_2) \; \cdots \; T(\vec{e}_n) \end{bmatrix}$$

Most common geometric transformations that keep the origin fixed are linear, including rotation, scaling, shearing, reflection, and orthogonal projection. If an affine transformation is not a pure translation it keeps some point fixed, and that point can be chosen as origin to make the transformation linear. For example, consider a rotation in a two-dimensional Euclidean plane by an angle θ counterclockwise about the origin. The functional form is $x' = x\cos\theta - y\sin\theta$ and $y' = x\sin\theta + y\cos\theta$. Written in matrix form, these equations become:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

The following code illustrates how to calculate the transform matrix in C#

```
public static RVector Transform(RMatrix mat, RVector vec)
{
    RVector result = new RVector(vec.GetVectorSize);
    if (!mat.IsSquared())
    {
       throw new Exception("The matrix must be squared!");
    }
    if (mat.GetnCols != vec.GetVectorSize)
    {
       throw new Exception("The ndim of the vector must be equal"
           + " to the number of cols of the matrix!");
    }
    for (int i = 0; i < mat.GetnRows; i++)
    {
       result[i] = 0.0;
       for (int j = 0; j < mat.GetnCols; j++)
       {
          result[i] += mat[i, j] * vec[j];
       }
    }
    return result;
}

public static RVector Transform(RVector vec, RMatrix mat)
{
    RVector result = new RVector(vec.GetVectorSize);
    if (!mat.IsSquared())
    {
        throw new Exception("The matrix must be squared!");
    }
    if (mat.GetnRows != vec.GetVectorSize)
    {
      throw new Exception("The ndim of the vector must be equal"
          + " to the number of rows of the matrix!");
    }
    for (int i = 0; i < mat.GetnRows; i++)
    {
       result[i] = 0.0;
       for (int j = 0; j < mat.GetnCols; j++)
       {
          result[i] += vec[j] * mat[j, i];
       }
    }
    return result;
}

public static RMatrix Transform(RVector v1, RVector v2)
{
 if (v1.GetVectorSize != v2.GetVectorSize)
 {
   throw new Exception("The vectors must have the same ndim!");
 }
 RMatrix result = new RMatrix(v1.GetVectorSize,v1.GetVectorSize);
 for (int i = 0; i < v1.GetVectorSize; i++)
 {
   for (int j = 0; j < v1.GetVectorSize; j++)
   {
     result[j, i] = v1[i] * v2[j];
   }
 }
 return result;
}
```

The determinant is an algebraic operation that transforms a square matrix A into a scalar following a specific procedure that is described in more detail below. Determinants are defined only for square matrices and can be generally expressed as

$$\det A = \begin{vmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{vmatrix}$$

A minor of a matrix A is the determinant of some smaller square matrix, cut down from A by removing one or more of its rows and columns. More specifically, the M_{ij} minor of an $n \times n$ square matrix A is defined as the determinant of the $(n-1) \times (n-1)$ matrix formed by removing from A its i-th row and j-th column. A minor that is formed by removing only one row and column from a square matrix A, M_{ij}, is called a first minor. When two rows and columns are removed, it is called a second minor, and so on. The (i, j)-th cofactor C_{ij} of a square matrix A is just $(-1)^{i+j}$ times the corresponding minor M_{ij}

$$C_{ij} = (-1)^{i+j} M_{ij}$$

The cofactor matrix of A, or matrix of A cofactors, typically denoted as C, is defined as the $n \times n$ matrix whose (i, j) entry is the (i, j) cofactor of A. The transpose of C is called the adjoint of A. The adjoint matrix is therefore just the transpose matrix of cofactors as shown below

$$\mathrm{A}^{-1} = \frac{1}{|\mathrm{A}|}(\mathrm{C}_{ij})^{\mathrm{T}} = \frac{1}{|\mathrm{A}|}(\mathrm{C}_{ji}) = \frac{1}{|\mathrm{A}|}\begin{pmatrix} \mathrm{C}_{11} & \mathrm{C}_{21} & \cdots & \mathrm{C}_{n1} \\ \mathrm{C}_{12} & \mathrm{C}_{22} & \cdots & \mathrm{C}_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{C}_{1n} & \mathrm{C}_{2n} & \cdots & \mathrm{C}_{nn} \end{pmatrix}$$

where $|A|$ is the determinant of A, C_{ij} is the matrix cofactor, and A^T represents the matrix transpose.

Adjoint matrices are used to compute the inverse of the square matrices. An $n \times n$ square matrix A is called invertible or non-singular if there exists an $n \times n$ matrix B such that

$$\mathrm{AB} = \mathrm{BA} = \mathrm{I}_n$$

where I_n denotes the $n \times n$ identity matrix and the multiplication used is ordinary matrix multiplication. If this is the case, then the matrix B is uniquely determined by

A and is called the inverse of A, denoted by A^{-1} The determinant of a matrix A can then be expressed very compactly in terms of the cofactor C_{ij} of matrix A as

$$|A| = \sum_{i=1}^{k} a_{ij} C_{ij}$$

with no implied summation over j and where C_{ij} is the cofactor of a_{ij}. This process is called determinant expansion by minors.

Let A be an $n \times n$ matrix. Then $\mathrm{Adj}(A)A = \det(A)I$ where $Adj(A)$ denotes the adjoint of A, $det(A)$ is the determinant, and I is the identity matrix. If $det(A)$ is invertible in R, then the inverse matrix of A is

$$A^{-1} = \frac{1}{\det(A)} \mathrm{Adj}(A).$$

As a simple example, the equation for calculating the inverse matrix of a general 2×2 matrix is given by

$$\mathrm{A}^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

The code for performing all these operations on matrices and determinants is listed below.

```
public static double Determinant(RMatrix mat)
{
  double result = 0.0;
  if (!mat.IsSquared())
  {
    throw new Exception("The matrix must be squared!");
  }
  if (mat.GetnRows == 1)
    result = mat[0, 0];
  else
  {
    for (int i = 0; i < mat.GetnRows; i++)
    {
      result += Math.Pow(-1, i) * mat[0, i] *
                    Determinant(RMatrix.Minor(mat, 0, i));
    }
  }
  return result;
}

public static RMatrix Minor(RMatrix mat, int row, int col)
{
  RMatrix mm = new RMatrix(mat.GetnRows-1,mat.GetnCols-1);
  int ii = 0, jj = 0;
  for (int i = 0; i < mat.GetnRows; i++)
  {
    if (i == row)
      continue;
```

```
        jj = 0;
        for (int j = 0; j < mat.GetnCols; j++)
        {
          if (j == col)
            continue;
          mm[ii, jj] = mat[i, j];
          jj++;
        }
        ii++;
      }
      return mm;
    }

    public static RMatrix Adjoint(RMatrix mat)
    {
      if (!mat.IsSquared())
      {
        throw new Exception("The matrix must be squared!");
      }
      RMatrix ma = new RMatrix(mat.GetnRows, mat.GetnCols);
      for (int i = 0; i < mat.GetnRows; i++)
      {
        for (int j = 0; j < mat.GetnCols; j++)
        {
          ma[i,j]=Math.Pow(-1,i+j)*(Determinant(Minor(mat,i,j)));
        }
      }
      return ma.GetTranspose();
    }

    public static RMatrix Inverse(RMatrix mat)
    {
      if (Determinant(mat) == 0)
      {
        throw new Exception("Cannot inverse a matrix with a zero
            determinant!");
      }
      return (Adjoint(mat) / Determinant(mat));
    }
}
```

4

Complex Numbers

4.1 Introduction

Complex numbers are often used not just in the field of pure or applied mathematics but also in a variety of scientific and engineering disciplines. First discovered in the 16th century by the Italian mathematicians Girolanmo Cardano and Niccolo Tartaglia, complex numbers were later refined by Rafael Bombelli who developed the formal rules for their addition, subtraction, multiplication and division. In the 17th century, the French mathematician Rene Descartes introduced the term *imaginary* to describe complex numbers. However, this terminology is just historical and perhaps even misleading. There is nothing mystical or weird about complex numbers, and the so-called imaginary part is just an ordinary real number with specific contextual meaning. In the 18th century, the famous Swiss mathematician Leonhard Euler introduced both the special symbol i to represent $\sqrt{-1}$ and also the famous expression $e^{i\theta}$ that now bears his name. However, the existence of complex numbers was not fully accepted until Caspar Wessel and Jean-Robert Argand developed a geometrical interpretation for them around the year 1799. After languishing for years in obscurity, most of these ideas were later rediscovered and popularized by the distinguished German mathematician Carl Friedrich Gauss in the 19th century. Gauss not only made important additional contributions to the field of complex numbers but was also the first one to use the term *complex* to describe these kinds of numbers that include the $\sqrt{-1}$. Afterwards, the theory of complex numbers advanced rapidly and is widely used in several mathematical, scientific and engineering fields today. Unfortunately, the present version of C# does not yet support an intrinsic complex number data type along with its own set of internal library routines. Therefore, this chapter was written to provide the reader with a comprehensive collection of numerical routines in C# for working with complex numbers.

4.2 Fundamental Concepts

Complex numbers are usually represented by the symbol z and can be expressed in any one of the following four formats [18]

- Algebraically, as $z = x + iy$ where both the real part $x = Re(z)$ and the imaginary part $y = Im(z)$ are real numbers and the imaginary unit $i = \sqrt{-1}$.
- Trigonometrically, in polar form as $z = r\cos\theta + i\,r\sin\theta$ where, as displayed in Figure 4.1, (x, y) and (r, θ) are related by

$$x = r\cos\theta \qquad y = r\cos\theta \qquad r = \sqrt{x^2 + y^2} \qquad \text{and} \qquad \theta = \arctan\left(\frac{y}{x}\right)$$

- Exponentially, as the exponential function $z = re^{i\theta}$ with a real radius r and a real phase angle θ as depicted in Figure 4.1.
- Graphically, as a coordinate (x, y) in a modified Cartesian plane with the real part of the complex number represented by a displacement along the x-axis and the imaginary part by a displacement along the y-axis as illustrated in Figure 4.1.

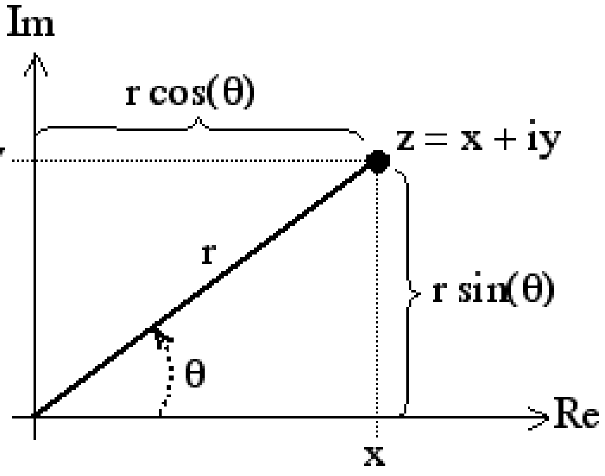


FIGURE 4.1
Geometric Representation of Complex Numbers

The argument of z, denoted by $arg(z)$, is just another name used for representing the angle θ which is multi-valued with a period of 2π. Thus, if θ is one value of $arg(z)$, the other values are given by arg(z) = $\theta + 2\pi n$, where n is any integer. If $r = 0$, then θ can be set to any real value. However, if $r \neq 0$, then in order to get a unique value, θ must be limited to an interval of size 2π which is traditionally chosen as being either the interval $[0, 2\pi)$ or $(-\pi, \pi]$ depending on the convention used for taking the branch cut. As a result, one must keep track of the signs for both variables x and y in order to correctly calculate the location and value of θ. Fortunately, many modern programming languages, such as C#, avoid having to directly handle this problem by simply using the `atan2(y/x)` function, which has separate arguments for both the x and the y values. As a result, the output from the function `atan2(y/x)` consists of a unique value within the interval $(-\pi, \pi]$ and will be henceforth referred to as the *principal* value of θ.

The absolute value or *modulus* of z is defined to be $|z| = r = \sqrt{x^2 + y^2}$. Graphically, $|z|$ represents the distance from the origin to the point (x, y) in the complex plane.

The complex conjugate of a complex number is found by simply changing the sign of the imaginary part. Thus, the conjugate of the complex number $z = x + iy = re^{i\theta}$ is expressed by $z^* = \bar{z} = x - iy = re^{-i\theta}$.

Two complex numbers are equal if and only if their real parts are equal and their imaginary parts are equal. In other words, if the two complex numbers are written as $x_1 + iy_1$ and $x_2 + iy_2$ with x_1, y_1, x_2, y_2 all reals, then they are equal if and only if $x_1 = x_2$ *and* $y_1 = y_2$.

4.3 Complex Number Arithmetic

Complex numbers are said to form a field, known as the complex number field, which is denoted by $\mathbb{C}$. A field is an algebraic structure with addition, subtraction, multiplication, and division operations that satisfy certain algebraic laws. In particular, this means that complex numbers have

- An additive identity ("zero"), $0 + i0$.
- A multiplicative identity ("one"), $1 + i0$.
- An additive inverse for every complex number. The additive inverse of $x + iy$, for example, is $-x - iy$.
- A multiplicative inverse (reciprocal) for every nonzero complex number. The reciprocal of $x + iy$, for example, is $\frac{1}{x+iy} = \frac{1}{x+iy} * \frac{x-iy}{x-iy} = \frac{x}{x^2+y^2} + \frac{-iy}{x^2+y^2}$
- Addition, subtraction, multiplication and division operations defined by formally applying the associative, commutative and distributive laws of algebra, together with the equation $i^2 = -1$. Thus,

$$z_1 \pm z_2 = (x_1 + iy_1) \pm (x_2 + iy_2) = (x_1 \pm x_2) + i(y_1 \pm y_2)$$
$$z_1 * z_2 = (x_1 + iy_1) * (x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1)$$
$$\frac{z_1}{z_2} = \frac{x_1 + iy_1}{x_2 + iy_2} = \frac{(x_1 + iy_1)(x_2 - iy_2)}{(x_2 + iy_2)(x_2 - iy_2)} = \frac{x_1x_2 + y_1y_2}{x_2^2 + y_2^2} + i(\frac{-x_1y_2 + x_2y_1}{x_2^2 + y_2^2})$$

Although the basic formulas for the addition, subtraction, multiplication and division of complex numbers are mathematically correct as they have been written and can be immediately applied to do numerical calculations, in actual practice there are some additional important computational issues that merit some extra attention. In some computers, for example, multiplication operations have been found to run somewhat slower than those involving addition. As a result, Press et al. [22] have proposed a slight rearrangement of the original multiplication formula in order to allegedly make it run faster since it contains fewer multiplication steps

$$z_1 * z_2 = (x_1 + iy_1) * (x_2 + iy_2) = (x_1x_2 - y_1y_2) + i[(x_1 + y_1)(x_2 + y_2) - x_1x_2 - y_1y_2]$$

Midy and Yakolev [23], on the other hand, have pointed out that the existing formulas for numerically calculating elementary functions of a complex number on a computer are not always as reliable as one might expect particularly when handling very small or very large numbers. For example, forgetting or ignoring the existence of potential numeric *underflows* and/or *overflows* can sometimes lead to misleading or even erroneous results. Consequently, some insightful numerical algorithms have been proposed over the years to help prevent or at least minimize the chances for such undesired problems from occurring during certain complex number operations. Of these, perhaps the best known and most widely used algorithm is the one for doing complex number division as proposed by Smith [24]

$$\frac{z_1}{z_2} = \frac{x_1 + iy_1}{x_2 + iy_2} = \begin{cases} \dfrac{[x_1 + y_1(y_2/x_2)] + i[y_1 - x_1(y_2/x_2)]}{x_2 + y_2(y_2/x_2)} & \text{if } |x_2| \geq |y_2| \\ \dfrac{[x_1(x_2/y_2) + y_1)] + i[y_1(x_2/y_2) - x_1]}{x_2(x_2/y_2) + y_2} & \text{if } |x_2| < |y_2| \end{cases}$$

Unfortunately, small flaws involving unwanted underflows and/or overflows have also reportedly been found with this algorithm. Although Stewart [25] has attempted to correct this problem, Midy and Yakolev [23] later pointed out that this newer and supposedly improved algorithm is not completely free from generating overflows either and, in addition, non-renormalized underflows were not successfully addressed. Nevertheless, applying the same general ideas of this scaling method to the formula for the modulus of a complex number z will result in the following equations

$$|z| = |x + iy| = r = \sqrt{x^2 + y^2} = \begin{cases} |x|\sqrt{1 + (y/x)^2} & \text{if } |x| \geq |y| \\ |y|\sqrt{1 + (x/y)^2} & \text{if } |x| < |y| \end{cases}$$

Unfortunately, most if not all of these algorithms that have been devised to supposedly handle potential numerical overflow and underflow problems in complex number operations have eventually been found to be somewhat flawed in one way or another. These problems arise primarily because computers are merely machines and as such they have a limited capacity for handling real numbers whose actual range of values is infinite. For example, irrational numbers like π, can only be approximated by a computer to a certain number of significant figures. In addition, as we have seen in Chapter 1, some precision is also lost in doing numerical calculations because of the way numbers are stored in a computer. Although our numbering system is base 10, computers use the binary system of base 2 to store and manipulate numbers. This feature can occasionally lead to a loss of some precision during the many conversions that ultimately may have to be done back and forth between the two bases during the course of running an actual application. One should therefore always proceed with great care and exercise caution when applying these and other algorithms near extreme points that may generate numerical overflow and/or underflow. Aside from modifying numerical algorithms to improve their ability to handle values near extreme or critical points as illustrated here, another perhaps riskier and less desirable way to deal with this problem is to write code to catch and properly handle all exceptions thrown by the compiler.

4.4 Elementary Functions of a Complex Number

Using the reference data published by Abramowitz and Stegun [19] and Thompson [26], this section covers the standard elementary transcendental functions for complex numbers including the exponential, trigonometric and hyperbolic functions along with their corresponding inverses. As before, the focus will be on developing a set of practical computational tools in C# for use in numerical applications.

4.4.1 Exponentials

By definition, the general exponential function with a fixed real number base $b > 1$ and a real number power x is the function expressed by the formula $f(x) = b^x$. If the base equals the Euler number e, then the exponential function is called the *natural* exponential function and is expressed by $f(x) = e^x = exp(x)$. The basic rules used in manipulating exponential functions of real numbers are well known and are given by $b^{x+y} = b^x b^y$, $b^{x-y} = b^x/b^y$, $b^0 = 1$, $b^{xy} = (b^x)^y$, and $b^{-x} = 1/(b^x)$.

The exponential function can also be defined on the complex plane in various different, but nonetheless, equivalent forms. Some of these definitions mirror the formulas for the real-valued exponential function and even retain a certain number of important properties, such as $e^{z+w} = e^z e^w$, where z and w are complex numbers. For practical numerical computational purposes, the exponential function, e^z, where z is a complex variable, can therefore be written as

$$\begin{aligned} e^z = e^{x+iy} &= e^x e^{iy} = e^x(\cos y + i \sin y) \\ &= e^x \cos y + i e^x \sin y \end{aligned}$$

4.4.2 Logarithms

In general, the inverse of any exponential function is a logarithmic function. The logarithm of a number to a given positive real number base is the power or exponent to which the base must be raised in order to produce the number. By definition, the logarithm of x to a base b is written as $\log_b(x)$ or, if the base is implicit, as $\log(x)$. Hence, for a number x, a base b and an exponent y, if $x = b^y$ then $y = \log_b(x)$.

The logarithm function can also be extended to include complex numbers in which case it is simply the inverse of the corresponding complex exponential function. This means that for the natural base e, the logarithm of a complex number z is a complex number w such that if $z = e^w$ then $w = \log z$. By writing z in polar form, $z = re^{i\theta}$, and taking the natural logarithm of both sides we obtain a general expression for the complex logarithm of a complex number z as

$$\log z = \log r + i(\theta + 2\pi n) \quad \text{where} \quad z \neq 0 \quad \text{and} \quad n = \text{any integer.}$$

Strictly speaking, for a function to have an inverse, it must map distinct arguments with distinct values. Note, however, that the polar angle θ in the logarithmic expression above is ambiguous since any integral multiple of 2π could be added to θ without changing the value of $\log z$. Therefore, $\log z$ is both periodic and multi-valued and so it does not have an inverse function in the standard sense. Likewise, since $e^{w+2\pi ni} = e^w$ for any integer n, the complex exponential function e^w is also both periodic and multi-valued. Therefore, e^w also does not have an inverse function in the standard sense. Fortunately, there are two ways around this problem. One approach is to view the logarithm as a function whose domain is not a region in the complex plane, but a Riemann surface that covers the punctured complex plane in an infinite-to-1 way. The other solution is to restrict the domain of the exponential function to a region that does not contain any two numbers differing by an integer multiple of 2π. This approach naturally leads to the concept of branches. By definition, a branch cut is a curve in the complex plane across which an analytic multi-valued function is discontinuous. In order to work with single-valued complex functions, it is customary to construct branch cuts in the complex plane where there is a well-defined branch of the function in question. The principal branch is a function which selects one *branch*, or *slice*, of a multi-valued function from which one obtains single, unique values which are more commonly known as *principal* values. For complex logarithms, a branch cut, usually along the negative real axis, can limit the imaginary part so it lies between $-\pi$ and π. The principal branch of the complex logarithmic function is often expressed with a capital letter $\mathrm{Log}(z)$ where $n = 0$ and is given by

$$\mathrm{Log}(z) = \log r + i\,\theta \quad \text{where} \quad z = r\,e^{i\,\theta} \quad \text{and} \quad -\pi < \theta \leq \pi$$

For practical numerical computational purposes, the *principal* logarithm function of a complex number $z = x + iy$ can be written as

$$\begin{aligned}\mathrm{Log}(z) &= ln(x+iy) = ln\,r + i\theta \\ &= (\frac{1}{2})\,ln(x^2+y^2) + i\arctan(\frac{y}{x})\end{aligned}$$

4.4.3 Powers and Roots

Raising a complex number z to some integer power n means multiplying z by itself repeatedly for a total of n times. The n-th power of a complex number z follows directly from De Moivre's Theorem and is given by

$$z^n = (re^{i\theta})^n = r^n e^{in\theta} = r^n \cos(n\theta) + ir^n sin(n\theta)$$

De Moivre's Theorem also holds true for the ratio of two integers, say m and n. Thus,

$$z^{m/n} = (re^{i\theta})^{m/n} = r^{m/n} e^{im\theta/n}$$

If $m = 1$, the formula above can also be used to extract the n-th root of z. More formally, the n-th root of a complex number $a \in \mathbb{C}$, satisfies the equation $z^n = a$. Since $e^{i\theta}$ is periodic with a period of 2π then $e^{i\theta} = e^{i(\theta+2\pi k)}$ and so the general formula for the n-th root of z can be written as

$$z^{\frac{1}{n}} = (re^{i\theta})^{\frac{1}{n}} = r^{\frac{1}{n}} \exp(\frac{i(\theta + 2\pi k)}{n}) \quad \text{where} \quad k \in \{0, 1, 2, \ldots n-1\}$$

In addition, if m and n are both integers, then the usual properties for handling real valued bases and exponents can also be extended to include the complex domain. For example, $z^{m+n} = z^m z^n$, $z^{m-n} = z^m/z^n$, $(z^m)^n = z^{mn}$, and $z^{-n} = 1/(z^n)$.

On the other hand, raising a complex number z to the power of some other complex number w is an ambiguous process because complex powers give rise to multi-valued logarithmic functions. For example, writing z^w as an exponential to the natural base e gives the following expression

$$z^w = e^{w \log z} = \exp[w(\log r + i(\theta + 2\pi n))] \text{ where } z \neq 0 \text{ and } n = \text{any integer}$$

As before, a branch cut can be used to select a specific value. The most common branch cut or principal value chosen corresponds to θ being confined to the interval $(-\pi, \pi]$. Then in order to compute a numerical value for $z^w = u + iv$ we can express $z = x_1 + iy_1$ and $w = x_2 + iy_2$ and show that

$$u = ({x_1}^2 + {y_1}^2)^{x_2/2} \exp(-y_2 \arctan(\frac{y_1}{x_1})) \cos(\frac{y_2}{2} \ln({x_1}^2 + {y_1}^2) + x_2 \arctan(\frac{y_1}{x_1}))$$

$$v = ({x_1}^2 + {y_1}^2)^{x_2/2} \exp(-y_2 \arctan(\frac{y_1}{x_1})) \sin(\frac{y_2}{2} \ln({x_1}^2 + {y_1}^2) + x_2 \arctan(\frac{y_1}{x_1})$$

Unfortunately, some earlier identities used to manipulate powers and logarithms for positive real numbers will fail when raising a complex number to the power of some other complex number no matter how the complex power and complex logarithm are defined. For example, the formula $\log(x)^b = b \log x$ holds whenever x is a positive real number and b is a real number. But for the principal branch of the complex logarithm function, one obtains the following inequality

$$i\pi = \log(-1) = \log((-i)^2) \neq 2\log(-i) = 2(-i\pi/2) = -i\pi$$

regardless of which branch of the logarithm is used. Similarly, it can be shown that many other exponential properties for real numbers will fail when carelessly applied to complex numbers. For example, if z, w and u are all complex numbers then

$$(e^z)^w \neq e^{(zw)} \quad \text{and} \quad (zw)^u \neq z^u w^u \quad \text{and} \quad (z/w)^u \neq z^u/w^u$$

Although the square root of a complex number can be easily calculated using one of the formulas described earlier, Press et al. [22] have published another formula that is allegedly more computationally efficient and, in addition, is also better able to handle potential undesired underflow and overflow problems.

$$\sqrt{z} = \sqrt{x+iy} = \begin{cases} 0 & \text{if } w = 0 \\ w + i(\dfrac{y}{2w}) & \text{if } w \neq 0,\ x \geq 0 \\ \dfrac{|y|}{2w} + iw & \text{if } w \neq 0,\ x < 0,\ y \geq 0 \\ \dfrac{|y|}{2w} - iw & \text{if } w \neq 0,\ x < 0,\ y < 0 \end{cases}$$

where

$$w = \begin{cases} 0 & \text{if } x = y = 0 \\ \sqrt{|x|}\sqrt{\dfrac{1+\sqrt{1+(y/x)^2}}{2}} & \text{if } |x| \geq |y| \\ \sqrt{|y|}\sqrt{\dfrac{|x/y|+\sqrt{1+(x/y)^2}}{2}} & \text{if } |x| < |y| \end{cases}$$

4.4.4 Trigonometric and Hyperbolic Functions

A computationally feasible version of the complex and hyperbolic functions can be easily found by applying Euler's formula to a general complex number z. Using $e^{iz} = \cos z + i \sin z$, along with its corresponding conjugate $e^{-iz} = \cos z - i \sin z$, one can solve for $\cos z$ and $\sin z$ to obtain

$$\cos z = \frac{e^{iz} + e^{-iz}}{2} \quad \text{and} \quad \sin z = \frac{e^{iz} - e^{-iz}}{2i}$$

The hyperbolic cosine and hyperbolic sine functions of a complex number z are analogously given by

$$\cosh z = \frac{e^{z} + e^{-z}}{2} \quad \text{and} \quad \sinh z = \frac{e^{z} - e^{-z}}{2}$$

Using these results, together with the following trigonometric addition formulas,

$$\sin(x+y) = \sin x \cos y + \cos x \sin y$$
$$\cos(x+y) = \cos x \cos y - \sin x \sin y$$

one arrives at the following numerically computable expression for the two most fundamental trigonometric functions of a general complex number z

$$\begin{aligned}\sin z &= \sin(x+iy)\\ &= \sin(x)\cos(iy)+\cos(x)\sin(iy)\\ &= \sin(x)\cosh(y)+i\cos(x)\sinh(y)\\ &= -i\sinh(iz)\\ \cos z &= \cos(x+iy)\\ &= \cos(x)\cos(iy)-\sin(x)\sin(iy)\\ &= \cos(x)\cosh(y)-i\sin(x)\sinh(y)\\ &= \cosh(iz)\end{aligned}$$

The remaining complex trigonometric functions then immediately follow as shown below.

$$\begin{aligned}\tan z &= \frac{\sin z}{\cos z} = \frac{\sin(2x)+i\sinh(2y)}{\cos(2x)+\cosh(2y)}\\ \cot z &= \frac{1}{\tan z} = \frac{\sin(2x)-i\sinh(2y)}{\cosh(2y)-\cos(2x)}\\ \sec z &= \frac{1}{\cos z} = \frac{\cos(x)\cosh(y)+i\sin(x)\sinh(y)}{\cos^2(x)+\sinh^2(y)}\\ \csc z &= \frac{1}{\sin z} = \frac{\sin(x)\cosh(y)-i\cos(x)\sinh(y)}{\sin^2(x)+\sinh^2(y)}\end{aligned}$$

Similarly, using the above results along with the following hyperbolic addition formulas

$$\begin{aligned}\sinh(x+y) &= \sinh x\cosh y+\cosh x\sinh y\\ \cosh(x+y) &= \cosh x\cosh y+\sinh x\sinh y\end{aligned}$$

one arrives at the following numerically computable expression for the two most fundamental hyperbolic functions of a general complex number z

$$\begin{aligned}\sinh z &= \sinh(x+iy)\\ &= \sinh(x)\cosh(iy)+\cosh(x)\sinh(iy)\\ &= \sinh(x)\cos(y)+i\cosh(x)\sin(y)\\ &= -i\sin(iz)\\ \cosh z &= \cosh(x+iy)\\ &= \cosh(x)\cosh(iy)+\sinh(x)\sinh(iy)\\ &= \cosh(x)\cos(y)+i\sinh(x)\sin(y)\\ &= \cos(iz)\end{aligned}$$

Likewise, the remaining complex hyperbolic functions then immediately follow as shown below.

$$
\begin{aligned}
\tanh z &= \frac{\sinh z}{\cosh z} = \frac{\sinh 2x + i \sin 2y}{\cosh 2x + \cos 2y} = -i \tan(iz) \\
\coth z &= \frac{1}{\tanh z} = \frac{\sinh(2x) - i \sin(2y)}{\cosh(2x) - \cos(2y)} = i \cot(iz) \\
\operatorname{sech} z &= \frac{1}{\cosh z} = \frac{\cosh(x)\cos(y) - i \sinh(x)\sin(y)}{\cos^2(y) + \sinh^2(x)} = \sec(iz) \\
\operatorname{csch} z &= \frac{1}{\sinh z} = \frac{\sinh(x)\cos(y) - i \cosh(x)\sin(y)}{\sin^2(y) + \sinh^2(x)} = i \csc(iz)
\end{aligned}
$$

4.4.5 Inverse Trigonometric and Hyperbolic Functions

To find analytical expressions for calculating the inverse of complex trigonometric and hyperbolic functions one can either manually derive them or look them up in a reliable reference book such as the one by Abramowitz and Stegun [19]. Fortunately, the steps involved in deriving expressions for both the trigonometric and hyperbolic inverse functions are very straight forward and pretty much follow a pattern which can be extended and applied towards deriving all the remaining functions. For example, consider finding an analytical expression for the inverse complex sine function. Let both z and w be complex numbers so that $w = \arcsin z$. By taking the sine of both sides of this equality and expressing $\sin w$ in exponential form we have

$$z = \sin w = \frac{e^{iw} - e^{-iw}}{2i}$$

from which we can solve for $e^{iw} = iz + \sqrt{1 - z^2}$ and then for w and thus finally arrive at the following expression for the inverse sine function of a complex number z

$$\arcsin z = -i \ln(iz + \sqrt{1 - z^2})$$

Similar derivations for the inverse cosine and tangent functions of a complex number z yields

$$
\begin{aligned}
\arccos z &= -i \ln(z + i\sqrt{1 - z^2}) \\
\arctan z &= (\frac{i}{2}) \ln(\frac{i + z}{i - z}) \quad \text{where } z \neq i
\end{aligned}
$$

Once these three inverse trigonometric functions have been obtained, the remaining others can be easily calculated by using the following identities

$$\begin{aligned}
\operatorname{arccsc} z &= \arcsin(\frac{1}{z}) = -i\ln(\frac{\sqrt{z^2-1}+i}{z}) \\
\operatorname{arcsec} z &= \arccos(\frac{1}{z}) = \quad i\ln(\frac{\sqrt{1-z^2}+1}{z}) \\
\operatorname{arccot} z &= \arctan(\frac{1}{z}) = \frac{i}{2}\ln(\frac{z-i}{z+i})
\end{aligned}$$

Inverse hyperbolic functions in the complex domain are defined analogously to those in the real domain. Since all these functions may be expressed using complex logarithms, they too are infinitely multi-valued. Therefore, their principal values are obtained from the corresponding principal values of their logarithms. Following a similar derivation process for the inverse trigonometric functions, the complex inverse hyperbolic functions can be shown to be

$$\begin{aligned}
\operatorname{arcsinh} z &= \ln(z+\sqrt{z^2+1}) \\
\operatorname{arccosh} z &= \ln(z+\sqrt{z^2-1}) \\
\operatorname{arctanh} z &= \frac{1}{2}\ln(\frac{1+z}{1-z})
\end{aligned}$$

As before, once these three inverse hyperbolic functions have been obtained, the remaining others can be easily calculated by using the following identities

$$\begin{aligned}
\operatorname{arccsch} z &= \operatorname{arcsinh}(\frac{1}{z}) = \ln(\frac{\sqrt{z^2+1}+1}{z}) \\
\operatorname{arcsech} z &= \operatorname{arccosh}(\frac{1}{z}) = \ln(\frac{\sqrt{1-z^2}+1}{z}) \\
\operatorname{arccoth} z &= \operatorname{arctanh}(\frac{1}{z}) = \frac{1}{2}\ln(\frac{z+1}{z-1})
\end{aligned}$$

As esthetically and appealing as these inverse trigonometric and hyperbolic functions may look, they are in fact computationally intensive due to the several underlying steps that must be undertaken by a computer to numerically evaluate them using complex numbers. From a purely computational perspective, it would be much faster and efficient to evaluate those functions by first expressing them directly in terms of their real and complex components. This approach not only eliminates the steps involved to convert complex numbers back and forth but also eliminates the computational overhead in having to frequently call upon special procedures to handle all the complex number arithmetic. Both Abramowitz and Stegun [19] and Thompson [26] have published the analytical expressions needed to numerically calculate these inverse trigonometric and hyperbolic functions for complex numbers in terms of their real and complex components. These functions are all listed below for reference and for later coding implementation in C#. Therefore, with $z = x + iy$ we have

$$\begin{aligned}
\arccos z &= \pm\{\arccos\beta - i\,\mathrm{sgn(y)}\ln[\alpha + \sqrt{\alpha^2 - 1}]\}\\
\arcsin z &= \pm\{\arcsin\beta + i\,\mathrm{sgn(y)}\ln[\alpha + \sqrt{\alpha^2 - 1}]\}\\
\arctan z &= (\frac{1}{2})\arctan(\frac{2y}{1 - x^2 - y^2}) + (\frac{i}{4})\ln[\frac{x^2 + (y+1)^2}{x^2 + (y-1)^2}] \quad \text{where } z^2 \neq -1
\end{aligned}$$

$$\begin{aligned}
\operatorname{arccosh} z &= \ln[\alpha + \sqrt{\alpha^2 - 1}] + i\,\mathrm{sgn(y)}\arccos\beta\\
\operatorname{arcsinh} z &= \mathrm{sgn(x)}\ln[\alpha' + \sqrt{\alpha'^2 - 1}] - i\arcsin\beta'\\
\operatorname{arctanh} z &= (\frac{1}{4})\ln[\frac{(x+1)^2 + y^2}{(x-1)^2 + y^2}] + (\frac{i}{2})\arctan(\frac{2y}{1 - x^2 - y^2}) \quad \text{where } y^2 \neq -1
\end{aligned}$$

where $sgn(y)$ = sign of y, as defined in Chapter 2, and

$$\left.\begin{matrix}\alpha\\ \beta\end{matrix}\right\} = \frac{1}{2}\left(\sqrt{(x+1)^2 + y^2} \pm \sqrt{(x-1)^2 + y^2}\right)$$

$$\left.\begin{matrix}\alpha'\\ \beta'\end{matrix}\right\} = \frac{1}{2}\left(\sqrt{x^2 + (y-1)^2} \pm \sqrt{x^2 + (y+1)^2}\right)$$

As before, once these inverse trigonometric and hyperbolic functions have been obtained, the remaining ones can be easily calculated by using the identities for arccsc z, arcsec z, arccot z, arccsch z, arcsech z, arccoth z given earlier in this chapter.

4.5 A Complex Number Library in C#

The pages that follow contain the source code listing of a proposed numerical library for handling both complex number arithmetic along with the other elementary mathematical functions in C# as described in this chapter. In some cases I have coded multiple versions of the same function for two reasons. First, some functions can be expressed using different formulas and this may have some impact on the time it takes to run a particular function on a given computer. In the case where processing time is of the essence, the reader might want to run timing experiments to see what version runs faster on a particular computer. Second, it is to show how close the coding can resemble the actual formulas. While this feature may be esthetically pleasing to the eye, it also serves to illustrate the powerful features of object oriented programming that comes with C#.

```
//////////////////////////////////////////////////////
// The complex struct represents a complex number  //
// of the general format z = x + iy where x = Re(z)//
// = real part and y = Im(z) = imaginary part. A   //
// complex instance has public double-precision    //
// floating point data members real and imag.      //
//////////////////////////////////////////////////////
[Serializable]
public struct Complex
{
    private double real;
    private double imag;

    public double Real
    {
        get {return(real);}
        set {real = value;}
    }

    public double Imag
    {
        get {return (imag);}
        set {imag = value;}
    }

    public Complex(double x, double y)
    {
        this.real = x;
        this.imag = y;
    }

    public static implicit operator double(Complex z)
    {
        return z.real;
    }

    public static explicit operator Complex(double x)
    {
        return (new Complex(x,0.0));
    }

    public static Complex CZero = new Complex(0.0,0.0);
    public static Complex COne = new Complex(1.0,0.0);
    public static Complex i = new Complex(0.0, 1.0);
    public static Complex CNaN = new Complex(double.NaN,double.NaN);
    public static Complex CInfinity =
        new Complex(double.PositiveInfinity,double.PositiveInfinity);

    public bool IsCZero
    {get {return ((real==0.0) && (imag==0.0));}}

    public bool IsCOne
    {get {return ((real==1.0) && (imag==0.0));}}
```

```
public bool Isi
{get {return ((real==0.0) && (imag==1.0));}}

public bool IsCNaN
{get {return double.IsNaN(real) || double.IsNaN(imag);}}

public bool IsCInfinity
{get {return double.IsInfinity(real) ||
             double.IsInfinity(imag);}}

public bool IsReal
{get {return (imag==0.0);}}

public bool IsImag
{get {return (real==0.0);}}

// Returns the argument of a complex number.
public static double CArg(Complex z)
{
    return (Math.Atan2(z.imag,z.real));
}

// Returns the conjugate of a complex number z.
public static Complex CConj(Complex z)
{
    return (new Complex(z.real,-z.imag));
}

// Returns the norm (or modulus) of a complex number.
public static double CNorm(Complex z)
{
    return (Math.Sqrt((z.real*z.real) +
                      (z.imag*z.imag)));
}

// Returns the norm (or modulus) of a complex number
// avoiding potential overflow and/or underflow for
// very small or very large numbers.
public static double CNorm2(Complex z)
{
    double x = z.real;
    double y = z.imag;

    if (Math.Abs(x) < Math.Abs(y))
    {
      return (Math.Abs(y)*Math.Sqrt(1.0+(x/y)*(x/y)));
    }
    else
    {
      return (Math.Abs(x)*Math.Sqrt(1.0+(y/x)*(y/x)));
    }
}

// Returns the inverse of a complex number.
public static Complex Inv(Complex z)
{ return (1.0/z); }
```

```
// Returns the real part of a complex number.
public static double Re(Complex z)
{
    return (z.real);
}

// Returns the imaginary part of a complex number.
public static double Im(Complex z)
{
    return (z.imag);
}

// Converts:(r,theta) ----> (x,y)
public static Complex FromPolarToXY(double r,double theta)
{
  return (new Complex((r*Math.Cos(theta)),(r*Math.Sin(theta))));
}

// Converts:    (x,y) ----> (r,theta)
public static Complex FromXYToPolar(Complex z)
{
  return (new Complex(CNorm(z),CArg(z)));
}

// Returns the negation of a complex number.
public static Complex CNeg(Complex z)
{
    return (-z);
}

// Returns the sum of two complex numbers z1 and z2.
public static Complex CAdd(Complex z1, Complex z2)
{
    return (z1+z2);
}

// Returns the sum of a real with a complex number.
public static Complex CAdd(double x, Complex z)
{
    return (x+z);
}

// Returns the sum of a complex with a real number.
public static Complex CAdd(Complex z, double x)
{
    return (z+x);
}

// Returns the difference between two complex numbers.
public static Complex CSub(Complex z1, Complex z2)
{
    return (z1-z2);
}
```

```
// Returns the difference between a real and a complex number.
public static Complex CSub(double x, Complex z)
{
    return (x-z);
}

// Returns the difference between a complex and a real number.
public static Complex CSub(Complex z, double x)
{
    return (z-x);
}

// Returns the product between two complex numbers.
public static Complex CMult(Complex z1, Complex z2)
{
    return (z1*z2);
}

// Returns the product of a real with a complex number.
public static Complex CMult(double x, Complex z)
{
    return (x*z);
}

// Returns the product of a complex and a real number.
public static Complex CMult(Complex z, double x)
{
    return (z*x);
}

// Returns the quotient of dividing two complex numbers.
public static Complex CDiv(Complex z1, Complex z2)
{
    return (z1/z2);
}

// Returns the quotient of dividing a real by a complex number.
public static Complex CDiv(double x, Complex z)
{
    return (x/z);
}

// Returns the quotient of dividing a complex by a real number.
public static Complex CDiv(Complex z, double x)
{
    return (z/x);
}

// Returns the quotient of dividing two complex numbers
// avoiding potential underflow and/or overflow for
// very small or very large numbers
public static Complex CDiv2(Complex z1, Complex z2)
{
    double x1 = z1.real; double y1 = z1.imag;
    double x2 = z2.real; double y2 = z2.imag;
```

```
        Complex u;
        double denom;

        if (z2.IsCZero) return Complex.CInfinity;

        if (Math.Abs(x2) < Math.Abs(y2))
        {
             denom = x2*(x2/y2)+y2;
            u.real = (x1*(x2/y2)+y1)/denom;
            u.imag = (y1*(x2/y2)-x1)/denom;
        }
        else
        {
             denom = x2+y2*(y2/x2);
            u.real = (x1+y1*(y2/x2))/denom;
            u.imag = (y1-x1*(y2/x2))/denom;
        }
        return u;
    }

    public static Complex operator +(Complex z)
    {
        return z;
    }

    public static Complex operator +(Complex z1,Complex z2)
    {
        return (new Complex(z1.real+z2.real,z1.imag+z2.imag));
    }

    // Returns the sum of a real number with a complex number.
    public static Complex operator +(double x,Complex z)
    {
        return (new Complex(x+z.real,z.imag));
    }

    // Returns the sum of a complex number with a real number.
    public static Complex operator +(Complex z,double x)
    {
        return (new Complex(z.real+x,z.imag));
    }

    // Returns the negation of a complex number.
    public static Complex operator -(Complex z)
    {
        return (new Complex(-z.real,-z.imag));
    }

    // Returns the difference between two complex numbers.
    public static Complex operator -(Complex z1,Complex z2)
    {
        return (new Complex(z1.real-z2.real,
                            z1.imag-z2.imag));
    }
```

```
// Returns the difference of a real with a complex number.
public static Complex operator -(double x,Complex z)
{
    return (new Complex(x-z.real,-z.imag));
}

// Returns the difference of a complex with a real number.
public static Complex operator -(Complex z,double x)
{
    return (new Complex(z.real-x,z.imag));
}

// Returns the product of two complex numbers z1 * z2.
public static Complex operator *(Complex z1,Complex z2)
{
    double x = (z1.real*z2.real)-(z1.imag*z2.imag);
    double y = (z1.real*z2.imag)+(z1.imag*z2.real);
    return (new Complex(x,y));
}

// Returns the product of a real and a complex number.
public static Complex operator *(double x, Complex z)
{
    return (new Complex(x*z.real,x*z.imag));
}

// Returns the product of a complex and a real number.
public static Complex operator *(Complex z,double x)
{
    return (new Complex(z.real*x, z.imag*x));
}

// Returns the quotient of two complex numbers z1 / z2.
public static Complex operator /(Complex z1,Complex z2)
{
  if (z2.IsCZero) return Complex.CInfinity;
  double denom = (double)(Math.Pow(z2.real, 2.0) +
                          Math.Pow(z2.imag, 2.0));
  double x = ((z1.real*z2.real)+(z1.imag*z2.imag))/denom;
  double y = ((z1.imag*z2.real)-(z1.real*z2.imag))/denom;
  return (new Complex(x,y));
}

// Returns the quotient of dividing a real by a complex number.
public static Complex operator /(double x,Complex z)
{
  if (z.IsCZero) return Complex.CInfinity;
  double denom = (double)(Math.Pow(z.real, 2.0) +
                          Math.Pow(z.imag, 2.0));
  double re = (x*z.real)/denom;
  double im = (0.0-(x*z.imag))/denom;
  return (new Complex(re,im));
}
```

```
// Returns the quotient of dividing a complex by a real number.
public static Complex operator /(Complex z,double x)
{
    if (x==0.0) return Complex.CInfinity;
    double re = z.real/x;
    double im = z.imag/x;
    return (new Complex(re,im));
}

// Tests for equality of two complex numbers.
public static bool operator ==(Complex z1,Complex z2)
{
    return ((z1.real==z2.real) && (z1.imag==z2.imag));
}

// Tests for inequality of two complex numbers.
public static bool operator !=(Complex z1, Complex z2)
{
    return (!(z1==z2));
}

// Tests for equality of between two complex numbers.
public override bool Equals(Object obj)
{
    return ((obj is Complex) && (this == (Complex)obj));
}

// Returns an integer hash code for this complex number.
// If you override Equals, override GetHashCode too.
public override int GetHashCode()
{
    //return this.ToString().GetHashCode();
    return (real.GetHashCode() ^ imag.GetHashCode());
}

// Returns a formatted string representation in
// the form z = x + iy for a complex number.
public override string ToString()
{
    return (String.Format("{0} + {1}i", real, imag));
}

public static Complex CExp(Complex z)
{
  double x = z.real;
  double y = z.imag;
  double expx = Math.Exp(x);
  return (new Complex(expx*Math.Cos(y),expx*Math.Sin(y)));
}

// Logarithm of complex z to base e
public static Complex CLog(Complex z)
{
    return (new Complex(Math.Log(Complex.CNorm(z)),
                        Math.Atan2(z.imag,z.real)));
}
```

```
// Another version of logarithm of complex z to base e
public static Complex CLog2(Complex z)
{
    double x = z.real;
    double y = z.imag;
    double re = 0.5*Math.Log(x*x + y*y);
    double im = Math.Atan2(y,x);
    return (new Complex(re,im));
}

// Logarithm of complex z to base 10
public static Complex CLog10(Complex z)
{
    return (Complex.CLog(z)/Complex.CLog((Complex)10.0));
}

// Logarithm of complex z1 to complex base z2
public static Complex CLogb(Complex z1, Complex z2)
{
    return (Complex.CLog(z1)/Complex.CLog(z2));
}

// Logarithm of real x to complex base z2
public static Complex CLogb(double x, Complex z2)
{
    return (Complex.CLog((Complex)x)/Complex.CLog(z2));
}

// Logarithm of complex z1 to real base x
public static Complex CLogb(Complex z1, double x)
{
    return (Complex.CLog(z1)/Complex.CLog((Complex)x));
}

// Complex z raised to the power of complex w
public static Complex CPow(Complex z, Complex w)
{
    return (Complex.CExp(w*Complex.CLog(z)));
}

// Complex z raised to the power of complex w (ver 2)
public static Complex CPow2(Complex z, Complex w)
{
  double x1 = z.real; double y1 = z.imag;
  double x2 = w.real; double y2 = w.imag;

  double r1 = Math.Sqrt(x1*x1 + y1*y1);
  double theta1 = Math.Atan2(y1,x1);
  double phi = theta1*x2 + y2*Math.Log(r1);

  double re = Math.Pow(r1,x2)*Math.Exp(-theta1*y2)*Math.Cos(phi);
  double im = Math.Pow(r1,x2)*Math.Exp(-theta1*y2)*Math.Sin(phi);

  return (new Complex(re,im));
}
```

```
// Complex z raised to the power of real x
public static Complex CPow(Complex z, double x)
{
    return (Complex.CExp(x*Complex.CLog(z)));
}

// Complex z raised to the power of real x (ver 2)
public static Complex CPow2(Complex z, double x)
{
    double x1 = z.real;
    double y1 = z.imag;

    double r1 = Math.Sqrt(x1*x1 + y1*y1);
    double theta1 = Math.Atan2(y1,x1);
    double phi = theta1*x;

    double re = Math.Pow(r1,x)*Math.Cos(phi);
    double im = Math.Pow(r1,x)*Math.Sin(phi);

    return (new Complex(re, im));
}

// Real x raised to the power of complex z
public static Complex CPow(double x, Complex z)
{
    return (Complex.CExp(z*Math.Log(x)));
}

// Real x raised to the power of complex z (ver 2)
public static Complex CPow2(double x, Complex w)
{
    double x2 = w.real;
    double y2 = w.imag;

    double r1 = Math.Sqrt(x*x);
    double theta1 = Math.Atan2(0.0,x);
    double phi = theta1*x2 + y2*Math.Log(r1);

    double re = Math.Pow(r1,x2)*Math.Cos(phi);
    double im = Math.Pow(r1,x2)*Math.Sin(phi);

    return (new Complex(re,im));
}

// Complex root w of complex number z
public static Complex CRoot(Complex z,Complex w)
{
    return (Complex.CExp(Complex.CLog(z)/w));
}

// Real root x of complex number z
public static Complex CRoot(Complex z,double x)
{
    return (Complex.CExp(Complex.CLog(z)/x));
}
```

```
// Complex root z of real number x
public static Complex CRoot(double x,Complex z)
{
    return (Complex.CExp(Math.Log(x)/z));
}

// Complex square root of complex number z
public static Complex CSqrt(Complex z)
{
    return (Complex.CExp(Complex.CLog(z)/2.0));
}

// Complex sine of complex number z
public static Complex CSin(Complex z)
{
  return ((Complex.CExp(i*z)-Complex.CExp(-i*z))/(2.0*i));
}

// Complex sine of complex number z (ver 2)
public static Complex CSin2(Complex z)
{
    double x = z.real;
    double y = z.imag;
    double re = Math.Sin(x)*Math.Cosh(y);
    double im = Math.Cos(x)*Math.Sinh(y);
    return (new Complex(re,im));
}

// Complex cosine of complex number z
public static Complex CCos(Complex z)
{
    return((Complex.CExp(i*z)+Complex.CExp(-i*z))/2.0);
}

// Complex cosine of complex number z (ver 2)
public static Complex CCos2(Complex z)
{
    double x = z.real;
    double y = z.imag;
    double re =  Math.Cos(x)*Math.Cosh(y);
    double im = -Math.Sin(x)*Math.Sinh(y);
    return (new Complex(re,im));
}

// Complex tangent of complex number z
public static Complex CTan(Complex z)
{
    return (Complex.CSin(z)/Complex.CCos(z));
}

// Complex tangent of complex number z (ver 2)
public static Complex CTan2(Complex z)
{
    double x2 = 2.0*z.real;
    double y2 = 2.0*z.imag;
```

```
        double denom = Math.Cos(x2)+Math.Cosh(y2);
        if (denom == 0.0) return Complex.CInfinity;
        double re = Math.Sin(x2)/denom;
        double im = Math.Sinh(y2)/denom;
        return (new Complex(re,im));
    }

    // Complex cotangent of complex number z
    public static Complex CCot(Complex z)
    {
        return (Complex.CCos(z)/Complex.CSin(z));
    }

    // Complex cotangent of complex number z (ver 2)
    public static Complex CCot2(Complex z)
    {
        double x2 = 2.0*z.real;
        double y2 = 2.0*z.imag;
        double denom = Math.Cosh(y2)-Math.Cos(x2);
        if (denom==0.0) return Complex.CInfinity;
        double re =  Math.Sin(x2)/denom;
        double im = -Math.Sinh(y2)/denom;
        return (new Complex(re,im));
    }

    // Complex secant of complex number z
    public static Complex CSec(Complex z)
    {
        return (1.0/Complex.CCos(z));
    }

    // Complex secant of complex number z (ver 2)
    public static Complex CSec2(Complex z)
    {
        double x = z.real;
        double y = z.imag;

        double denom = Math.Cos(x)*Math.Cos(x)+
                       Math.Sinh(y)*Math.Sinh(y);
        if (denom == 0.0) return Complex.CInfinity;
        double re = Math.Cos(x)*Math.Cosh(y)/denom;
        double im = Math.Sin(x)*Math.Sinh(y)/denom;
        return (new Complex(re,im));
    }

    // Complex cosecant of complex number z
    public static Complex CCsc(Complex z)
    {
        return (1.0/Complex.CSin(z));
    }

    // Complex cosecant of complex number z (ver 2)
    public static Complex CCsc2(Complex z)
    {
        double x = z.real;
        double y = z.imag;
        double denom = Math.Sin(x)*Math.Sin(x)+
                       Math.Sinh(y)*Math.Sinh(y);
        if (denom==0.0) return Complex.CInfinity;
        double re =  Math.Sin(x)*Math.Cosh(y)/denom;
        double im = -Math.Cos(x)*Math.Sinh(y)/denom;
        return (new Complex(re,im));
    }

    // Complex ArcSine of complex number z
    public static Complex CArcSin(Complex z)
    {
     return (-i*Complex.CLog((i*z)+Complex.CSqrt(1.0-(z*z))));
    }

    // Complex ArcSine of complex number z (ver 2)
    public static Complex CArcSin2(Complex z)
    {
        double x = z.real;
        double y = z.imag;

        double ysqd = y*y;
        double rtpos = Math.Sqrt(Math.Pow(x+1.0,2.0)+ysqd);
        double rtneg = Math.Sqrt(Math.Pow(x-1.0,2.0)+ysqd);
        double alpha = 0.5*(rtpos+rtneg);
        double  beta = 0.5*(rtpos-rtneg);

        double InvSinZRe = Math.Asin(beta);
        double InvSinZIm = Math.Sign(y) *
            Math.Log(alpha + Math.Sqrt(alpha*alpha-1.0));

        return (new Complex(InvSinZRe, InvSinZIm));
    }

    // Complex ArcCosine of complex number z
    public static Complex CArcCos(Complex z)
    {
        return(-i*Complex.CLog(z+i*Complex.CSqrt(1.0-(z*z))));
    }

    // Complex ArcCosine of complex number z (ver 2)
    public static Complex CArcCos2(Complex z)
    {
        double x = z.real;
        double y = z.imag;

        double ysqd = y*y;
        double rtpos = Math.Sqrt(Math.Pow(x+1.0,2.0)+ysqd);
        double rtneg = Math.Sqrt(Math.Pow(x-1.0,2.0)+ysqd);
        double alpha = 0.5*(rtpos+rtneg);
        double  beta = 0.5*(rtpos-rtneg);
        double InvCosZRe = Math.Acos(beta);
        double InvCosZIm = -Math.Sign(y) *
            Math.Log(alpha + Math.Sqrt(alpha*alpha-1.0));

        return (new Complex(InvCosZRe,InvCosZIm));
    }
```

```
// Complex ArcTangent of complex number z
public static Complex CArcTan(Complex z)
{
  return ((i/2.0)*Complex.CLog((i+z)/(i-z)));
}

// Complex ArcTangent of complex number z (ver 2)
public static Complex CArcTan2(Complex z)
{
  double x = z.real;
  double y = z.imag;

  double xsqd = x * x;
  double ysqd = y * y;

  double InvTanZRe=0.5*Math.Atan2(2.0*x,1.0-xsqd-ysqd);
  double InvTanZIm=0.25*Math.Log((xsqd+
  Math.Pow(y+1.0,2.0))/(xsqd+Math.Pow(y-1.0,2.0)));

  return (new Complex(InvTanZRe,InvTanZIm));
}

// Complex ArcCotangent of complex number z
public static Complex CArcCot(Complex z)
{
  return (Complex.CArcTan(1.0/z));
}

// Complex ArcCotangent of complex number z (ver 2)
public static Complex CArcCot2(Complex z)
{
  return ((i/2.0)*(Complex.CLog((z-i)/(z+i))));
}

// Complex ArcSecant of complex number z
public static Complex CArcSec(Complex z)
{
  return (Complex.CArcCos(1.0/z));
}

// Complex ArcSecant of complex number z (ver 2)
public static Complex CArcSec2(Complex z)
{
  return (i*Complex.CLog((Complex.CSqrt(1.0-(z*z))+1.0)/z));
}

// Complex ArcCosecant of complex number z
public static Complex CArcCsc(Complex z)
{
  return (Complex.CArcSin(1.0/z));
}

// Complex ArcCosecant of complex number z (ver 2)
public static Complex CArcCsc2(Complex z)
{ return (-i*Complex.CLog((Complex.CSqrt((z*z)-1.0)+i)/z)); }
```

```
// hyperbolic sine of complex number z
public static Complex CSinh(Complex z)
{
    return ((Complex.CExp(z)-Complex.CExp(-z))/2.0);
}

// Hyperbolic Sine of complex number z (ver 2)
public static Complex CSinh2(Complex z)
{
    double x = z.real;
    double y = z.imag;
    double SinhZRe = Math.Sinh(x)*Math.Cos(y);
    double SinhZIm = Math.Cosh(x)*Math.Sin(y);
    return (new Complex(SinhZRe,SinhZIm));
}

// Complex Hyperbolic Sine
// of complex number z (ver 3)
public static Complex CSinh3(Complex z)
{
    return (-i*Complex.CSin(i*z));
}

// Hyperbolic Cosine of complex number z
public static Complex CCosh(Complex z)
{
    return ((Complex.CExp(z)+Complex.CExp(-z))/2.0);
}

// Hyperbolic Cosine of complex number z (ver 2)
public static Complex CCosh2(Complex z)
{
    double x = z.real;
    double y = z.imag;
    double CoshZRe = Math.Cosh(x)*Math.Cos(y);
    double CoshZIm = Math.Sinh(x)*Math.Sin(y);
    return (new Complex(CoshZRe,CoshZIm));
}

// Hyperbolic Cosine of complex number z (ver 3)
public static Complex CCosh3(Complex z)
{
    return (Complex.CCos(i*z));
}

// Complex Hyperbolic Tangent of complex number z
public static Complex CTanh(Complex z)
{
    return (Complex.CSinh(z)/Complex.CCosh(z));
}

// Hyperbolic Tangent of complex number z (ver 2)
public static Complex CTanh2(Complex z)
{
    double twox = 2.0*z.real;
```

```
        double twoy = 2.0*z.imag;
        double denom = Math.Cosh(twox)+Math.Cos(twoy);

        double TanhZRe = Math.Sinh(twox)/denom;
        double TanhZIm = Math.Sin(twoy)/denom;
        return (new Complex(TanhZRe,TanhZIm));
    }

    // Hyperbolic Tangent of complex number z (ver 3)
    public static Complex CTanh3(Complex z)
    {
        return (-i*Complex.CTan(i*z));
    }

    // Hyperbolic Cotangent of complex number z
    public static Complex CCoth(Complex z)
    {
        return (Complex.CCosh(z)/Complex.CSinh(z));
    }

    // Hyperbolic Cotangent of complex number z (ver 2)
    public static Complex CCoth2(Complex z)
    {
        return (Complex.CCosh2(z)/Complex.CSinh2(z));
    }

    // Hyperbolic Cotangent of complex number z (ver 3)
    public static Complex CCoth3(Complex z)
    {
        double twox = 2.0*z.real;
        double twoy = 2.0*z.imag;
        double denom = Math.Cosh(twox)-Math.Cos(twoy);

        double CothZRe = Math.Sinh(twox)/denom;
        double CothZIm = -Math.Sin(twoy)/denom;
        return (new Complex(CothZRe,CothZIm));
    }

    // Hyperbolic Cotangent of complex number z (ver 4)
    public static Complex CCoth4(Complex z)
    {
        return (i*Complex.CCot(i*z));
    }

    // Hyperbolic Secant of complex number z
    public static Complex CSech(Complex z)
    {
        return (1.0/Complex.CCosh(z));
    }

    // Hyperbolic Secant of complex number z (ver 2)
    public static Complex CSech2(Complex z)
    {
        return(1.0/Complex.CCosh2(z));
    }
```

```
// Hyperbolic Secant of complex number z (ver 3)
public static Complex CSech3(Complex z)
{
    double CoshX = Math.Cosh(z.real);
    double CosY = Math.Cos(z.imag);
    double SinhX = Math.Sinh(z.real);
    double SinY = Math.Sin(z.imag);

    double denom = CosY*CosY+SinhX*SinhX;

    double CSechZRe =  (CoshX*CosY)/denom;
    double CSechZIm = -(SinhX*SinY)/denom;
    return (new Complex(CSechZRe,CSechZIm));
}

// Hyperbolic Secant of complex number z (ver 4)
public static Complex CSech4(Complex z)
{
    return (Complex.CSec(i*z));
}

// Hyperbolic Cosecant of complex number z
public static Complex CCsch(Complex z)
{
    return (1.0/Complex.CSinh(z));
}

// Hyperbolic Cosecant of complex number z (ver 2)
public static Complex CCsch2(Complex z)
{
    return (1.0/Complex.CSinh2(z));
}

// Hyperbolic Cosecant of complex number z (ver 3)
public static Complex CCsch3(Complex z)
{
    double CoshX = Math.Cosh(z.real);
    double CosY = Math.Cos(z.imag);
    double SinhX = Math.Sinh(z.real);
    double SinY = Math.Sin(z.imag);

    double denom = SinY*SinY+SinhX*SinhX;

    double CSechZRe =  (SinhX*CosY)/denom;
    double CSechZIm = -(CoshX*SinY)/denom;
    return (new Complex(CSechZRe, CSechZIm));
}

// Hyperbolic Cosecant of complex number z (ver 4)
public static Complex CCsch4(Complex z)
{
    return (i*Complex.CCsc(i*z));
}
```

```
// Inverse Hyperbolic Sine of complex number z
public static Complex CArcSinh(Complex z)
{
    return (Complex.CLog(z+Complex.CSqrt((z*z)+1.0)));
}

// Inverse Hyperbolic Sine of complex number z (ver 2)
public static Complex CArcSinh2(Complex z)
{
    double x = z.real;
    double y = z.imag;

    double xsqd = x*x;
    double rtpos = Math.Sqrt(Math.Pow(y-1.0,2.0)+xsqd);
    double rtneg = Math.Sqrt(Math.Pow(y+1.0,2.0)+xsqd);
    double alphap = 0.5*(rtpos+rtneg);
    double betap = 0.5*(rtpos-rtneg);

    double InvSinhZRe = Math.Sign(x) *
        Math.Log(alphap+Math.Sqrt(alphap*alphap-1));
    double InvSinhZIm = -Math.Asin(betap);

    return (new Complex(InvSinhZRe,InvSinhZIm));
}

// Inverse Hyperbolic Cosine of complex number z
public static Complex CArcCosh(Complex z)
{
    return (Complex.CLog(z+Complex.CSqrt(z*z-1.0)));
}

// Inverse Hyperbolic Cosine of complex number z (ver 2)
public static Complex CArcCosh2(Complex z)
{
    double x = z.real;
    double y = z.imag;

    double ysqd = y*y;
    double rtpos = Math.Sqrt(Math.Pow(x+1.0,2.0)+ysqd);
    double rtneg = Math.Sqrt(Math.Pow(x-1.0,2.0)+ysqd);
    double alpha = 0.5*(rtpos+rtneg);
    double beta = 0.5*(rtpos-rtneg);

    double InvCoshZRe =
        Math.Log(alpha+Math.Sqrt(alpha*alpha-1));
    double InvCoshZIm = Math.Sign(y)*Math.Acos(beta);

    return (new Complex(InvCoshZRe,InvCoshZIm));
}

// Inverse Hyperbolic Tangent of complex number z
public static Complex CArcTanh(Complex z)
{
    return (0.5*Complex.CLog((1.0+z)/(1.0 -z)));
}
```

```
    // Inverse Hyperbolic Tangent of complex number z (ver 2)
    public static Complex CArcTanh2(Complex z)
    {
     double x = z.real;
     double y = z.imag;
     double xsqd = x*x;
     double ysqd = y*y;

     double InvTanhZRe = 0.25 * Math.Log((ysqd +
     Math.Pow(x+1.0,2.0)) / (ysqd+Math.Pow(x-1.0,2.0)));

     double InvTanhZIm = 0.5*Math.Atan2(2.0*y,1.0-xsqd-ysqd);

     return (new Complex(InvTanhZRe,InvTanhZIm));
    }

    // Inverse Hyperbolic Cotangent of complex number z
    public static Complex CArcCoth(Complex z)
    {
        return (Complex.CArcTanh(1.0/z));
    }

    // Inverse Hyperbolic Cotangent of complex number z (ver 2)
    public static Complex CArcCoth2(Complex z)
    {
        return (0.5 * Complex.CLog((z+1.0)/(z-1.0)));
    }

    // Inverse Hyperbolic Secant of complex number z
    public static Complex CArcSech(Complex z)
    {
        return (Complex.CArcCosh(1.0/z));
    }

    // Inverse Hyperbolic Secant of complex number z (ver 2)
    public static Complex CArcSech2(Complex z)
    {
     return (Complex.CLog((1.0+Complex.CSqrt(1.0-(z*z)))/z));
    }

    // Inverse Hyperbolic Cosecant of complex number z
    public static Complex CArcCsch(Complex z)
    {
        return (Complex.CArcSinh(1.0/z));
    }

    // Inverse Hyperbolic Cosecant of complex number z (ver 2)
    public static Complex CArcCsch2(Complex z)
    { return (Complex.CLog((1.0+Complex.CSqrt(1.0+(z*z)))/z)); }
}
```

4.6 A Complex Number Vector Library in C#

Following the same concepts introduced in an earlier chapter on the topic of real number vectors, we can now extend those ideas to enable vector structures to handle complex numbers. I will, however, omit repeating myself with a full blown account of detailed explanations of the concepts involved. Instead, I will only point out the major differences between the complex and the real number vector structures and include a listing of the source code to illustrate the implementation of these new concepts. The basic definitions and mathematical operations of complex vectors are similar to those of real vectors. However, instead of using real numbers, we now use complex numbers and the mathematical operations now all follow the well established rules for complex numbers. Although the basic definitions and mathematical operations of complex vectors are similar to those of real vectors, there is a minor difference in the way that the dot product is handled. For two complex vectors, their dot product is defined by taking the conjugate one of the two vectors and applying the dot product formula as in the case of real number vectors.

```
public struct CVector : ICloneable
{
    // Fields
    private int ndim;
    private Complex[] vector;

    // Constructors
    public CVector(int ndim)
    {
        this.ndim = ndim;
        this.vector = new Complex[ndim];
        for (int i = 0; i < ndim; i++)
        {
            vector[i] = Complex.CZero;
        }
    }

    public CVector(Complex[] cv)
    {
        this.ndim = cv.Length;
        this.vector = cv;
    }

    public Complex this[int i] //Indexers
    {
        get
        {
            if (i < 0 || i > ndim)
            { throw new Exception("i is out of range!"); }
            return vector[i];
        }
        set { vector[i] = value; }
    }
```

```
// Accessors
public int GetCVectorSize
{
    get { return ndim; }
}

// Override Methods
public override string ToString()
{
    string str = "(";
    for (int i = 0; i < ndim - 1; i++)
    {
        str += vector[i] + ", ";
    }
    str += vector[ndim - 1] + ")";
    return str;
}

public override bool Equals(object obj)
{
    return (obj is CVector) && this.Equals((CVector)obj);
}

public bool Equals(CVector cv)
{
    return vector == cv.vector;
}

public override int GetHashCode()
{
    return vector.GetHashCode();
}

public static bool operator ==(CVector v1,CVector v2)
{
    return v1.Equals(v2);
}

public static bool operator !=(CVector v1,CVector v2)
{
    return !v1.Equals(v2);
}

public static CVector operator +(CVector cv)
{
    return cv;
}

public static CVector operator +(CVector v1,CVector v2)
{
    CVector result = new CVector(v1.GetCVectorSize);
    for (int i = 0; i < v1.GetCVectorSize; i++)
    { result[i] = v1[i] + v2[i]; }
    return result;
}
```

```
public static CVector operator +(CVector cv,double d)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    {
        result[i] = cv[i] + d;
    }
    return result;
}

public static CVector operator +(double d,CVector cv)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    {
        result[i] = cv[i] + d;
    }
    return result;
}

public static CVector operator +(CVector cv, Complex cn)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    {
        result[i] = cv[i] + cn;
    }
    return result;
}

public static CVector operator +(Complex cn, CVector cv)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    {
        result[i] = cv[i] + cn;
    }
    return result;
}

public static CVector operator -(CVector cv)
{
    Complex[] result = new Complex[cv.GetCVectorSize];
    for (int i = 0; i < cv.GetCVectorSize; i++)
    {  result[i] = -cv[i]; }
    return new CVector(result);
}

public static CVector operator -(CVector v1, CVector v2)
{
    CVector result = new CVector(v1.GetCVectorSize);
    for (int i = 0; i < v1.GetCVectorSize; i++)
    { result[i] = v1[i] - v2[i]; }
    return result;
}
```

```
public static CVector operator -(CVector cv, double d)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    {
        result[i] = cv[i] - d;
    }
    return result;
}

public static CVector operator -(double d, CVector cv)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    {
        result[i] = d - cv[i];
    }
    return result;
}

public static CVector operator -(CVector cv, Complex cn)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    {
        result[i] = cv[i] - cn;
    }
    return result;
}

public static CVector operator -(Complex cn, CVector cv)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    {
        result[i] = cn - cv[i];
    }
    return result;
}

public static CVector operator *(CVector cv, double d)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    { result[i] = cv[i] * d; }
    return result;
}

public static CVector operator *(double d, CVector cv)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    { result[i] = d * cv[i]; }
    return result;
}
```

```
public static CVector operator *(CVector cv, Complex cn)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    {
        result[i] = cv[i] * cn;
    }
    return result;
}

public static CVector operator *(Complex cn, CVector cv)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    {
        result[i] = cn * cv[i];
    }
    return result;
}

public static CVector Product(CVector v1, CVector v2)
{
    CVector result = new CVector(v1.GetCVectorSize);
    for (int i = 0; i < v1.GetCVectorSize; i++)
    {
        result[i] = v1[i] * v2[i];
    }
    return result;
}

public static CVector operator /(CVector cv, double d)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    {
        result[i] = cv[i] / d;
    }
    return result;
}

public static CVector operator /(double d, CVector cv)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    { result[i] = d / cv[i]; }
    return result;
}

public static CVector operator /(CVector cv, Complex cn)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    { result[i] = cv[i] / cn; }
    return result;
}
```

```
public static CVector operator /(Complex cn, CVector cv)
{
    CVector result = new CVector(cv.GetCVectorSize);
    for (int i = 0; i < cv.GetCVectorSize; i++)
    {
        result[i] = cn / cv[i];
    }
    return result;
}

// Makes a clone copy of a complex vector
public CVector Clone()
{
    CVector cv = new CVector(vector);
    cv.vector = (Complex[])vector.Clone();
    return cv;
}

object ICloneable.Clone()
{
    return Clone();
}

// Methods
// Calculates the dot product of a complex vector
public static Complex DotProduct(CVector v1, CVector v2)
{
    Complex result = Complex.CZero;
    for (int i = 0; i < v1.GetCVectorSize; i++)
    {
        result += CConj(v1[i]) * v2[i];
    }
    return result;
}

// Calculates the norm of a complex vector
public double GetNorm()
{
    Complex result = Complex.CZero;
    for (int i = 0; i < this.GetCVectorSize; i++)
    {
        result += CConj(vector[i]) * vector[i];
    }
    return Math.Sqrt(result.Real*result.Real +
                     result.Imag*result.Imag);
}

// Calculates the square of the norm of a complex vector
public double GetNormSquare()
{
    Complex result = Complex.CZero;
    for (int i = 0; i < this.GetCVectorSize; i++)
    { result += CConj(vector[i]) * vector[i]; }
    return (result.Real*result.Real+result.Imag*result.Imag);
}
```

```
    // Normalizes a complex vector
    public void Normalize()
    {
      double norm = GetNorm();
      if (norm == 0)
      {
       throw new Exception("Normalized a vector with norm of zero!");
      }
      for (int i = 0; i < this.GetCVectorSize; i++)
      {
       vector[i] /= norm;
      }
    }

    // Calculates the unit vector of a complex vector
    public CVector GetUnitVector()
    {
        CVector result = new CVector(vector);
        result.Normalize();
        return result;
    }

    // Calculates the complex conjugate of a complex vector
    public CVector GetConjugate()
    {
        for (int i = 0; i < this.GetCVectorSize; i++)
        {
            vector[i] = CConj(vector[i]);
        }
        return new CVector(vector);
    }

    // Swaps entries in a complex vector
    public CVector SwapCVectorEntries(int m, int n)
    {
        Complex temp = vector[m];
        vector[m] = vector[n];
        vector[n] = temp;
        return new CVector(vector);
    }

    // Calculates the cross product between two complex vectors
    public static CVector CrossProduct(CVector v1, CVector v2)
    {
        if (v1.GetCVectorSize != 3)
        {
          throw new Exception("Vector v1 must be 3 dimensional!");
        }
        CVector result = new CVector(3);
        result[0] = v1[1] * v2[2] - v1[2] * v2[1];
        result[1] = v1[2] * v2[0] - v1[0] * v2[2];
        result[2] = v1[0] * v2[1] - v1[1] * v2[0];
        return result;
    }
}
```

4.7 A Complex Number Matrix Library in C#

Following the same concepts introduced in an earlier chapter on the topic of real number matrices, we can now extend those ideas to enable matrix structures to handle complex numbers. I will, however, omit repeating myself with a full blown account of detailed explanations of the concepts involved. Instead, I will only point out the major differences between the complex and the real number matrix structures and include a listing of the source code to illustrate the implementation of these new concepts. The basic definitions and mathematical operations of complex matrices are similar to those of real matrices. However, instead of using real numbers, we now use complex numbers and the mathematical operations now all follow the well established rules for complex numbers.

```
public struct CMatrix : ICloneable
{
    // Fields
    private int nRows;  private int nCols;
    private Complex[,] matrix;

    // Constructors
    public CMatrix(int nRows, int nCols)
    {
        this.nRows = nRows;  this.nCols = nCols;
        this.matrix = new Complex[nRows, nCols];
        for (int i = 0; i < nRows; i++)
        {
            for (int j = 0; j < nCols; j++)
            {
                matrix[i, j] = Complex.CZero;
            }
        }
    }

    public CMatrix(Complex[,] matrix)
    {
        this.nRows = matrix.GetLength(0);
        this.nCols = matrix.GetLength(1);
        this.matrix = matrix;
    }

    public CMatrix IdentityMatrix()
    {
        CMatrix m = new CMatrix(nRows, nCols);
        for (int i = 0; i < nRows; i++)
        {
            for (int j = 0; j < nCols; j++)
            {
                if (i == j) { m[i, j] = new Complex(1, 0); }
            }
        }
        return m;
    }
```

```
// Accessors
public int GetnRows
{ get { return nRows; } }

public int GetnCols
{ get { return nCols; } }

// Indexers
public Complex this[int m, int n]
{
    get
    {
        if (m < 0 || m > nRows)
        {
          throw new Exception("m-th row is out of range!");
        }
        if (n < 0 || n > nCols)
        {
          throw new Exception("n-th col is out of range!");
        }
        return matrix[m, n];
    }
    set { matrix[m, n] = value; }
}
// Override Methods
public override string ToString()
{
    string strMatrix = "(";
    for (int i = 0; i < nRows; i++)
    {
        string str = "";
        for (int j = 0; j < nCols - 1; j++)
        {
            str += matrix[i, j].ToString() + ", ";
        }
        str += matrix[i, nCols - 1].ToString();
        if (i != nRows - 1 && i == 0)
            strMatrix += str + "\n";
        else if (i != nRows - 1 && i != 0)
            strMatrix += " " + str + "\n";
        else
            strMatrix += " " + str + ")";
    }
    return strMatrix;
}

public override bool Equals(object obj)
{ return (obj is CMatrix) && this.Equals((CMatrix)obj); }

public bool Equals(CMatrix cm)
{ return matrix == cm.matrix; }

public override int GetHashCode()
{ return matrix.GetHashCode(); }
```

```
public static bool operator ==(CMatrix cm1, CMatrix cm2)
{ return cm1.Equals(cm2); }

public static bool operator !=(CMatrix cm1, CMatrix cm2)
{ return !cm1.Equals(cm2); }

public static CMatrix operator +(CMatrix cm)
{ return cm; }

public static CMatrix operator +(CMatrix cm1, CMatrix cm2)
{
    if (!CMatrix.CompareDimension(cm1, cm2))
    {
      throw new Exception("The dimensions of 2 matrices must be
          the same!");
    }
    CMatrix result = new CMatrix(cm1.GetnRows, cm1.GetnCols);
    for (int i = 0; i < cm1.GetnRows; i++)
    {
        for (int j = 0; j < cm1.GetnCols; j++)
        {
            result[i, j] = cm1[i, j] + cm2[i, j];
        }
    }
    return result;
}

public static CMatrix operator +(CMatrix cm, Complex cn)
{
    CMatrix result = new CMatrix(cm.GetnRows, cm.GetnCols);
    for (int i = 0; i < cm.GetnRows; i++)
    {
        for (int j = 0; j < cm.GetnCols; j++)
        {
            result[i, j] = cm[i, j] + cn;
        }
    }
    return result;
}

public static CMatrix operator +(Complex cn, CMatrix cm)
{
    CMatrix result = new CMatrix(cm.GetnRows, cm.GetnCols);
    for (int i = 0; i < cm.GetnRows; i++)
    {
        for (int j = 0; j < cm.GetnCols; j++)
        {
            result[i, j] = cm[i, j] + cn;
        }
    }
    return result;
}
```

```
public static CMatrix operator -(CMatrix cm)
{
    for (int i = 0; i < cm.GetnRows; i++)
    {
        for (int j = 0; j < cm.GetnCols; j++)
        {
            cm[i, j] = -cm[i, j];
        }
    }
    return cm;
}

public static CMatrix operator -(CMatrix cm1, CMatrix cm2)
{
    if (!CMatrix.CompareDimension(cm1, cm2))
    {
      throw new Exception("The dimensions of two matrices must be
           the same!");
    }
    CMatrix result = new CMatrix(cm1.GetnRows, cm1.GetnCols);
    for (int i = 0; i < cm1.GetnRows; i++)
    {
        for (int j = 0; j < cm1.GetnCols; j++)
        {
            result[i, j] = cm1[i, j] - cm2[i, j];
        }
    }
    return result;
}

public static CMatrix operator -(CMatrix cm, Complex cn)
{
    CMatrix result = new CMatrix(cm.GetnRows, cm.GetnCols);
    for (int i = 0; i < cm.GetnRows; i++)
    {
        for (int j = 0; j < cm.GetnCols; j++)
        {
            result[i, j] = cm[i, j] - cn;
        }
    }
    return result;
}

public static CMatrix operator -(Complex cn, CMatrix cm)
{
    CMatrix result = new CMatrix(cm.GetnRows, cm.GetnCols);
    for (int i = 0; i < cm.GetnRows; i++)
    {
        for (int j = 0; j < cm.GetnCols; j++)
        {
            result[i, j] = cn - cm[i, j];
        }
    }
    return result;
}
```

```
public static CMatrix operator *(CMatrix cm, Complex cn)
{
    CMatrix result = new CMatrix(cm.GetnRows, cm.GetnCols);
    for (int i = 0; i < cm.GetnRows; i++)
    {
        for (int j = 0; j < cm.GetnCols; j++)
        {
            result[i, j] = cm[i, j] * cn;
        }
    }
    return result;
}

public static CMatrix operator *(Complex cn, CMatrix cm)
{
    CMatrix result = new CMatrix(cm.GetnRows, cm.GetnCols);
    for (int i = 0; i < cm.GetnRows; i++)
    {
        for (int j = 0; j < cm.GetnCols; j++)
        {
            result[i, j] = cm[i, j] * cn;
        }
    }
    return result;
}

public static CMatrix operator *(CMatrix cm1, CMatrix cm2)
{
    if (cm1.GetnCols != cm2.GetnRows)
    {
        throw new Exception("# columns of the matrix 1 must = #
            columns of the matrix 2");
    }

    Complex ctmp;
    CMatrix result = new CMatrix(cm1.GetnRows, cm2.GetnCols);

    for (int i = 0; i < cm1.GetnRows; i++)
    {
        for (int j = 0; j < cm2.GetnCols; j++)
        {
            ctmp = result[i, j];
            for (int k = 0; k < result.GetnRows; k++)
            {
                ctmp += cm1[i, k] * cm2[k, j];
            }
            result[i, j] = ctmp;
        }
    }
    return result;
}

public static CMatrix operator /(CMatrix cm, Complex cn)
{
    CMatrix result = new CMatrix(cm.GetnRows, cm.GetnCols);
    for (int i = 0; i < cm.GetnRows; i++)
```

```
        {
            for (int j = 0; j < cm.GetnCols; j++)
            {
                result[i, j] = cm[i, j] / cn;
            }
        }
        return result;
    }

    public static CMatrix operator /(Complex cn, CMatrix cm)
    {
        CMatrix result = new CMatrix(cm.GetnRows, cm.GetnCols);
        for (int i = 0; i < cm.GetnRows; i++)
        {
            for (int j = 0; j < cm.GetnCols; j++)
            {
                result[i, j] = cm[i, j] / cn;
            }
        }
        return result;
    }

    // Methods
    // Checks for a square matrix where #rows = #cols
    public bool IsSquared()
    {
        if (nRows == nCols)
            return true;
        else
            return false;
    }

    // Compares the dimension of two complex matrices
    public static bool CompareDimension(CMatrix cm1, CMatrix cm2)
    {
        if (cm1.GetnRows == cm2.GetnRows && cm1.GetnCols == cm2.
            GetnCols)
            return true;
        else
            return false;
    }

    // Makes a clone copy of a complex matrix
    public CMatrix Clone()
    {
        CMatrix cm = new CMatrix(matrix);
        cm.matrix = (Complex[,])matrix.Clone();
        return cm;
    }

    object ICloneable.Clone()
    {
        return Clone();
    }
```

```
// Sets up a call to calculate the transpose of a complex matrix
public CMatrix GetTranspose()
{
    CMatrix ct = this;
    ct.Transpose();
    return ct;
}

// Calculates the transpose of a complex matrix
public void Transpose()
{
    CMatrix cm = new CMatrix(nCols, nRows);
    for (int i = 0; i < nRows; i++)
    {
        for (int j = 0; j < nCols; j++)
        {
            cm[j, i] = matrix[i, j];
        }
    }
    this = cm;
}

// Calculates the trace of a complex matrix
public Complex GetTrace()
{
    Complex sum_of_diag = Complex.CZero;
    for (int i = 0; i < nRows; i++)
    {
        for (int j = 0; j < nCols; j++)
        {
            if (i == j)
                sum_of_diag += matrix[i, j];
        }
    }
    return sum_of_diag;
}


// Extracts a row vector from a complex matrix at specified row
public CVector GetRowCVector(int m)
{
    if (m < 0 || m > nRows)
    {
        throw new Exception("m-th row is out of range!");
    }
    CVector RowCVector = new CVector(nCols);
    for (int i = 0; i < nCols; i++)
    {
        RowCVector[i] = matrix[m, i];
    }
    return RowCVector;
}


// Extracts a column vector from a complex matrix at a
// specified column
```

```
public CVector GetColCVector(int m)
{
    if (m < 0 || m > nCols)
    { throw new Exception("n-th col is out of range!"); }
    CVector ColCVector = new CVector(nRows);
    for (int i = 0; i < nRows; i++)
    {
        ColCVector[i] = matrix[i, m];
    }
    return ColCVector;
}

// Swaps specificed complex matrix row with another row
public CMatrix SwapCMatrixRow(int m, int n)
{
    Complex ctemp = Complex.CZero;
    for (int i = 0; i < nCols; i++)
    {
        ctemp = matrix[m, i];
        matrix[m, i] = matrix[n, i];
        matrix[n, i] = ctemp;
    }
    return new CMatrix(matrix);
}

// Swaps specificed complex matrix column with another column
public CMatrix SwapCMatrixColumn(int m, int n)
{
    Complex ctemp = Complex.CZero;
    for (int i = 0; i < nRows; i++)
    {
        ctemp = matrix[i, m];
        matrix[i, m] = matrix[i, n];
        matrix[i, n] = ctemp;
    }
    return new CMatrix(matrix);
}

// Calculates the transform of a complex matrix
public static CVector CTransform(CMatrix cm, CVector cv)
{
  CVector result = new CVector(cv.GetCVectorSize);
  if (!cm.IsSquared())
  {throw new Exception("The matrix must be squared!");}
  if (cm.GetnCols != cv.GetCVectorSize)
  {throw new Exception("Vector size must = # rows in matrix");}
  for (int i = 0; i < cm.GetnRows; i++)
  {
    result[i] = Complex.CZero;
    for (int j = 0; j < cm.GetnCols; j++)
    {
      result[i] += cm[i, j] * cv[j];
    }
  }
  return result;
}
```

```
public static CVector CTransform(CVector cv, CMatrix cm)
{
  CVector result = new CVector(cv.GetCVectorSize);
  if (!cm.IsSquared())
  {throw new Exception("The matrix must be squared!");}
  if (cm.GetnRows != cv.GetCVectorSize)
  {throw new Exception("Vector size must = # rows in matrix");}
  for (int i = 0; i < cm.GetnRows; i++)
  {
    result[i] = Complex.CZero;
    for (int j = 0; j < cm.GetnCols; j++)
    {
      result[i] += cv[j] * cm[j, i];
    }
  }
  return result;
}

public static CMatrix CTransform(CVector cv1, CVector cv2)
{
  if (cv1.GetCVectorSize != cv2.GetCVectorSize)
  {throw new Exception("The vectors must have the same size!");}
  CMatrix result = new CMatrix(cv1.GetCVectorSize,
                               cv1.GetCVectorSize);
  for (int i = 0; i < cv1.GetCVectorSize; i++)
  {
    for (int j = 0; j < cv1.GetCVectorSize; j++)
    {
      result[j, i] = cv1[i] * cv2[j];
    }
  }
  return result;
}

// Calculates the determinant of a complex matrix
public static Complex Determinant(CMatrix cm)
{
  Complex result = new Complex(0.0, 0.0);
  if (!cm.IsSquared())
  { throw new Exception("The matrix must be squared!"); }
  if (cm.GetnRows == 1)
      result = cm[0, 0];
  else
  {
    for (int i = 0; i < cm.GetnRows; i++)
    {
      result +=
      Math.Pow(-1,i)*cm[0,i]*Determinant(CMatrix.Minor(cm,0,i));
    }
  }
  return result;
}

// Calculates the minor of a complex matrix at a specified row
// and column
```

```
public static CMatrix Minor(CMatrix cm, int row, int col)
{
    CMatrix cmm = new CMatrix(cm.GetnRows - 1, cm.GetnCols - 1);
    int ii = 0, jj = 0;
    for (int i = 0; i < cm.GetnRows; i++)
    {
        if (i == row) continue;
        jj = 0;
        for (int j = 0; j < cm.GetnCols; j++)
        {
            if (j == col) continue;
            cmm[ii, jj] = cm[i, j];
            jj++;
        }
        ii++;
    }
    return cmm;
}

// Calculates the adjoint of a complex matrix
public static CMatrix Adjoint(CMatrix cm)
{
  if (!cm.IsSquared())
  {
    throw new Exception("The matrix must be squared!");
  }
  CMatrix ma = new CMatrix(cm.GetnRows, cm.GetnCols);
  for (int i = 0; i < cm.GetnRows; i++)
  {
    for (int j = 0; j < cm.GetnCols; j++)
    {
      ma[i,j] = Math.Pow(-1,i+j)*(Determinant(Minor(cm,i,j)));
    }
  }
  return ma.GetTranspose();
}

// Calculates the inverse of a complex matrix
public static CMatrix Inverse(CMatrix cm)
{
  if (Determinant(cm) == new Complex(0, 0))
  {
    throw new Exception("Cannot inverse a matrix with 0
        determinant!");
  }
  return (Adjoint(cm) / Determinant(cm));
}

// Replaces the n-th row of a complex matrix with
// contents of a complex vector
public CMatrix ReplaceCRow(CVector cv, int m)
{
    if (m < 0 || m > nRows)
    {
        throw new Exception("m-th row is out of range!");
    }
```

```
        if (cv.GetCVectorSize != nCols)
        {
            throw new Exception("Vector size is out of range!");
        }
        for (int i = 0; i < nCols; i++)
        {
            matrix[m, i] = cv[i];
        }
        return new CMatrix(matrix);
    }

    // Replaces the n-th column of a complex matrix with
    // contents of a complex vector
    public CMatrix ReplaceCCol(CVector cv, int n)
    {
        if (n < 0 || n > nCols)
        {
            throw new Exception("n-th col is out of range!");
        }
        if (cv.GetCVectorSize != nRows)
        {
            throw new Exception("Vector size is out of range!");
        }
        for (int i = 0; i < nRows; i++)
        {
            matrix[i, n] = cv[i];
        }
        return new CMatrix(matrix);
    }
}
```

4.8 Generic vs. Non-Generic Coding

One of the main advantages of implementing generics is that data type information is retained until runtime, thus making it possible to reduce code duplication of the same data structure for different data types. Because of their rich mathematical structure, scientific and engineering applications seem, at first, ideally well suited for applying generic data types. However, there are some additional significant issues that arise when using generics for programming numerical applications that merit some attention.

Unfortunately, at the time of this writing there are still some technical as well as performance issues associated with the use of generics in C# for numerically intensive computer calculations. As a result, feasibility and performance studies have even been done on the suitability of using generics for scientific computing in various programming languages, including C#. In particular, Dragan and Watt [27] have pointed out that the implementation of generics in current C# compilers must be improved before generic coding methods can be used effectively in numerically intensive scientific and engineering applications. To illustrate a good example of the major source of this problem, let's start by examining a simple program. A generic equivalent version of the complex number library that was just described might start like this:

```
using System;
using System.Collections.Generic;

namespace GenericComplexNumberLibrary
{
  public struct Complex<T> where T: struct
  {
    private T real;
    private T imag;

    public T Real
    {
      get { return real; }
      set { real = value; }
    }

    public T Imag
    {
      get { return imag; }
      set { imag = value; }
    }

    public Complex(T x, T y)
    {
      this.real = x;
      this.imag = y;
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      Complex<Int32> z = new Complex<Int32>(3, 7);
      Console.WriteLine("z = {0} + i{1}", z.Real, z.Imag);
      Console.ReadLine();
    }
  }
}
```

The code above compiles just fine and the output $z = 3 + i7$ will be displayed on the monitor screen. However, if we continue to expand this library by adding a simple generic function, such as the one for calculating the norm of a complex number:

```
public T CNorm
{
  return Math.Sqrt( real * real + imag * imag );
}
```

then the program will no longer compile and an error message will be displayed on the screen instead. This problem comes about because type parameters without any constraints are assumed to be of the `System.Object` type. As a result, the compiler does not know *how* to perform arithmetic operations on two arbitrary objects. Ideally, there should be some way to constrain the type parameter `T` so that it has the necessary computational support for implementing at least the basic arithmetic operators $+$, $-$, $/$, and $*$. Unfortunately, there currently exists no practical way to constrain type parameters by requiring the existence of certain operators or methods. The only way to constrain type parameters is by requiring the type to inherit a base class or to implement an interface. Actually, there is one special case of a method constraint, the `new()` constraint, which requires the existence of a parameterless constructor but this is completely useless in this case. In addition, interface constraints are a bit limited because interfaces cannot contain static methods or operators. Nevertheless, an interface for types that support all basic arithmetic operations might look like this:

```
interface IArithmetic<T>
{
  T Add(T x);
  T Subtract(T x);
  T Multiply(T x);
  T Divide(T x);
}
```

However, one has to also consider the existence of some types like `System.String` which also support addition, but in a different context than the standard arithmetic addition operation, and so this approach only serves to complicate matters further. Nevertheless, one possible solution is to let the basic data types inherit an interface for arithmetic operations, similar to the `IComparable<T>` interface constraint that is used by the `System.Collections.Generic` namespace, and then wrap the basic data types to make them support the required interface. A more radical approach involves supporting method constraints instead of just interface constraints. However, this approach requires the programmer to specify the exact signature of the method desired. Again, it's impossible for the compiler to know in advance if the type given for `T` in a constructed type or if at some point in the future it even supports the arithmetic operators. As a result, a common technique is to externalize the operation from the `Complex<T>` definition itself and then require the user of `Complex<T>` to provide code for the particular desired arithmetic operation. Nash [28] provides some very nice examples of using this technique. However, this approach requires writing quite a lot of additional code which will, therefore, very likely result in undesired slower performance. A quick search on the Internet at the time of this writing has revealed that Microsoft seems to be aware of these problems and will likely provide some sort of solution in a future release of their C# compiler.

5

Sorting and Searching Algorithms

5.1 Introduction

The .NET Framework Class Library provides several classes, called collections, that are used to store groups of related objects. Along with methods for organizing, storing and retrieving data, these classes also provide methods for sorting and searching that require no additional programming and thus can substantially reduce application development time. For example, the `List<T>` class is a generic class found in the `System.Collections.Generic` namespace and the `ArrayList` class is found in the `System.Collections` namespace. Both of these classes have properties that are very similar to C# arrays and, in addition, also come with their own methods for performing efficient sorting and searching. However, one key advantage of using collection classes over conventional arrays is that collections can dynamically grow and shrink as their number of elements change. Arrays, on the other hand, do not automatically adjust their size at runtime to accommodate changes in their initial number of alloted elements unless the programmer manually codes in a new array or uses the array class's `Resize` method. With all these tools at our disposal, one may very well question the wisdom of going through the effort of discussing the various different types of sorting and searching algorithms. First, instead of just randomly choosing any sorting or searching algorithm, it is important to have a basic general understanding of how all these different algorithms actually work in order to make a good decision about which algorithm to use in a particular application. After all, there is no known ideal algorithm for either sorting or searching that meets all the requirements for optimal speed and efficiency. As the size and type of input data changes, different sorting and searching algorithms offer different degrees of strengths and weaknesses. Furthermore, certain applications may require some type of sorting and/or searching as part of their internal structure. As a result, at some point programmers may be required to write their own custom set of sorting and/or searching routines instead of simply relying on an unfamiliar canned version of one or more of these algorithms. Finally, because of their importance, sorting and searching algorithms are also an integral part of the standard curriculum for both elementary and advanced computer science courses.

5.2 Sorting Algorithms

A sorting algorithm is essentially a recipe containing detailed computer code instructions for organizing the elements of a list into a well-defined numerical or alphabetical order. A list is an abstract concept consisting of a finite collection of fixed-length entities that can be arranged either in random order or in an increasing or decreasing sequential order. In practice, a list is usually expressed in the form of an array or a more advanced data structure such as a linked list. Sorting is often used in conjunction with the processing of either experimentally measured or computer generated data. In addition, sorting is sometimes used by other algorithms, such as search and merge algorithms, whose own optimization require sorted lists to work correctly and efficiently. Because of its frequent use in a wide range of engineering, mathematical and scientific applications, sorting has attracted a lot of research interest going as far back as to the earliest days of computing. Sorting often involves large volumes of data and so research into this topic has primarily focused on developing increasingly fast and efficient algorithms that strive to minimize both the computer processing time involved and the amount of computer memory needed. Although many consider sorting to be a solved problem, new interesting and useful sorting algorithms are still being invented [29, 30] and so it is still very much a vibrant evolving subject matter that is worth covering.

Since there exists a large number of sorting algorithms, there also exists a number of important considerations that ought to be taken into account before selecting one of these algorithms for use in a particular application. The various sorting algorithms available today are often classified by a wide variety of different factors that among which include their degree of computational complexity, their internal structure, their stability, and their efficient use of computer resources. Whatever the case, the final output from a sorting process must be either in a decreasing or increasing order obtained by a permutation, or reordering, of the original input data.

Perhaps the primary parameter of interest used in evaluating the quality and efficiency of sorting algorithms is their running time. After all, we live in a world where time is money and most people just want everything done as quickly as possible. Computational complexity refers to the theoretical calculation and estimation of the worst, average and best running times required to sort a list of n records. The level of complexity is usually expressed by the big-Oh notation which is just an abbreviation for the phrase "of the order of". The simplest sorting algorithms typically require a running time that is proportional to n^2 in order to sort n records and so their level of complexity is expressed by $O(n^2)$. It can also be shown [31] that no algorithm that sorts by comparing elements can perform any better than $O(n \log n)$ in the average or worst case. The ideal sorting algorithm, of course, would require only one pass to sort a list of n records and so it would have an order of complexity given by $O(n)$. However, these figures are simply theoretical approximations. In practice, the actual running time also depends on a host of other factors. For example, slight modifications of the internal structure of some of these sorting algorithms by introducing recursion where possible have sometimes led to substantial improvements in their original running time [31, 32]. As a result, some sorting algorithms are either recursive or non-recursive, while others, such as in the case of Merge sort, may contain features from both. A recursive algorithm is an algorithm which calls itself with increasingly smaller input values, and which obtains the result for the current input by repeatedly applying operations to the returned value for the smaller input until a final solution is finally obtained.

The internal structure of sorting algorithms is also an important factor in determining the effectiveness of its overall performance. Sorting algorithms can be broadly classified as comparison and non-comparison based sorts to explicitly indicate how the sorting is actually accomplished. The most common approach used for sorting is called in-place, comparison based sort. In-place sorting means that, in order to save memory, the algorithm does not allow for the use of any additional storage space aside from that which has already been set aside for the items being sorted. Comparison based sorting means that in the sorting algorithm there exists a function for comparing two elements, say x and y, from the input data list that can only tell if $x < y$, $x > y$ or $x = y$ without providing any additional information. Sorting is then attained by essentially comparing and then, if needed, swapping two elements at a time following some clever scheme until the entire list is eventually sorted. Some of the most well known comparison sorts include Quicksort, Heapsort, Mergesort, Introsort, Insertion sort, Selection sort and Bubble sort. Non-comparison based sorts, on the other hand, are sorting algorithms that assume that one can extract some ordinal information in the keys, and then use that information to improve the efficiency of the algorithm itself. Some examples of non-comparison sort algorithms include Radix sort, Counting sort and Bucket sort. However, some of the more advanced sorting algorithms actually employ a combination of different sorting methods and so grouping them in this manner is only meant to provide a helpful tool for remembering how all these various algorithms operate internally.

Sorting can also be done on complex records consisting of several different fields provided that at least one or more of the internal components is chosen to be the sort key. Stable sorting maintains the relative order of records with equal sort keys. That is, a sorting algorithm is said to be stable if whenever there are two records X and Y with the same key and with X appearing before Y in the original list, X will appear before Y in the sorted list. Otherwise the sorting is said to be unstable.

Advanced computer resources, such as those with multiple CPUs, may allow programmers to do parallel sorting with divide-and-conquer style algorithms which are generally much faster than the traditional sequential sorting algorithms by computers with just a single CPU. However, parallel sorting requires the use of specialized coding, such as threading, which may create additional unwanted problems that can also potentially lead to an increase in development time. Although we usually want the input data sorted, there may be times where we may just want access to a sorted version of that data while keeping the original input data unsorted. Some sorting algorithms naturally work in-place while other algorithms can provide a sorted copy of the input data or at least a list of indexes which index the original items in a sorted order.

If the data to be sorted fits completely in computer memory, then the sorting method is said to be internal otherwise it is said to be external. As expected, internal sorting is usually preferred as it tends to run much faster than external sorting. However, if the amount of input data is very large so that it will not all fit inside the available computer memory, then the slower external sorting may be required.

Also important to consider is the type of data structure that will be used for storing and manipulating the input and output data. For example, sorting can be done on lists, arrays or linked lists to name just a few. Although all these different data structures can be used to sort data efficiently, some sorting algorithms work better and more efficiently with certain specific data structures than with others.

Another important point to consider is whether the sorting will be done all at once or in incremental steps. Sorting all the data at once may not always be possible or even desirable. For example, if the input data is being collected over time, you may want to periodically sort whatever data has been collected during each specified time interval and then follow up with a final sort of all the individually collected data sets at the end. Since the data sets are already partially sorted, the final sorting process will generally be much faster than trying to sort the entire data set from scratch once the input data stream has ended.

After considering all these various issues, it seems reasonable to conclude that the ideal sorting algorithm would (1) be stable so that equal keys are not re-ordered, (2) operate in place requiring only $O(1)$ extra memory space, (3) have a worst case $O(n \log n)$ key comparisons, (4) have a worst case $O(n)$ swaps and (5) be adaptive so that it can speed up to $O(n)$ when the data is nearly sorted, inversely sorted or when there are few unique keys to be sorted. Unfortunately, there is no ideal sorting algorithm that meets all these requirements. Moreover, their individual behavior is not necessarily a definitive deciding factor in choosing the best algorithm for use in a particular project. Instead, selecting the optimal sorting algorithm ultimately depends on a complex combination of several additional factors that among which include the initial state and the amount of data to be sorted. For example, some sorting algorithms are better suited than others to handle different volumes of data. Also, the initial input data may be completely random, partially or nearly sorted, completely reversed or contain a number of unique keys. Consequently, all these features can have a substantial impact on the actual running time of a sorting algorithm.

Quicksort [32] is perhaps the most popular sorting algorithm known to exist and it only makes an average of $O(n \log n)$ comparisons to sort n elements. Unfortunately, even this fast sorting algorithm has its own list of advantages and disadvantages. As a result, many other sorting algorithms of various degrees of complexity and efficiency have been discovered and analyzed. Therefore, it makes sense to develop a firm understanding of the different sorting methods that are available including their respective advantages and disadvantages. For completeness and pedagogical purposes, the material covered in this chapter will include the basic highlights of not just the most popular but also some of the other lesser known sorting algorithms. Every algorithm discussed in this chapter will be immediately followed up by a code snippet to illustrate its implementation in C#. A more complete coverage of all kinds of algorithms can be found in Sedgewick [32], Knuth [33] or Cormen [31].

5.3 Comparison Sorts

5.3.1 Bubble Sort

Bubble sort is perhaps the easiest and best known sorting algorithm because of its intuitive and straightforward simplicity. Bubble sort works by stepping through the entire list to be sorted while comparing two items at a time and swapping their positions if they are found to be in the wrong order. This process is repeated until no swaps are needed thereby indicating that the list has been sorted. The algorithm gets its name from the way smaller elements seem to *bubble* to the top of the list. In fact, one of the many performance problems with the Bubble sort algorithm is the so called rabbit-turtle effect where large values at the bottom of the list seem to bubble up very quickly (rabbits) but small values at the top of the list seem to require many passes before they sink to the bottom of the list (turtles). Another performance problem with the Bubble sort algorithm is that it has an average and worst case complexity of $O(n^2)$ and so it is generally highly inefficient, particularly for large data sets. However, if the input data is already nearly sorted so that the algorithm needs to make, say only 1 pass, then Bubble sort could also have a best case complexity of just $O(n)$. Because of all these issues, the Bubble sort algorithm is rarely used in practice except in introductory computer science courses. Nevertheless, a basic Bubble sort algorithm can be implemented in C# as shown below.

```
static void bubbleSort1(ref int[] x)
{
    bool exchanges;
    do
    {
        exchanges = false;
        for (int i = 0; i < x.Length - 1; i++)
        {
            if (x[i] > x[i + 1])
            {
                // Exchange elements
                int temp = x[i];
                x[i] = x[i + 1];
                x[i + 1] = temp;
                exchanges = true;
            }
        }
    } while (exchanges);
}
```

In spite of its inefficiency, the original Bubble sort algorithm has evolved in order to help improve its performance. All these changes, however, still have one thing in common in that the modified algorithm continues to compare only adjacent pairs of elements and so these new variations of the algorithm still retain an undesired and inefficient complexity of $O(n^2)$. One way to improve the running time of the Bubble sort algorithm is to note that each inner loop is one shorter than the previous one because the largest items are being moved towards the end of the list. Given a list of size *n*, the *n*th element will always be guaranteed to be in its proper place and so it suffices to sort just the remaining $n - 1$ elements. Therefore, this slightly improved version of Bubble sort makes a fixed number of passes over the list to be sorted and can be implemented in C# as shown below.

```
static void bubbleSort2(ref int[] x)
{
    for (int pass = 1; pass < x.Length - 1; pass++)
    {
        // Count how many times this next looop
        // becomes shorter and shorter
        for (int i = 0; i < x.Length - pass; i++)
        {
            if (x[i] > x[i + 1])
            {
                // Exchange elements
                int temp = x[i];
                x[i] = x[i + 1];
                x[i + 1] = temp;
            }
        }
    }
}
```

The basic concepts behind the two different versions of the Bubble sort algorithm that have been presented so far may be combined together to form a still better algorithm. In this new situation, the loop stops when there are no more swaps and also sorts a smaller range of items with each iteration. A C# implementation of this variation of the Bubble sort algorithm is given below.

```
static void bubbleSort3(ref int[] x)
{
    bool exchanges;
    int n = x.Length;
    do
    {
        n--; // Make loop smaller each time.
        // and assume this is last pass over array
        exchanges = false;
        for (int i = 0; i < x.Length-1; i++)
        {
            if (x[i] > x[i + 1])
            {
                // Exchange elements
                int temp = x[i];
                x[i] = x[i + 1];
                x[i + 1] = temp;
                exchanges = true;
            }
        }
    } while (exchanges);
}
```

Another variation of the Bubble sort algorithm can be obtained by noticing that after the first pass, it's only necessary to sort the list from the position just below the first swap, since small values may move lower, to the position just before the last swap, since largest values won't move higher. Everything that was not swapped must therefore be in the correct order. As a result, after each pass the upper and lower bounds for the next pass are set from the positions of the first and last swaps on the previous pass. Below is a C# implementation of this improved version of the Bubble sort algorithm that, on each pass, looks only at the region of the list where more swaps might be necessary. A C# implementation for this variation of the Bubble sort algorithm is given below.

```
static void bubbleSortRange(ref int[] x)
{
  int lowerBound = 0; // First position to compare.
  int upperBound = x.Length-1; // First position NOT to compare.
  int n = x.Length-1;

  // Continue making passes while there is a potential exchange.
  while (lowerBound <= upperBound)
  {
    // assume impossibly high index for low end.
    int firstExchange = n;
    // assume impossibly low index for high end.
    int lastExchange  = -1;

    // Make a pass over the appropriate range.
    for (int i=lowerBound; i<upperBound; i++)
    {
      if (x[i] > x[i+1])
      {
        // Exchange elements
        int temp = x[i];
        x[i] = x[i+1];
        x[i+1] = temp;
        // Remember first and last exchange indexes.
        if (i<firstExchange)
        { // True only for first exchange.
          firstExchange = i;
        }
        lastExchange = i;
      }
    }

    //--- Prepare limits for next pass.
    lowerBound = firstExchange-1;
    if (lowerBound < 0)
    {
      lowerBound = 0;
    }
    upperBound = lastExchange;
  }
}
```

5.3.2 Cocktail Sort

The Cocktail sort, also known as the Bi-directional Bubble sort, the Shaker sort, the Ripple sort, the Shuttle sort, the Children sort and the Happy Hour sort, is just another slightly improved variation of the fundamental Bubble sort algorithm. The difference between the Cocktail and the Bubble sort algorithms is that instead of repeatedly iterating through an input list from bottom to top, the Cocktail sort iterates alternating from bottom to top and then from top to bottom. By performing bi-directional iterations, the Cocktail sort can achieve a slightly better performance time than the standard Bubble sort algorithm which only iterates through the input list in one direction and therefore can only reposition items by one step per iteration. A C# implementation of the Cocktail sort algorithm is given below.

```
static void CocktailSort(ref int[] x)
{
    for (int k = x.Length - 1; k > 0; k--)
    {
        bool swapped = false;
        for (int i = k; i > 0; i--)
            if (x[i] < x[i - 1])
            {
                // swap
                int temp = x[i];
                x[i] = x[i - 1];
                x[i - 1] = temp;
                swapped = true;
            }

        for (int i = 0; i < k; i++)
            if (x[i] > x[i + 1])
            {
                // swap
                int temp = x[i];
                x[i] = x[i + 1];
                x[i + 1] = temp;
                swapped = true;
            }

        if (!swapped)
            break;
    }
}
```

5.3.3 Odd-Even Sort

The Odd-Even sort algorithm works by comparing all odd and even indexed pairs of adjacent items in the input list and, if a pair is found to be in the wrong order, the items are then switched. The next step repeats this process for even and odd indexed pairs of adjacent elements. This algorithm then alternates between odd/even and even/odd steps until the list is completely sorted. A C# implementation of the Odd-Even sort algorithm is given below.

```
static void OddEvenSort(ref int[] x)
{
    int temp;
    for (int i = 0; i < x.Length/2; ++i)
    {
        for (int j = 0; j < x.Length-1; j += 2)
        {
            if (x[j] > x[j+ 1])
            {
                temp = x[j];
                x[j] = x[j + 1];
                x[j + 1] = temp;
            }
        }

        for (int j = 1; j < x.Length-1; j += 2)
        {
            if (x[j] > x[j + 1])
            {
                temp = x[j];
                x[j] = x[j + 1];
                x[j + 1] = temp;
            }
        }
    }
}
```

5.3.4 Comb Sort

The Comb sort algorithm [34] is basically just a modification of the Bubble sort algorithm that exploits the concept of comparing and swapping items that are separated by a gap instead of those adjacent to each other. Although Shell sort is also based on this very same idea, it is a modification of Insertion sort instead of Bubble sort. Comb sort works by iterating several times through the data while comparing pairs of elements and swapping them if they are not in order with respect to each other. The initial gap is usually set to be the size of the input list, but it is then divided by a shrink factor at the end of every iteration until it finally reaches the value of 1 at which point the comb algorithm actually turns into the Bubble sort algorithm for its last pass.

As one might expect, the efficiency of Comb sort greatly depends on the value chosen for the shrink factor. If the shrink factor value is chosen too small, it will slow the algorithm down because then more comparisons must be made. If the shrink factor is chosen too high, then not enough small values near the top end of the input list will migrate down towards the bottom during the sorting process and this will cause a slowdown towards the end when Comb sort turns into Bubble sort. In the original article, the authors suggested the value of 1.3 as the ideal shrink factor and noted that using this value will result in only three possible ways for the list of gaps to end: $(9,6,4,3,2,1),(10,7,5,3,2,1)\text{or}(11,8,6,4,3,2,1)$. Of these three possible choices, only the last one was found to completely eliminate all the small values around the top of the input list before the gap becomes 1.

```
private static int newGap(int gap)
{
    gap = gap * 10 / 13;
    if(gap == 9 || gap == 10)
       gap = 11;
    if(gap < 1)
       return 1;
    return gap;
}

private static void CombSort(ref int[] x)
{
    int gap = x.Length;
    bool swapped;
    do
    {
        swapped = false;
        gap = newGap(gap);
        for (int i = 0; i < (x.Length - gap); i++)
        {
            if(x[i] > x[i + gap])
            {
                swapped = true;
                int temp = x[i];
                x[i] = x[i + gap];
                x[i + gap] = temp;
            }
        }
    } while(gap > 1 || swapped);
}
```

5.3.5 Gnome Sort

The Gnome sort was originally developed by D. Grune [35] and is based on the technique allegedly used by the standard Dutch garden Gnome to sort flower pots. The Gnome sort algorithm works by comparing the current item with the previous one. If they are in order then move on to the next item or stop if the end is reached. If they are not in order, swap them and move to the previous item. If there is no previous item, then move to the next item. The Gnome sort is a sorting algorithm which is similar to insertion sort, except that moving an item to its proper place is accomplished by a series of swaps, as in bubble sort. While conceptually simple to understand, the Gnome sort has a complexity of $O(n^2)$ and is therefore also very inefficient. However, in practice this sorting algorithm has allegedly been reported to run as fast as Insertion sort. A C# implementation of the Gnome sort algorithm is given below.

```
static void GnomeSort(ref int[] x)
{
    int i = 0;
    while (i < x.Length)
    {
        if (i == 0 || x[i - 1] <= x[i]) i++;
        else
        {
            int temp = x[i];
            x[i] = x[i - 1];
            x[--i] = temp;
        }
    }
}
```

5.3.6 Quicksort

Quicksort is arguably the fastest and most popular of all the sorting algorithms known to exist today [32]. Developed in 1962 by C. Hoare [36], Quicksort makes an average of $O(n\log n)$ comparisons to sort n items. Unfortunately and like all the other sorting algorithms, Quicksort also has drawbacks. For example, Quicksort is not stable, makes about $O(n^2)$ comparisons to sort n items in the worst case and, if not implemented correctly, can perform very badly in certain situations. On average, however, Quicksort is significantly faster in practice than almost any other $O(n\log n)$ sorting algorithm.

Quicksort uses a divide-and-conquer method for sorting that starts by first partitioning the input list into two parts. Each partition is then sorted independently by recursively calling itself over and over again until the entire input list is sorted. Careful selection of the pivot point during the partition process is critical to the success or failure of the overall sorting process. Not surprisingly, a general strategy for partitioning the input list exists which, in most but not necessarily all cases, has proven to be very successful. First, an arbitrary pivot point, sometimes also called the partitioning point, is chosen. Then the list is reordered so that all the items which are less than the partitioning point are placed before it and all the items which are greater than the partitioning point are placed after it with equal values going either way. At the completion of this step, the pivot point is now clearly in its final position. Finally, recursively call in sequence this same algorithm to sort the list of items that are less than the pivot point followed by the list of items that are greater than the pivot point. Note that the base case of the recursion are lists of size zero or one, which are always sorted and so by the time this point is reached, the entire input list will have also been sorted. A C# implementation of the Quicksort algorithm is given below.

```
public static void QuickSort(ref int[] x)
{
    qs(x, 0, x.Length - 1);
}

static void qs(int[] x, int left, int right)
{
    int i, j;
    int pivot, temp;

    i = left;
    j = right;
    pivot = x[(left + right) / 2];

    do
    {
        while ((x[i] < pivot) && (i < right)) i++;
        while ((pivot < x[j]) && (j > left)) j--;

        if (i <= j)
        {
            temp = x[i];
            x[i] = x[j];
            x[j] = temp;
            i++; j--;
        }
    } while (i <= j);

    if (left < j) qs(x, left, j);
    if (i < right) qs(x, i, right);
}
```

5.3.7 Insertion Sort

Conceptually, the Insertion sort algorithm works by first creating two list structures: one to hold the input data and the other to store the output data. It then steps through the input list reading each item and inserting it into its proper sorted position in the output list. In practice, however, most implementations of the Insertion sort algorithm use a memory saving in-place sort process that starts by dividing the input array into two partitions: one partition for sorted values and another partition for unsorted values. Initially, only the first element in the list belongs to the sorted partition. Then the first element in the unsorted partition is picked up and inserted into its appropriate position in the sorted partition. The actual insertion takes place by moving the element that was picked up in the unsorted partition past the already sorted elements and then repeatedly swapping it with the preceding element until it is found to be in the appropriate position in the sorted partition. This process is then repeated until all the elements in the unsorted partition have been assigned to their new correct positions in the sorted partition. Although insertion sort is also of the order $O(n^2)$ and is therefore considered inefficient, in actual practice it is faster than either the Bubble or the Selection sort algorithms. As a result, Insertion sort is also often used in conjunction with more sophisticated algorithms. A C# implementation of the Insertion sort algorithm is given below.

```
static void InsertionSort(ref int[] x)
{
    int n = x.Length-1;
    int i, j, temp;

    for (i = 1; i <= n; ++i)
    {
        temp = x[i];
        for (j = i - 1; j >= 0; --j)
        {
            if (temp < x[j]) x[j + 1] = x[j];
            else break;
        }
        x[j + 1] = tmp;
    }
}
```

5.3.8 Shell Sort

The Shell sort algorithm [37] is fundamentally an improvement of the Insertion sort algorithm. The key concept behind the Shell sort algorithm is that it begins by comparing and swapping items that are distant rather than adjacent to each other. This feature allows an item to take longer steps toward its expected final position. As the algorithm loops through the entire input list, the gap between each item steadily decreases until the items being compared and swapped are adjacent to each other. The gap between the numbers being sorted on each pass through the data is called an increment and the Shell sort algorithm is sometimes also called a Diminishing Increment sort. The Shell sort algorithm is not to be confused with Comb sort. The Comb sort algorithm is a modification of Bubble sort whereas Shell sort is a modification of Insertion sort. The original proposed initial increment was $n/2$ where n is the number of records being sorted. However, the resulting sequence $\ldots 8,4,2,1$ was found not to be a good choice for gaps especially if n is a power of 2. As a result, much research went into finding the best possible sequence of increments but, unfortunately, to date the optimum sequence has not yet been found. However, Knuth [33] has proposed an increment sequence that is generated by the following recurrence relation:

$$i_0 = 1, \quad i_{k+1} = 3i_k + 1, \quad k = 0,1,2,\ldots$$

and is regarded to produce the best increment sequence that is currently available for use today. A C# implementation of the Shell sort algorithm is given below.

```
public static void ShellSort(ref int[] x)
{
    int i, j, temp;
    int increment = 3;

    while (increment > 0)
    {
        for (i = 0; i < x.Length; i++)
        {
            j = i;
            temp = x[i];

            while ((j>=increment) && (x[j-increment]>temp))
            {
                x[j] = x[j-increment];
                j = j-increment;
            }

            x[j] = temp;
        }

        if (increment/2 != 0)
        {
            increment = increment/2;
        }
        else if (increment==1)
        {
            increment = 0;
        }
        else
        {
            increment = 1;
        }
    }
}
```

5.3.9 Selection Sort

Selection sort works by first finding the minimum value of the list to be sorted. It then swaps that minimum value with the value found in the first position of the list. It then finds the second smallest value in the list and then swaps it with the value in the second position of the list. This process is continued until the entire list is sorted. This algorithm is called Selection sort because it works by repeatedly selecting the smallest remaining item of the list and then swapping it with the item in the corresponding position of the list. In so doing, the list is effectively split into two parts: one sublist of the items already sorted and another sublist of the items remaining to be sorted. Unlike other sorting algorithms, the running time of selection sort is not affected by the prior ordering of the list because it always performs the same number of operations on a list of n records. Although Selection sort was originally designed to improve the performance of Bubble sort, it also has a complexity of the order of $O(n^2)$ making it inefficient for sorting large lists. However, in certain situations Selection sort has been shown to have some performance advantages over more complicated and allegedly better sorting algorithms.

```
public static void SelectionSort(ref int[] x)
{
    int i, j, min, temp;
    for (i = 0; i < x.Length - 1; i++)
    {
        min = i;
        for (j = i + 1; j < x.Length; j++)
        {
            if (x[j] < x[min])
            { min = j; }
        }
        temp = x[i];
        x[i] = x[min];
        x[min] = temp;
    }
}
```

5.3.10 Merge Sort

The Merge sort algorithm is based on a divide-and-conquer strategy. First, the data to be sorted is divided into two halves. Next, each half is sorted independently and may but need not be sorted recursively. Then the two sorted halves are merged together to form the complete sorted sequence. Merge sort has a computed time complexity of $O(n\log(n))$ to sort n records and is therefore one of the optimal sorting algorithms presently in existence. Below is a C# implementation of the Merge sort algorithm. Unlike the other sort algorithms that can be called by passing just the input data array, the merge sort algorithm is recursive and needs a kick start to get it going. As a result, you need to pass not only the input data array to be sorted but also the initial left and right pivot points, such as in the following function call example: `MergeSort(ref xArray, 0, xArray.Length - 1);`

```
public static void MergeSort(ref int[] x, int left, int right)
{
    if (left < right)
    {
        int middle = (left + right) / 2;
        MergeSort(ref x, left, middle);
        MergeSort(ref x, middle + 1, right);
        Merge(ref x, left, middle, middle + 1, right);
    }
}

public static void Merge(ref int[] x, int left, int middle, int
    middle1, int right)
{
    int oldPosition = left;
    int size = right - left + 1;
    int[] temp = new int[size];
    int i = 0;

    while (left <= middle && middle1 <= right)
    {
        if (x[left] <= x[middle1])
            temp[i++] = x[left++];
        else
            temp[i++] = x[middle1++];
    }
    if (left > middle)
        for (int j = middle1; j <= right; j++)
            temp[i++] = x[middle1++];
    else
        for (int j = left; j <= middle; j++)
            temp[i++] = x[left++];
    Array.Copy(temp, 0, x, oldPosition, size);
}
```

5.3.11 Bucket Sort

Bucket sorting in C# consists of taking an array of elements that have some sort of numeric value. Each element is stored in a conceptual *bucket*, using the value as an index. When the bucket is emptied, the result will be a sorted list in order. The space requirements for n elements is n and the running time can be characterized as $O(n)$ since elements are directly being stored in a bucket. However, one must be careful to account for the possibility of duplicate elements. This means that the bucket cannot directly store just values because there would be no way to tell exactly how many of each value there is. The solution is to make the bucket an array of `List<>` items. That way elements are added to a list at index `i`. The Bucket sort algorithm can therefore be summarized as follows.

- Find the maximum and minimum values in the array.
- Initialize a bucket array of `List<>` elements with the size given by `maxValue-minValue+1`.
- Move elements in array to the bucket.
- Write the bucket out, in order, to the original array.

As is, the original version of the Bucket sort algorithm contains some minor flaws which can fortunately be easily fixed. In practice, for example, some buckets may go completely unused and thus waste some valuable computer resources. One way to avoid this problem is avoid initializing buckets unless it is actually necessary. This can be easily accomplished by first checking to see if a bucket is null before actually initializing it. The second improvement, is to use a `LinkedList` instead of `List`. Although this gives only a slight improvement in speed, it is an improvement nonetheless. In any event, a C# implementation of the Bucket sort algorithm is presented below.

```
public static void BucketSort(ref int[] x)
{
   //Verify input
   if (x == null || x.Length <= 1)
        return;

   //Find the maximum and minimum values in the array
   int maxValue = x[0];
   int minValue = x[0];

   for (int i = 1; i < x.Length; i++)
   {
       if (x[i] > maxValue)
           maxValue = x[i];
       if (x[i] < minValue)
           minValue = x[i];
   }

   //Create a temporary "bucket" to store the values in order
   //each value will be stored in its corresponding index
   //scooting everything over to the left as much as possible.
   LinkedList<int>[] bucket =
                new LinkedList<int>[maxValue-minValue+1];

   //Move items to bucket
   for (int i = 0; i < x.Length; i++)
   {
       if (bucket[x[i] - minValue] == null)
           bucket[x[i] - minValue] = new LinkedList<int>();

       bucket[x[i] - minValue].AddLast(x[i]);
   }

   //Move items in the bucket back into the
   //original array in order
   int k = 0; //index for original array
   for (int i = 0; i < bucket.Length; i++)
   {
      if (bucket[i] != null)
      {
         //start add head of linked list
         LinkedListNode<int> node = bucket[i].First;

         while (node != null)
         {
            //get value of current linked node
            x[k] = node.Value;
            //move to next linked node
            node = node.Next;
            k++;
         }
      }
   }
}
```

5.3.12 Heap Sort

Heapsort is an in-place, comparison-based sorting algorithm that has the advantage of a worst-case $O(n\log n)$ runtime. Heapsort begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full. A C# implementation of the Heap sort algorithm is presented below.

```
public static void Heapsort(ref int[] x)
{
    int i;
    int temp;
    int n = x.Length;

    for (i = (n/2)-1; i >= 0; i--)
    { siftDown(ref x, i, n); }
    for (i = n-1; i >= 1; i--)
    {
        temp = x[0];
        x[0] = x[i];
        x[i] = temp;
        siftDown(ref x, 0, i-1);
    }
}

public static void siftDown(ref int[] x,int root,int bottom)
{
    bool done = false;
    int maxChild;
    int temp;

    while ((root * 2 <= bottom) && (!done))
    {
        if (root * 2 == bottom)
            maxChild = root * 2;
        else if (x[root * 2] > x[root * 2 + 1])
            maxChild = root * 2;
        else
            maxChild = root * 2 + 1;

        if (x[root] < x[maxChild])
        {
            temp = x[root];
            x[root] = x[maxChild];
            x[maxChild] = temp;
            root = maxChild;
        }
        else
        {
            done = true;
        }
    }
}
```

5.4 Count Sort

The idea underlying the Count sort algorithm is that each key data can indirectly work an initial index for itself. The routine parses the original array determining its minimum and maximum value, then builds a temporary array with one element for each possible value of the key. The routine parses the original array again, this time counting the occurrences of each distinct value. After this step, it is very easy to evaluate where each value has to go in the definitive, sorted array, and this can be accomplished in the third step. A C# implementation of the count sort algorithm is presented below.

```
public static void Count_Sort(ref int[] x)
{
   try
   {
      int i = 0;
      int k = FindMax(x);

      // output array holds the sorted output
      int[] output = new int[x.Length];

      // provides temperarory storage
      int[] temp = new int[k + 1];
      for (i = 0; i < k + 1; i++)
      {
         temp[i] = 0;
      }

      for (i = 0; i < x.Length; i++)
      {
         temp[x[i]] = temp[x[i]] + 1;
      }

      for (i = 1; i < k + 1; i++)
      {
         temp[i] = temp[i] + temp[i - 1];
      }

      for (i = x.Length - 1; i >= 0; i--)
      {
         output[temp[x[i]] - 1] = x[i];
         temp[x[i]] = temp[x[i]] - 1;
      }

      for (i = 0; i < x.Length; i++)
      {
         x[i] = output[i];
      }
   }

   catch (System.Exception e)
   {
      Console.WriteLine(e.ToString());
      Console.ReadLine();
   }
}
```

5.5 Radix Sort

Radix sort can be used to sort items that are identified by unique keys. Every key is a string or number, and radix sort sorts these keys in some particular lexicographic-like order.

Radix sort sorts integers by processing individual digits. Since most computers internally represent their data as binary numbers, this arrangement works out rather well. Radix sorts are classified as either least significant digit (LSD) radix sorts *or* most significant digit (MSD) radix sorts. LSD radix sorts process the integer representations starting from the least significant digit and move towards the most significant digit. MSD radix sorts work the other way around.

A least significant digit, LSD, radix sort is a fast stable sorting algorithm which can be used to sort keys in lexicographic order. Keys may be a string of characters, or numerical digits in a given radix. The processing of the keys begins at the least significant digit (*i.e.* the rightmost digit), and proceeds to the most significant digit (*i.e.*, the leftmost digit). The sequence in which digits are processed by a least significant digit LSD radix sort is the opposite of the sequence in which digits are processed by a most significant digit MSD radix sort. A radix sort algorithm works as follows:

- Take the least significant digit (or group of bits) of each key.
- Sort the list of elements based on that digit, but keep the order of elements with the same digit.
- Repeat the sort with each more significant digit.

The sort in step 2 is usually done using bucket sort or counting sort, which are efficient in this case since there are usually only a small number of digits. A C# implementation of the radix sort algorithm is presented below.

```
//RadixSort takes an array and the number of bits used as
//the key in each iteration.
public static void RadixSort(ref int[] x, int bits)
{
  //Use an array of the same size as the original array
  //to store the result of each iteration.
  int[] b = new int[x.Length];
  int[] b_orig = b;

  //Mask is the bitmask used to extract the sort key.
  //We start with the bits least significant bits and
  //left-shift it the same amount at each iteration.
  //When all the bits are shifted out of the word, we are done
  int rshift = 0;
  for (int mask = ~(-1 << bits); mask != 0; mask <<= bits, rshift +=
      bits)
  {
    //An array is needed to store the count for each key value.
    int[] cntarray = new int[1 << bits];

    //Count each key value
    for (int p = 0; p < x.Length; ++p)
    {
      int key = (x[p] & mask) >> rshift;
      ++cntarray[key];
    }
    //Sum up how many elements there are with lower
    //key values, for each key.
    for (int i = 1; i < cntarray.Length; ++i)
            cntarray[i] += cntarray[i - 1];

    //The values in cntarray are used as indexes
    //for storing the values in b. b will then be
    //completely sorted on this iteration's key.
    //Elements with the same key value are stored
    //in their original internal order.
    for (int p = x.Length - 1; p >= 0; --p)
    {
       int key = (x[p] & mask) >> rshift;
       --cntarray[key];
       b[cntarray[key]] = x[p];
    }

    //Swap the a and b references, so that the
    //next iteration works on the current b,
    //which is now partially sorted.
    int[] temp = b; b = x; x = temp;
  }
}
```

5.6 Search Algorithms

Search algorithms comprise a very important set of tools in computer programming. Like sorting algorithms, there are many types of search algorithms from which to choose from. For example, tree search algorithms are the heart of searching techniques for structured data. These algorithms search trees of nodes, whether the tree is explicit or implicit. The basic principle is that a node is taken from a data structure, and then its successors are examined and added to the data structure accordingly. Then by manipulating the data structure, the tree is explored in different orders. For instance, level by level (breadth-first search) or reaching a leaf node first and backtracking (depth-first search). Other examples of tree-searches include iterative-deepening search, depth-limited search, bidirectional search, and uniform-cost search. The efficiency of a tree search, compared to other search methods, is highly dependent upon the number and structure of nodes in relation to the number of items on that node. In addition, many of the problems in graph theory can be solved using graph traversal algorithms, such as Dijkstra's algorithm, Kruskal's algorithm, the nearest neighbor algorithm, and Prim's algorithm. These can be seen as extensions of the tree-search algorithms.

List search algorithms, however, are perhaps the most basic kind of search algorithm. The goal is to find one element of a set by some key that perhaps contains other information related to the key. The simplest such algorithm is linear search, which simply examines each element of the list in order. It has a running time in the order of $O(n)$, where n is the number of items in the list and can be used directly on any unprocessed list. A more sophisticated list search algorithm is binary search which runs in $O(\log n)$ time. This timing is significantly better than linear search for large lists of data, but it requires that the list be sorted before searching and also be random access. With a best-case complexity of $O(\log(\log n))$, interpolation search is better than binary search for large sorted lists with fairly even distributions but has a worst-case running time of $O(n)$. The remainder of this chapter will focus on these three most important list search algorithms and how they may be implemented in C#.

5.6.1 Linear Search

Linear search is a search algorithm, also known as sequential search, that is suitable for searching a list of data for a particular value. It operates by checking every element of a list one at a time in sequence until a match is found. Linear search runs in $O(n)$. If the data are distributed randomly, the expected number of comparisons that will be necessary is:

$$\begin{cases} n, & k = 0 \\ \dfrac{n+1}{k+1}, & 1 \le k \le n \end{cases}$$

where n is the number of elements in the list and k is the number of times that the value being searched for appears in the list. The best case is that the value is equal to the first element tested, in which case only 1 comparison is needed. The worst case is that the value is not in the list, or it appears only once at the end of the list, in which case n comparisons are needed. The simplicity of the linear search means that if just a few elements are to be searched it is less trouble than more complex methods that require preparation such as sorting the list to be searched or more complex data structures, especially when entries may be subject to frequent revision. Another possibility is when certain values are much more likely to be searched for than others and it can be arranged that such values will be amongst the first considered in the list. In any event, an implementation of the linear search algorithm in C# is shown below.

```
public int LinearSearch(ref int[] x, int valueToFind)
{
    for (int i=0; i<x.Length; i++)
    {
        if (valueToFind == x[i])
        {
            return i;
        }
    }
    return -1;
}
```

5.6.2 Binary Search

The Binary search algorithm is a technique for locating a particular value in a *sorted list*. The method makes progressively better guesses, and closes in on the location of the sought value by selecting the middle element in the span which, because the list is in sorted order, is the median value, comparing its value to the target value, and determining if it is greater than, less than, or equal to the target value. A guessed index whose value turns out to be too high becomes the new upper bound of the span, and if its value is too low that index becomes the new lower bound. Only the sign of the difference is inspected. There is no attempt at an interpolation search based on the size of the difference. Pursuing this strategy iteratively, the method reduces the search span by a factor of two each time, and soon finds the target value or else determines that it is not in the list at all. A binary search is an example of a dichotomic divide and conquer search algorithm.

```
public static int BinSearch(ref int[] x, int searchValue)
{
    // Returns index of searchValue in sorted input data
    // array x, or -1 if searchValue is not found
    int left = 0;
    int right = x.Length;
    return binarySearch(ref x, searchValue, left, right);
}

private static int binarySearch(ref int[] x,int searchValue,int left,
    int right)
{
    if (right < left) return -1;

    int mid = (left + right) >> 1;
    if (searchValue > x[mid])
    {
      return binarySearch(ref x, searchValue, mid + 1, right);
    }
    else if (searchValue < x[mid])
    {
      return binarySearch(ref x, searchValue, left, mid - 1);
    }
    else return mid;
}
```

5.6.3 Interpolation Search

Interpolation search is an algorithm for searching a *sorted* array by estimating the next position to check based on a linear interpolation of the search key and the values at the ends of the search interval. In each search step it calculates where in the remaining search space the sought item might be based on the key values at the bounds of the search space and the value of the sought key, usually via a linear interpolation. The key value actually found at this estimated position is then compared to the key value being sought. If it is not equal, then depending on the comparison, the remaining search space is reduced to the part before or after the estimated position.

This method works only if calculations on the size of differences between key values are sensible.

```
public static int InterpolationSearch(ref int[] x, int searchValue)
{
  // Returns index of searchValue in sorted input data
  // array x, or -1 if searchValue is not found
  int low = 0;
  int mid;
  int high = x.Length - 1;

  while (x[low] < searchValue && x[high] >= searchValue)
  {
   mid=low+((searchValue-x[low])*(high-low))/(x[high]-x[low]);
   if (x[mid] < searchValue)
      low = mid + 1;
   else if (x[mid] > searchValue)
      high = mid - 1;
   else
      return mid;
  }
  if (x[low] == searchValue)
    return low;
  else
    return -1; // Not found
}
```

5.6.4 Searching for the Maximum and Minimum Values

Computer programs are often asked to search an array, or through some other data structure, for the maximum or minimum values. The following code illustrates how to search through an array for maximum or minimum values. Of course, you can always use the built-in methods provided by the .NET Framework Math Library: `Math.Max()` and `Math.Min()`. Nevertheless, a simple implementation of the algorithms for finding the maximum and minimum values of an integer array is given below. It can be easily adapted to accommodate other data types.

```
public static int FindMax(int[] x)
{
    int max = x[0];
    for (int i = 1; i < x.Length; i++)
    { if (x[i] > max) max = x[i]; }
    return max;
}

public static int FindMin(int[] x)
{
    int min = x[0];
    for (int i = 1; i < x.Length; i++)
    { if (x[i] < min) min = x[i]; }
    return min;
}
```

5.6.5 Searching for the N-th Largest or M-th Smallest Value

Finding the *n*-th largest or the *m*-th smallest value in an array is a bit more challenging, especially if the input array is unsorted and we desire to preserve its original order. Press et al. [22], for example, provide a rather long and fancy routine in C++ that reportedly does all these things without disturbing the order of the original input array. My approach, which is listed below, may not be the fastest or most optimal but its logic flow is perhaps much easier to follow. First, we copy the original input array into a temporary array. This way we can preserve the original order of the input array while working with the temporary array. Then it is simply a matter of sorting the data in the temporary array and returning the *n*-th largest or *m*-th smallest element. It's that simple. As an alternate way to solve this problem, I also present an entirely different method that resembles that of an incomplete selection sort. The code below illustrates how both of these methods may be implemented in C#.

```
public static int NthLargest1(int[] array, int n)
{
    //Copy input data array into a temporary array
    //so that original array is unchanged
    int[] tempArray = new int[array.Length];
    array.CopyTo(tempArray, 0);
    //Sort the temporary array
    QuickSort(ref tempArray);
    //Return the n-th largest value in the sorted array
    return tempArray[tempArray.Length - n];
}

public static int NthLargest2(int[] array, int k)
{
    int maxIndex;
    int maxValue;
    //Copy input data array into a temporary array
    //so that original array is unchanged
    int[] tempArray = new int[array.Length];
    array.CopyTo(tempArray, 0);

    for (int i = 0; i < k; i++)
    {
        maxIndex = i;
        maxValue = tempArray[i];
        for (int j = i + 1; j < tempArray.Length; j++)
        {
            // if we've located a higher value
            if (tempArray[j] > maxValue)
            {   // capture it
                maxIndex = j;
                maxValue = tempArray[j];
            }
        }
        Swap(ref tempArray[i], ref tempArray[maxIndex]);
    }
    return tempArray[k - 1];
}
```

```
public static int MthSmallest1(int[] array, int m)
{
    //Copy input data array into a temporary array
    //so that original array is unchanged
    int[] tempArray = new int[array.Length];
    array.CopyTo(tempArray, 0);
    //Sort the temporary array
    QuickSort(ref tempArray);
    //Return the m-th smallest value in the sorted array
    return tempArray[m - 1];
}

public static int MthSmallest2(int[] array, int m)
{
    int minIndex;
    int minValue;
    //Copy input data array into a temporary array
    //so that original array is unchanged
    int[] tempArray = new int[array.Length];
    array.CopyTo(tempArray, 0);

    for (int i = 0; i < m; i++)
    {
        minIndex = i;
        minValue = tempArray[i];
        for (int j = i + 1; j < array.Length; j++)
        {
            if (tempArray[j] < minValue)
            {   // capture it
                minIndex = j;
                minValue = tempArray[j];
            }
        }
        Swap(ref tempArray[i], ref tempArray[minIndex]);
    }
    return tempArray[m - 1];
}
```

5.6.6 Some Useful Utilities

We end this chapter with a brief collection of useful utilities for quickly determining whether an array is sorted or not and if it is sorted, whether it is sorted in an ascending or a descending order. Routines are given for both integer and string arrays and can be easily adopted to other data types.

```
// Determines if int array is sorted from 0 -> Max
public static bool IsSorted(int[] arr)
{
    for (int i = 1; i < arr.Length; i++)
    {
        if (arr[i - 1] > arr[i]) return false;
    }
    return true;
}
```

```
// Determines if string array is sorted from A -> Z
public static bool IsSorted(string[] arr)
{
    for (int i = 1; i < arr.Length; i++)
    {
        if (arr[i - 1].CompareTo(arr[i]) > 0) return false;
    }
    return true;
}

// Determines if int array is sorted from Max -> 0
public static bool IsSortedDescending(int[] arr)
{
    for (int i = arr.Length - 2; i >= 0; i--)
    {
        if (arr[i] < arr[i + 1]) return false;
    }
    return true;
}

// Determines if string array is sorted from Z -> A
public static bool IsSortedDescending(string[] arr)
{
    for (int i = arr.Length - 2; i >= 0; i--)
    {
        if (arr[i].CompareTo(arr[i + 1]) < 0) return false;
    }
    return true;
}
```

As mentioned in Chapter 1, the .NET Framework provides a generic collection class called `List<>` with sizes that can be adjusted dynamically and other desirable features that are not available to its conventional array data structure. However, through the use of some built-in methods, it is possible to convert between one data structure and the other. The examples shown below illustrate how.

```
Console.WriteLine("\n\nTesting Conversion from List to Array\n");
List<string> myList = new List<string>();
myList.Add("duck");
myList.Add("bunny");
myList.Add("goose");
myList.Add("chipmunk");
myList.Add("dove");
string[] myString = myList.ToArray();
foreach (string s in myString)
    Console.WriteLine(s);

Console.WriteLine("\n\nTesting Conversion from Array to List\n");
string[] str =
new string[] {"duck","bunny","goose","chipmunk","dove"};
List<string> myOtherList = new List<string>(str);
myOtherList = str.ToList();
foreach (string s in myOtherList)
    Console.WriteLine(s);
```

6

Bits and Bytes

6.1 Introduction

Programming computers to interact with the physical world is perhaps one of the most challenging, exciting and important tasks which programmers are sometimes asked to do. Typical computer interfacing projects in either the engineering or scientific laboratory often involve the development of software applications capable of controlling physical equipment by passing data to and from hardware devices. However, the successful development and deployment of such ambitious and exciting computer applications often require a deeper understanding and knowledge of both hardware features and more advanced software concepts. Although a full discussion of the hardware aspects of computer interfacing is beyond the scope of this book and actually merits an entirely new book exclusively dedicated to that topic, there are some important related software concepts that deserve some attention. Consequently, this chapter will focus on the topic of bit manipulation, bitwise operations and other related software issues that make such innovative computer applications possible. As with all the other chapters in this book, my focus will be largely on developing practical examples of how one might apply some of these advanced but basic concepts to the development of effective software applications.

6.2 Numeric Systems

A numeric system is a conceptual tool for thinking about numbers and expressing them in a consistent manner by using arbitrary symbols. The earliest and perhaps most primitive of these is called the unary numeric system where every natural number is represented by a corresponding symbol, usually a slash. Today, the most commonly used numeric system consists of Hindu-Arabic symbols which were originally developed around the 5th century A.D. Because humans have ten fingers, this numeric system uses ten basic symbols called *digits*, 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 which when grouped together form the well known decimal number system we have today. Therefore, working in a base 10 numeric system and using what has come to be known as the place-value notation, one can use the position of a digit to signify the power of ten by which the digit is to be multiplied and added to construct a particular number. For example, $507 = 5\text{x}10^2 + 0\text{x}10^1 + 7\text{x}10^0$. In this context, the numeral 101 can be interpreted as the decimal numeral for one hundred and one, the binary numeral for 5, or some other numerical value depending on the numeric base being used.

The electronic components of modern computers function only in two possible states of either high or low voltages. Denoting high voltages by 1 and low voltages by 0 leads to the entire set of binary digits and, as a result, the binary number system was adopted long ago to handle all aspects of computer hardware design and software development. Collectively, these *b*inary dig*its* are called *bits* and can be grouped together into groups of 4 (*nibble*), 8 (*byte*), 16 (*word*), 32 (*dword*) or even 64 (*qword*) bits. Generally speaking, a *word* is a term for the natural unit of data used by a particular computer design. A word is simply a fixed-sized group of bits that are handled together by the machine. The number of bits in a word (the word size or word length) is an important characteristic of a computer architecture. Modern computers usually have a word size of 16, 32, or 64 bits. In any event, it would be useful to develop some routines in C# to convert numerical values back and forth between all these different numerical bases.

In a positional-based numeric system, where b is a positive natural number also known as the *radix* or *base*, there are a total of b fundamental symbols (or digits) corresponding to the first b natural numbers including zero. Therefore, if b is the base, one can write any number in the numeral system of base b by expressing it in the following generalized format: $a_nb^n + a_{n-1}b^{n-1} + a_{n-2}b^{n-2} + \cdots + a_0b^0$ which then yields the enumerated digits $a_na_{n-1}a_{n-2}\cdots a_0$ in a descending order. The digits are natural numbers between 0 and $b-1$, inclusive. Unless implied by context and in order to avoid ambiguity when handling different numeric bases, numbers without any subscripts are traditionally considered to be decimals. Numbers meant to represent numeric values in other bases are followed by a subscript added to the lower right side of the number. Thus in the example given above, it would be considered better practice to specify the base being used right alongside the number as shown: $101_2 = 5_{10}$. By introducing a dot called the radix point to split the digits into two groups, one can also write fractions in the positional system. For example, the base-2 numeral $10.11_2 = 1\text{x}2^1 + 0\text{x}2^0 + 1\text{x}2^{-1} + 1\text{x}2^{-2} = 2.75_{10}$. Therefore if b is a positive integer greater than 1, any number x in the base b system with n digits to the left of the radix point and m digits to the right of the radix point can be uniquely expressed in the following format [38]:

$$x = (a_na_{n-1}\cdots a_1a_0.a_{-1}a_{-2}a_{-3}\cdots a_{-m})_b = \sum_{k=-m}^{n} a_kb^k$$

where a_i is a non-negative integer, called a digit, that is less than b.

There is a very simple algorithm that can be used to construct the base b expansion of a number x and thus convert numerical values between different bases. In this algorithm, the integer and fractional parts of a number are each converted separately and then the results are added up. Let us consider the integer part of the number to be converted first. The basic idea in a nutshell is to start out by dividing the number to be converted by the base thus obtaining an initial quotient along with a remainder. Then repeatedly divide each subsequent quotient by the base until one obtains a quotient that is equal to zero. The remainder values are then set in the corresponding digit position.

The algorithm to calculate the fractional part is a little more challenging. First, multiply the fractional part of the number by the base in order to shift the radix point to the right by 1. If the result is greater than or equal to 1, keep only the integer part assigning it to the corresponding digit and discard the value left of the radix point. If the result is less than 1 assign the corresponding digit the value of 0. Repeat this process until the input fractional number value reaches 0. Unfortunately, there are situations where this algorithm may not terminate. For example, if you try to convert 0.3 to base 2, you will end up in an infinite loop. Fortunately, since 0.3 is rational, there will be a repeated set of digits and so with careful coding, you can stop the loop once you see a set of digits repeat. For irrational numbers where the loop is infinite, you can choose to terminate the loop after a certain number of values have been calculated.

Unfortunately, the .NET Framework does not provide specific routines for directly converting numerical values between different bases and so one has to program this function from scratch. The following code snippet contains one possible solution for converting unsigned positive integer values between different bases ranging anywhere from 2 to 36. The maximum base value of 36 allows for the use of all possible unique digits that can be formed using the standard numeric digits from 0 to 9 followed by the 26 letters of the alphabet from A to Z. The base conversion program provided below actually consists of a two step process in order to minimize small numerical errors from slowly creeping into the conversion calculations. First, the input numerical value is converted from its original base to its equivalent value in base 10. Then that value in base 10 is converted to the equivalent value in the desired target base. This code can be immediately improved in at least two ways. First, by adding functionality to accept both unsigned *and* signed integer values. Second, by adding functionality to enable it to accept fractional non-integer data values. These two suggested improvements will be left as an exercise for the reader.

```
public static string ConvertToBase(string number, int start_base, int
     end_base)
{
   long base10 = ConvertToBase10(number, start_base);
   string result = ConvertFromBase10(base10, end_base);
   return result;
}

public static long ConvertToBase10(string number, int start_base)
{
   if (start_base == 10) return Convert.ToInt64(number);
   if (start_base < 2 || start_base > 36) return 0;

   char[] CharDataArray = number.ToCharArray();
   int i = CharDataArray.Length - 1;
   int j = start_base;
   int k = 0;
   long result = 0;

   foreach (char c in CharDataArray)
   {
      if (char.IsNumber(c))
      k = int.Parse(c.ToString());
      else
         k = Convert.ToInt32(c) - 55;
         result += k * (Convert.ToInt64(Math.Pow(j, i)));
         i--;
   }
   return result;
}

public static string ConvertFromBase10(long number, int end_base)
{
   if (end_base == 10) return number.ToString();
   if (end_base < 2 || end_base > 36) return "";

   long quotient = number;
   long remainder;
   string result = "";

   while (quotient >= end_base)
   {
      remainder = quotient % end_base;
      quotient = quotient / end_base;

      if (remainder < 10)
         result = remainder.ToString() + result;
      else
         result = Convert.ToChar(remainder + 55).ToString() + result;
   }

   if (quotient < 10)
      result = quotient.ToString() + result;
   else
      result = Convert.ToChar(quotient + 55).ToString() + result;
   return result;
}
```

6.3 Bit Manipulation and Bitwise Operators

Computers represent data internally as sequences of bits and computer hardware process data as bits or groups of bits. Each bit can assume either the value 1 or the value 0 and a sequence of 8 bits is said to form a *byte*. Note that a byte is also the standard storage unit for a variable of data type byte. Other data types require larger number of bytes for storage.

In referencing specific bits within a binary number, it is common to assign a bit index number to each bit. The bit index number starts at 0 and is incremented by one for each subsequent bit position up to one less than the total number of bits. Therefore, an n-bit number would be written as $a_{n-1}a_{n-2}\cdots a_1a_0$ where the *least significant bit (lsb)* is the right-most bit position, a_0, and the *most significant bit (msb)* is the left-most bit position, a_{n-1}, in a binary integer. Similarly, the corresponding acronyms *LSB* and *MSB*, all in uppercase letters, stand for *Least Significant Byte* and *Most Significant Byte*, repectively.

Data can be stored in memory in one of two ways. In the *Big Endian* scheme, the most significant byte is stored in the smallest or lowest memory address whereas in the *Little Endian* configuration, the least significant byte is stored in the smallest or lowest memory address. As a memory aid to remember the difference between these two configuration directions, recall that if the least significant byte is stored first you have *Little Endian* and if the most significant byte is stored first, you have *Big Endian.* This property is particularly important when writing data to or reading data from files and when sending or receiving data over a network. Computers whose endianess differ from each other will see data in reverse order which will not make any sense. The `BitConverter` class, which will be covered later in this chapter, has an `IsLittleEndian` field that returns boolean value indicating whether the particular computer uses the *Little Endian* or the *Big Endian* configuration.

```
bool le = BitConverter.IsLittleEndian;
```

Although impressive, computers are nonetheless still primitive machines that can only differentiate between two states, 0 or 1, and, in addition, numbers can be stored only up to a limited number of digits. Accordingly, it was left to hardware designers to figure out a clever way for computers to handle negative numbers effectively. Recalling that in C# integer data types can be either signed or unsigned, the *msb* can also be used to indicate the sign of a signed binary number in one or two's complement notation where a msb of 1 means a negative number and a msb of 0 means a positive number. The method of *complements* is just a mathematical technique that was devised in order to subtract one number from another using only the addition of positive numbers.

The concept behind numerical complement scheme can be briefly explained as follows. The radix *complement* of an n digit number y in radix b is, by definition, $b^n - y$. Adding this to some other number x where $x \leq y$ results in the value $x + b^n - y$ or $x - y + b^n$ which is always greater than b^n. Thus, by subtracting b^n from this total, we obtain $x - y + b^n - b^n$ or just $x - y$, which is the desired result. In the decimal numbering system, the radix complement is called the ten's complement and the diminished radix complement is called the nines' complement. In the binary numbering system, the radix complement is called the two's complement and the diminished radix complement is called the ones' complement.

Since $b^n - 1 = b^n - 1^n = (b-1)(b^{n-1} + b^{n-2} + \cdots + b + 1) = (b-1)b^{n-1} + ... + (b-1)$, we observe that $(b^n - 1)$ is the just the digit $b-1$ repeated n times. Therefore, the radix complement $b^n - y$ is most easily obtained by adding 1 to the diminished radix complement, $(b^n - 1) - y$. That is, $b^n - y = (b^n - 1) - y + 1$. The diminished radix complement of a number y is found by subtracting each digit in y from $b-1$. Finally, adding 1 to obtain the radix complement can be done separately, but is most often combined with the addition of x and the complement of y. Any overflows in the total number of digits for a given radix, are simply discarded. The following two examples should illustrate the ideas behind this algorithm that was just described.

Consider the following decimal subtraction example: $873-218$ where here $x = 873$ and $y = 218$. To subtract a decimal number y from another number x using the method of complements, the ten's complement of y (nines' complement plus 1) is added to x. Typically, the nines' complement of y is first obtained by determining the complement of each digit. The complement of a decimal digit in the nines' complement system is the number that must be added to it to produce 9. The complement of 3 is 6, the complement of 7 is 2, and so on. Therefore,

```
  873  (x)
+ 781  (complement of y)
+   1  (to get the ten's complement of y)
=====
 1655
```

The first 1 digit is then dropped, giving 655, the correct answer. If the subtrahend has fewer digits than the minuend, leading zeros must be added which will become leading nines when the nines' complement is taken. For example, $48032-391 = 48032+99608+1 = 147641$. Dropping the left-most 1 gives the correct answer: 47641. The method of complements is especially useful in binary (radix 2) since the ones' complement is very easily obtained by just inverting each bit. Adding 1 to get the two's complement can be done by simulating a carry into the least significant bit. For example:

```
  01100100  (x, equals decimal 100)
- 00010110  (y, equals decimal 22)
```

becomes the sum:

```
  01100100  (x)
+ 11101001  (ones' complement of y)
+        1  (to get the two's complement)
==========
 101001110
```

Dropping the initial 1 gives the answer: 01001110.

The method of complements normally assumes that the operands are positive and that $y \leq x$. But what happens if $x < y$? In that case, there will not be a 1 digit to discard after the addition since $x-y+b^n$ will be less than b^n. For example, $185-329 = 185+670+1 = 856$ which is obviously the wrong answer since we were expecting the result to be -144. However, 856 just happens to be the 10's complement of 144. Therefore, if $x < y$ then in order to obtain the final correct result one needs to add an additional step and complement the result if there is no carry out of the most significant digit.

In C#, the bitwise operators can manipulate individual bits stored in variables consisting of the following data types: `sbyte`, `byte`, `char`, `short`, `ushort`, `int`, `uint`,

TABLE 6.1
Results of combining individual bits with the bitwise AND, OR and XOR operators.

Bit 1	Bit 2	Bit 1 & Bit 2	Bit 1 | Bit 2	Bit 1 ˆ Bit 2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

`long` and `ulong`. The operators bitwise AND (&), bitwise inclusive OR (|), and bitwise exclusive OR (ˆ) or XOR, operate similarly to their logical counterparts except that their bitwise versions operate on the level of bits. The actual functionality of each of these operators on individual bits is summarized in Table 6.1. In addition, the *left-shift* << operator shifts the bits of its left operand to the left by the number of bits specified in its right operand. The rightmost bits are replaced with 0s and any 1s shifted to the left are lost. The *right-shift* >> operator shifts the bits of its left operand to the right by the number of bits specified in its right operand. 0s replace the vacated bits on the left side if the number is positive and 1s replace the vacated bits if the number is negative. Any 1s shifted off to the right are lost.

As for negative numbers, for bits shifted to the left, regardless of sign, and for bits shifted to the right where the value is either unsigned or signed but positive, vacated bits are filled with 0s. However, for signed numbers that are negative, most compilers will attempt to preserve the negative sign of the value and so in a right shift, the vacated bits are replaced with 1s instead of with 0s. However, if you have a negative signed number and you want to make sure the left-most bits are replaced with 0s instead of 1s, you can AND with a mask containing 0s in the vacated bits and 1s in the other bits following the shift right operation as shown:

```
int negvalue = -1;
negvalue = negvalue >> numbitsToShift & 0xF
```

Note that this way of coding immediately places 0s on the left number of bits as specified by the variable `numbitsToShift` indicating how many bits to the right the negative value has been right-shifted. Although you can use more than one bit shift operation in a single expression, readers are cautioned that shift operations can produce undefined results if the bits to be shifted in either direction exceed the total number of allowed bits for the particular data type being used.

Note that since each bit represents a successively higher power of 2, shifting bits to the right has the effect of dividing a value by 2 for each bit shift. Conversely, shifting to the left has the effect of multiplying a value by 2 for each bit shift. These bit shifting features can sometimes provide us with not just an alternate or fancy way for computers to carry out some calculations but they also help increase the computation speed since fewer steps are then actually needed. For example, consider the problem of converting decimal numbers to their equivalent binary, octal and hexadecimal counterparts. In order to convert a number of base 10 to binary, we need to divide by 2 or simply shift the bitwise representation of that number by 1 bit. Likewise, in order to convert to octal, we need to shift the bitwise representation of the target number by 3 bits. As for conversion to hexadecimals, we need to shift by 4 bits and so on. The method `DecToBinOctOrHex`, shown below, illustrates this idea. This method is also overloaded in order to accommodate both both positive and negative values of int32 and double data types. In addition, in the case of numerical values of double data types, the base conversion routines now allow for the inclusion of fractional parts of a numerical value.

```
public static string DecToBinOctOrHex(int number, int baseval)
{
   int n, bitcounter, bitstoshift, startbit;
   string digit = "0123456789ABCDEF";

   switch (baseval)
   {
      case 2:
         startbit = 31;
         bitstoshift = 1;
         break;
      case 8:
         startbit = 30;
         bitstoshift = 3;
         break;
      case 16:
         startbit = 28;
         bitstoshift = 4;
         break;
      default:
         startbit = 0;
         bitstoshift = 0;
         break;
   }

   StringBuilder output = new StringBuilder();

   for (bitcounter = startbit, n = number; bitcounter >= 0;
        bitcounter -= bitstoshift)
   {
      output.Append(digit[(n >> bitcounter) & (baseval - 1)]);
   }
   return output.ToString();
}

public static string DecToBinOctOrHex(double number, int baseval)
{
   int n, bitcounter, bitstoshift, startbit;
   string digit = "0123456789ABCDEF";

   switch (baseval)
   {
      case 2:
         startbit = 31;
         bitstoshift = 1;
         break;
      case 8:
         startbit = 30;
         bitstoshift = 3;
         break;
      case 16:
         startbit = 28;
         bitstoshift = 4;
         break;
      default:
         startbit = 0;
         bitstoshift = 0;
         break;
   }

   StringBuilder output = new StringBuilder();

   for (bitcounter = startbit, n = (int)Math.Truncate(number);
        bitcounter >= 0; bitcounter -= bitstoshift)
   {
      output.Append(digit[(n >> bitcounter) & (baseval - 1)]);
   }
   output.Append(".");
   for (int i = 0; i < 12; i++)
   {
      number = (number - Math.Floor(number)) * baseval;
      output.Append(digit[(int)Math.Truncate(number)]);
   }
   return output.ToString();
}


//SIMPLE DRIVER PROGRAM TO TEST CODE
//Convert integer to binary
int intdata = 28;
string msg = "Integer  {0} in decimal = {1} in binary.\n";
Console.WriteLine(msg, intdata, DecToBinOctOrHex(intdata,2));
intdata = -28;
Console.WriteLine(msg, intdata, DecToBinOctOrHex(intdata,2));

//Convert double to binary
double dbldata = 3.141592653589793;
msg = "Double  {0} in decimal = \n{1} in binary.\n";
Console.WriteLine(msg, dbldata, DecToBinOctOrHex(dbldata,2));
dbldata = -3.141592653589793;
Console.WriteLine(msg, dbldata, DecToBinOctOrHex(dbldata,2));

//Convert integer to octal
intdata = 28;
msg = "Integer  {0} in decimal = {1} in octal.\n";
Console.WriteLine(msg, intdata, DecToBinOctOrHex(intdata,8));
intdata = -28;
Console.WriteLine(msg, intdata, DecToBinOctOrHex(intdata,8));

//Convert double to octal
dbldata = 3.141592653589793;
msg = "Double  {0} in decimal = {1} in octal.\n";
Console.WriteLine(msg, dbldata, DecToBinOctOrHex(dbldata,8));
```

```
dbldata = -3.141592653589793;
Console.WriteLine(msg, dbldata, DecToBinOctOrHex(dbldata,8));

//Convert integer to hexadecimal
intdata = 28;
msg = "Integer  {0} in decimal = {1} in hexadecmal.\n";
Console.WriteLine(msg, intdata, DecToBinOctOrHex(intdata,16));
intdata = -28;
Console.WriteLine(msg, intdata, DecToBinOctOrHex(intdata,16));

//Convert double to hexadecimal
dbldata = 3.141592653589793;
msg = "Double  {0} in decimal = {1} in hexadecmal.\n";
Console.WriteLine(msg, dbldata, DecToBinOctOrHex(dbldata,16));
dbldata = -3.141592653589793;
Console.WriteLine(msg, dbldata, DecToBinOctOrHex(dbldata,16));

OUTPUT:

Integer  28 in decimal =
00000000000000000000000000011100 in binary.

Integer -28 in decimal =
11111111111111111111111111100100 in binary.

Double  3.14159265358979 in decimal =
00000000000000000000000000000011.001001000011 in binary.

Double -3.14159265358979 in decimal =
11111111111111111111111111111101.110110111100 in binary.

Integer  28 in decimal = 00000000034 in binary.

Integer -28 in decimal = 77777777744 in binary.

Double  3.14159265358979 in decimal =
00000000003.110375524210 in octal.

Double -3.14159265358979 in decimal =
77777777775.667402253567 in octal.

Integer  28 in decimal = 0000001C in hexadecmal.

Integer -28 in decimal = FFFFFFE4 in hexadecmal.

Double  3.14159265358979 in decimal =
00000003.243F6A8885A3 in hexadecmal.

Double -3.14159265358979 in decimal =
FFFFFFFD.DBC095777A5D in hexadecmal.
```

The bitwise complement operator $\sim$ sets all 0 bits in its operand to 1 and all 1 bits in its operand to 0. That is, $\sim 0 = 1$ and $\sim 1 = 0$. This process, as explained earlier in this section, is also known as taking the one's complement of the value. Larger sets of bits are likewise individually changed to their complementary counterpart as shown in this example: $\sim 01011 = 10100$.

With the exception of the complement operator, each bitwise operator also has a corresponding assignment operator. Thus &= is the bitwise AND assignment operator, |= is the bitwise inclusive OR assignment operator, ^= is the bitwise exclusive OR assignment operator , <<= is the left-shift assignment operator, and >>= is the right-shift assignment operator.

Masks are simply values that may be used in conjunction with bitwise operators to manipulate specific bits. To set a bit means to make the value of the bit 1. To clear a bit means to make the value of the bit 0. Masks can be used to turn specific bits on or off, toggle bits or determine the value of one or more specific bits. Masks allow one to essentially ignore certain bits, whose values we may not know or even care about, while at the same time altering or ascertaining the values of target bits. These features are particularly useful in developing computer interfacing applications where it may be desired to set, reset, toggle or test both incoming or outgoing blocks of individual bits. The following paragraphs contain a series of brief descriptions along with ready-to-use C# code recipes for fiddling around with bit values.

TURNING DESIGNATED BITS ON: `num |= MASK` Recall that X OR $1 = 1$ and X OR $0 = X$. Therefore, to turn one or more designated bits in a value on and leave the rest unchanged, we OR the value with a mask that has a 1 in the target bit locations and 0s everywhere else. This process is also referred to as *setting bits*. To have the 5th bit in a byte always turned on, for example, a mask containing the value 00001000 is created and then ORed with the input data bits as shown:

```
    10011101   10010101
 OR 00001000   00001000 (mask byte)
  = 10011101   10011101
```

TURNING DESIGNATED BITS OFF: `num &= MASK` Recall that X AND $0 = 0$ and X AND $1 = 1$ only if $X = 1$. Therefore, to clear one or more designated bits in a value and leave the rest unchanged, we AND the value with either (1) a mask that has 0s in the bit or bits we want to turn off and 1s in all the other bits or, (2) the complement of the mask that has 1s in the bit we want to turn off and 0s in all other bits. This process is also referred to as *clearing bits*. To have the 5th bit in a byte always turned off, for example, a mask containing the value 00001000 is created and then ANDed with the input data bits as shown:

```
    10011101   10010101
AND 11110111   11110111 (mask byte)
  = 10010101   10010101
```

TURNING OFF ALL BITS IN A NUMBER: `num ^= num` Recall that the exclusive OR operator makes a bit-by-bit comparison between two integer values and, for each bit position, yields a 1 if one of the bits is 1 and the other is 0. The result is 0 if both bits are 0s or both are 1s. Therefore you can turn all bits in a number off by XORing the number with itself. Of course, you could also do the same thing by assigning 0 to the variable but doing it this way is both instructive and far more fun. Therefore, in the spirit of the examples just described we can turn clear all the bits of some byte value, say `10011101` by XORing it with itself as shown:

```
    10011101
XOR 10011101
  = 00000000
```

TOGGLING BITS: `num ^= MASK` By toggling a bit, we mean turning it off if it is on and vice versa. You can use the XOR operator and a mask to toggle one or more bits. Bits corresponding to 1s in the mask are toggled and those corresponding to 0s in the mask are left unchanged. However, if you want to toggle all bits at once, then just use the complement operator. Consider the following example that illustrates this idea:

```
    10011001
XOR 11011010 (byte mask)
  = 01000011
```

DETERMINING IF ONE OR MORE BITS ARE ON: `if (num & MASK==MASK)` To determine whether one or more bits in a variable or constant are turned on, we AND with a mask that contains 1s in the bits we want to look at and 0s in the other bits. Then we check to see whether the resulting value is equal to the mask. Bits that are ANDed with 0s yield 0s, as in the mask. If the target bits in the number being checked are 1s, so that ANDing with 1 yields 1s as in the mask, the result of `if (num & MASK == MASK)` will be true if the target bits all are on and false if one or more is off. Consider the results contained in the following example:

```
    10011101            11011111
AND 11010111            11010111 (mask byte)
  = 10010101 (false)    11010111 (true)
```

ASCERTAINING THE VALUE OF SEVERAL BITS AT ONCE: `num & MASK` To ascertain the value of several specific contiguous bits, all of which are low-order bits, you AND with a mask that has 0s in the bits you are not interested in and 1s in those you are. Consider ANDing some arbitrary input byte value with the mask `0x7 = 00000111` which has bits 0-2 on and the rest off. This will zero out bits 3-7 and leave bits 0-2 in their original settings, thus isolading the value they contain as shown in the following example:

```
    10011101
AND 00000111 (mask byte)
  = 00000101
```

The following listing contains source code to illustrate and expand on the ideas and basic material for bitwise manipulation and operations just described above.

```
// Converts an int value into its 32 bit binary representation
public static string DisplayBits(int value,int nBitsToDisplay)
{
  int mask = 1 << nBitsToDisplay - 1;
  StringBuilder output = new StringBuilder();

  // Get each bit, add space every 8 bits
  // for display formatting
  for (int bitcounter = 1; bitcounter<=nBitsToDisplay;bitcounter++)
  {
    // Append 0 or 1 depending on result of masking
    output.Append((value & mask) == 0 ? "0" : "1");
    // Shift left so that mask will find bit of
    // next digit during next iteration of loop
    value <<= 1;

    if (bitcounter % 8 == 0) output.Append(" ");
  }
  return output.ToString();
}

//Returns value with the mask bits set
public static int setBits(int value, int mask)
{
   return value | mask;
}

//Returns value with the all the bits set except the mask bits
public static int setBitsExcept(int value, int mask)
{
   return value | ~mask;
}

//Returns true if any of the bits in mask is set in value
public static bool isAnyBitSet(int value, int mask)
{
   return (value & mask) != 0;
}

//Returns true if all the bits in mask are set in value
public static bool areAllBitsSet(int value, int mask)
{
   return (value & mask) == mask;
}

//Flips bit at position n
public static int BitFlip(int value, int n)
{
   return ((value) ^ (1 << (n)));
}

//Toggles bit according to mask
public static void BitToggle(int value, int mask)
{
   mask ^= value;
}
```

```
//Returns value with the nth bit set
public static int setBitByPos(int value, int bitNumber)
{
   return value | (1 << bitNumber);
}

//Returns true if value has the the nth bit set
public static bool isBitSetByPos(int value, int bitNumber)
{
   return (value & (1 << bitNumber)) != 0;
}

//Returns value with the mask bits set or cleared depending
//on the value of set.
public static int setBits(int value, int mask, int bSet)
{
  return bSet != 0? setBits(value,mask):clearBits(value,mask);
}

//Returns value with the mask bits cleared
public static int clearBits(int value, int mask)
{
   return value & ~mask;
}

//Returns true if all the bits in mask are cleared in value
public static bool areAllBitsClear(int value, int mask)
{
   return (value & mask) == 0;
}

//Returns value with all the bits cleared except the mask bits
public static int clearBitsExcept(int value, int mask)
{
   return value & mask;
}

//Returns value with the nth bit cleared
public static int clearBitByPos(int value, int bitNumber)
{
   return value & ~(1 << bitNumber);
}

//Returns true if value has the the nth bit cleared
public static bool isBitClearByPos(int value, int bitNumber)
{
   return isBitSetByPos(value, bitNumber) == false;
}

//Returns value with add bits set and the remove bits cleared
public static int setClearBits(int value,int add,int remove)
{
   return (value | add) & ~remove;
}
```

```
//Returns the one's complement of an input int value
public static int OnesComplement(int value)
{
   return (~value);
}

//Returns the two's complement of an input int value
public static int TwosComplement(intvalue)
{
   return (~value + 1);
}

// Driver Program with examples bitwise operations
int a = 0x005A; // in binary = 0000 0000 0101 1010
int b = 0x3C5A; // in binary = 0011 1100 0101 1010

Console.WriteLine("a = {0} = \t{1}",a,DisplayBits(a,16));
Console.WriteLine("b = {0} = \t{1}\n",b,DisplayBits(b,16));

Console.WriteLine("a & b = {0:x} = \t{1}", a & b,
                  DisplayBits(a & b,16));
Console.WriteLine("a | b = {0:x} = \t{1}", a | b,
                  DisplayBits(a | b,16));
Console.WriteLine("a ^ b = {0:x} = \t{1}", a ^ b,
                  DisplayBits(a ^ b,16));
Console.WriteLine("~a = {0:x} = {1}\n",~a,DisplayBits(~a,16));

Console.WriteLine("a>>1 = {0} = \t{1}", a >> 1,
                  DisplayBits(a >> 1,16));
Console.WriteLine("a>>5 = {0} = \t{1}\n", a >> 5,
                  DisplayBits(a >> 5,16));

Console.WriteLine("a<<1 = {0} = \t{1}", a << 1,
                  DisplayBits(a << 1,16));
Console.WriteLine("a<<5 = {0} = \t{1}\n", a << 5,
                  DisplayBits(a << 5,16));

Console.WriteLine("Ones complement of {0}\n = {1}\n",
        DisplayBits(a,16), DisplayBits(OnesComplement(a),16));
Console.WriteLine("Twos complement of {0}\n = {1}",
        DisplayBits(a,16), DisplayBits(TwosComplement(a),16));
Console.WriteLine("Twos Complement of {0} in decimal
        format = {1}", a, TwosComplement(a));
```

Typical engineering and scientific programming projects sometimes require the development of software applications that can physically control laboratory equipment by passing data to and from hardware devices. As a result, programmers may find themselves needing to convert a built-in data type into an array of bytes. For example, some hardware device might require an integer value, but that value must be sent one byte at a time. The reverse situation can also occur where data is received as an ordered sequence of bytes that then needs to be converted back into one of the built-in data types before it can be interpreted, stored or analyzed. Fortunately, the .NET Framework provides the `BitConverter` class which contains many useful features especially designed for converting values of some particular data type into their corresponding byte arrays where individual bits can then be easily accessed and manipulated. Conversely, `BitConverter` also provides features designed specifically for converting byte arrays back into values corresponding to their respective data type. Together, these features provide an arsenal of excellent software tools for most bit twiddling and hardware device interfacing needs.

Nevertheless, some words of caution are also needed here. Although their names may be closely related, the `BitConverter` class should not be confused with the `Convert` class which contains methods designed for converting variables of one *data type* to another. In addition, although the `BitConverter` class provides a convenient set of tools for converting most basic value types to and from byte arrays, the decimal data type is an exception in which case you have to use a `System.IO.MemoryStream` object as the following two methods illustrate.

```
// Create a byte array from a decimal numeric value.
public static byte[] DecimalToByteArray(decimal src)
{
   // Create a MemoryStream as a buffer to hold the binary data.
   using (MemoryStream stream = new MemoryStream())
   {
      // Create a BinaryWriter to write binary data to the stream.
      using (BinaryWriter writer = new BinaryWriter(stream))
      {
         // Write the decimal to the BinaryWriter/MemoryStream.
         writer.Write(src);

         // Return the byte representation of the decimal.
         return stream.ToArray();
      }
   }
}

// Create a decimal numeric value from a byte array.
public static decimal ByteArrayToDecimal(byte[] src)
{
   // Create a MemoryStream containing the byte array.
   using (MemoryStream stream = new MemoryStream(src))
   {
      // Create a BinaryReader to read the decimal from the stream.
      using (BinaryReader reader = new BinaryReader(stream))
      {
         // Read and return the decimal from the
         // BinaryReader/MemoryStream.
         return reader.ReadDecimal();
      }
   }
}
```

Together with the two special methods for converting decimal data type values to and from byte arrays that were just described above, the following code examples illustrate the use of a few `BitConverter` class methods for various different data types.

```
//Convert an int to a byte array and display
byte[] b = null;
b = BitConverter.GetBytes(3678);
Console.WriteLine(BitConverter.ToString(b));
OUTPUT: 5E-0E-00-00

//Convert a byte array to an int and display
byte[] b = null;
b = BitConverter.GetBytes(3678);
Console.WriteLine(BitConverter.ToInt32(b,0));
OUTPUT: 3678

//Convert different data types to byte array
//and convert byte array back
string formatter = "{0,10}{1,10}{2,25}";
double aDoubl = 3.1415;
float aSingl = 137.5F;
long aLong = 123456789;
int anInt = 9732;
short aShort = 25;
char aChar = 'M';
bool aBool = true;
decimal aDec = 842.696m;

//Convert given data values into corresponding byte arrays
byte[] byteArrDouble = BitConverter.GetBytes(aDoubl);
byte[] byteArrSingl = BitConverter.GetBytes(aSingl);
byte[] byteArrLong = BitConverter.GetBytes(aLong);
byte[] byteArrInt = BitConverter.GetBytes(anInt);
byte[] byteArrShort = BitConverter.GetBytes(aShort);
byte[] byteArrChar = BitConverter.GetBytes(aChar);
byte[] byteArrBool = BitConverter.GetBytes(aBool);
byte[] byteArrDecimal = DecimalToByteArray(aDec);

//Convert byte arrays values back into data values
double valueDbl = BitConverter.ToDouble(byteArrDouble,0);
Single valueSingl = BitConverter.ToSingle(byteArrSingl,0);
long valueLong = BitConverter.ToInt64(byteArrLong,0);
int valueInt = BitConverter.ToInt32(byteArrInt,0);
short valueShort = BitConverter.ToInt16(byteArrShort,0);
char valueChar = BitConverter.ToChar(byteArrChar,0);
bool valueBool = BitConverter.ToBoolean(byteArrBool,0);
decimal valueDecimal = ByteArrayToDecimal(byteArrDecimal);

Console.WriteLine(
"Example of using methods in the BitConverter\n" +
"class to convert values of specified data\n" +
"types into corresponding byte arrays\n");
Console.WriteLine(formatter,"Data Type","Value","Byte Array");
Console.WriteLine(formatter,"--------","--------","--------");
Console.WriteLine(formatter,"Double", aDoubl,
BitConverter.ToString(byteArrDouble));
Console.WriteLine(formatter, "Single", aSingl,
BitConverter.ToString(byteArrSingl));
Console.WriteLine(formatter, "Long", aLong,
BitConverter.ToString(byteArrLong));
```

```
Console.WriteLine(formatter, "Int", anInt,
BitConverter.ToString(byteArrInt));
Console.WriteLine(formatter, "Short", aShort,
BitConverter.ToString(byteArrShort));
Console.WriteLine(formatter, "Char", aChar,
BitConverter.ToString(byteArrChar));
Console.WriteLine(formatter, "bool", aBool,
BitConverter.ToString(byteArrBool));
Console.WriteLine(formatter, "Decimal", aDec,
BitConverter.ToString(byteArrDecimal));

Console.WriteLine(
"\n\n Example of using methods in the BitConverter class\n" +
"to convert byte arrays back to values of a\n" +
"corresponding specified data type\n");
formatter = "{0,25}{1,15}{2,15}";
Console.WriteLine(formatter,"Byte Array","Value","Data Type");
Console.WriteLine(formatter,"-------","---------","--------");
Console.WriteLine(formatter,
BitConverter.ToString(byteArrDouble), valueDbl, "Double");
Console.WriteLine(formatter,
BitConverter.ToString(byteArrSingl),valueSingl,"Single");
Console.WriteLine(formatter,
BitConverter.ToString(byteArrLong),valueLong,"Long");
Console.WriteLine(formatter,
BitConverter.ToString(byteArrInt),valueInt,"Int");
Console.WriteLine(formatter,
BitConverter.ToString(byteArrShort),valueShort,"Short");
Console.WriteLine(formatter,
BitConverter.ToString(byteArrChar),valueChar,"Char");
Console.WriteLine(formatter,
BitConverter.ToString(byteArrBool),valueBool,"Boolean");
Console.WriteLine(formatter,
BitConverter.ToString(byteArrDecimal),valueDecimal,"Decimal");

OUTPUT:

Example of using methods in the BitConverter class to
convert values of specified data types into
corresponding byte arrays

 Data Type     Value               Byte Array
  ------------------               ----------
    Double    3.1415  6F-12-83-C0-CA-21-09-40
    Single     137.5              00-80-09-43
      Long 123456789  15-CD-5B-07-00-00-00-00
       Int      9732              04-26-00-00
     Short        25                    19-00
      Char         M                    4D-00
      Bool      True                       01
   Decimal   842.696  C8-DB-0C-00-00-00-00-00
                     -00-00-00-00-00-00-03-00

Example of using methods in the BitConverter class to
convert byte arrays back to values of a corresponding
specified data type
```

```
              Byte Array          Value      Data Type
                --------     ----------     ----------
 6F-12-83-C0-CA-21-09-40         3.1415         Double
             00-80-09-43          137.5         Single
 15-CD-5B-07-00-00-00-00      123456789           Long
             04-26-00-00           9732            Int
                   19-00             25          Short
                   4D-00              M           Char
                      01           True        Boolean
 C8-DB-0C-00-00-00-00-00        842.696        Decimal
-00-00-00-00-00-00-03-00
```

In general, a bit array, also known as a *bitmap* or a *bitset*, is simply an array data structure specifically designed to compactly store individual bits or Boolean values. Accordingly, the .NET Framework provides a very useful class, called `BitArray`, to manage compact arrays of bit values. This class comes loaded with a very nice collection of methods especially designed to facilitate not just bitwise operations but also various other aspects of bit manipulation. For example, the `BitArray` class constructor can be overloaded to accommodate boolean, bit and also byte arrays and these arrays are dynamically re-sizable. In addition, it is more memory efficient than both a simple array of bool or a generic List of bool because it uses only one bit for each element, whereas the bool type uses one byte for each element.

The `BitVector32` class is similar to the `BitArray` class in that it can also hold a packed array of Boolean values, one per bit. However, the `BitVector32` class is limited to only 32 bit elements and, because it is stack based, it is also faster than the `BitArray` class. In addition, the `BitVector32` class can store a set of small integers that take up at most 32 consecutive bits and is therefore very useful with bit-coded fields, such as those encountered when passing data to and from hardware devices.

A full, complete and updated description of all the features of the `BitArray`, the `BitVector32` and the `BitConverter` classes can be found at the MSDN Microsoft website [39] and so it is pointless to repeat all that information here. Instead, the following code snippet is meant to illustrate a few of the basic features of `BitArray` class methods. Additional examples will be provided later in this chapter as they are needed.

```
//Create a 10 element array of bits
BitArray bits = new BitArray(10);
//and a dynamic array of bytes.
byte[] byteArrInt32;
//Initialize bits array
for (int i = 0; i < bits.Length; i += 2)
{
   bits[i] = true;
}
foreach (bool bit in bits)
{
  //Display original contents of BitArray
  Console.WriteLine(bit);
  //Convert original contents of BitArray from
  //boolean to corresponding integer values and display
  Console.WriteLine(Convert.ToInt32(bit));

  //Convert bit data into an array of bytes and display
  byteArrInt32 = BitConverter.GetBytes(bit);
  Console.WriteLine(BitConverter.ToString(byteArrInt32));
}
```

There is a common but potentially serious source of error that may come up at runtime if we fail to properly consider numeric representations and their effect on conversion operations. Petrusha [40] has posted an excellent article on Microsoft's BCL Team Blog website that not only addresses this problem but also suggests some solutions. As a result, I have streamlined and summarized the key points of that article below for the convenience and benefit of my readers. This problem seems to surface most clearly when we convert the hexadecimal or octal string representation of a numeric value, that should be out of range of its target data type, back to its initial data type. For example, in running the following code, we would ordinarily expect to have an OverflowException thrown upon incrementing the upper range of a signed integer value by one, followed by a call to the Convert.ToString method to convert this integer value to its hexadecimal string representation before finally calling the Convert.ToInt32 method to convert the string back to an integer. The code snippet that follows illustrates this problem.

```
const int HEXADECIMAL = 16;
//Generate a number that it is out of range of the Int32 type.
long number = (long)int.MaxValue + 1;
//Convert the number to its hexadecimal string equivalent.
string numericString = Convert.ToString(number,HEXADECIMAL);
//Convert the number back to an integer.
//An OverflowException is expected but it is not thrown.
try
{
  int targetNumber = Convert.ToInt32(numericString, HEXADECIMAL);
  Console.WriteLine("0x{0} is equivalent to {1}.",
                    numericString, targetNumber);
}
catch (OverflowException)
{
  Console.WriteLine("0x{0} is out of the range of the Int32
                   data type.", numericString);
}
```

Instead of generating the expected OverflowException, this code produces what is apparently an erroneous result: 0x80000000 is equivalent to -2147483648. The source of this problem becomes immediately apparent if we examine the binary rather than the decimal or hexadecimal representations of this numeric operation. Starting with `Int32.MaxValue = 01111111111111111111111111111111` we see that every bit is set except for the last one, also known as the sign bit. Since this bit is not set, it indicates that the value is positive. By incrementing Int32.MaxValue by 1, we must assign the resulting output to a variable of data type Int64 in order to prevent exceeding the bounds of the Int32 data type and causing an OverflowException to be thrown. This action clears bit 0 through 30 and sets bit 31. Bits 32 through 62 remain unset and the sign bit in positiong 63 is set to 0 indicating that the resulting value is positive. However, since leading zeros are always dropped from the non-decimal string representations of numeric vlaues, the call to Convert.ToString(value, toBase) produces a binary string whose length reverts back to 32. Therefore, the unexpected output produced by the code above appears to be caused by two different programming errors. First, we inadvertently allowed the string representation of a 64-bit signed integer value to be interpreted as the string representation of a 32-bit signed integer value. Second, by ignoring how signed and unsigned integers are represented, we have allowed a positive integer to be misinterpreted as a signed negative integer.

Although conversions can still produce overflows at run time, the C# compiler enforces type safety by prohibiting implicit narrowing conversions. This constraint means that, in order to successfully compile code that performs a narrowing conversion, the programmer must explicitly use a C# casting operator. In our original example, by converting a numeric value to its string representation and then converting it back to a numeric value, we have bypassed the safeguards implemented by the C# compiler to inform us of the potential for experiencing data loss in a narrowing conversion. Therefore, it is up to programmers to be aware of and properly handle exceptions that may potentially be thrown by narrowing conversions. A much improved version of the original C# code correcting this problem is given below:

```
const int HEXADECIMAL = 16;
// Increment number so that it is out of range of the int type
long number = (long)int.MaxValue + 1;
// Convert the number to its hexadecimal string equivalent.
string numericString = Convert.ToString(number,HEXADECIMAL);
// Convert the number back to a long integer.
long targetNumber=Convert.ToInt64(numericString,HEXADECIMAL);
Console.WriteLine("0x{0} is equivalent to {1}.",
                  numericString, targetNumber);
```

The second source of error in our initial example is that we neglected to consider the effect that numeric representations have on conversion operations. Although this is a common source of errors in many programs and the C# compiler does provide some safeguards against data loss in narrowing conversions, there are no safeguards provided by the C# compiler when the programmer chooses to work directly with binary, octal or hexadecimal data either as a sequence of bits or with non-decimal string representation of numeric values. Actually this is true of any platform and is not limited to Microsoft Windows or the .NET Framework. Consequently, programmers need to be particularly cautious when writing code that performs bitwise operations and always make sure that both operands share the same binary representation. In addition, programmers also need to be especially careful when converting the string representation of a number to its numeric equivalent to always ensure that the numeric string representation is of the type expected by the conversion method or operator. Our initial example produced unexpected results because we passed the string representation of what turned out to be an *unsigned* 32-bit integer to a conversion method, `Convert.ToInt32(value,fromBase)`, that expected the value parameter to be the string representation of a *signed* 32-bit integer.

A clearer illustration of the kind of problems that may be encountered when working with binary values that have different numeric representations can perhaps be more easily seen when performing some bitwise operation on integers with different signs. Consider for example, the case of calculating the result of performing a bitwise AND operation on 16 and -3, i.e. `16 & -3`, which produces the rather unexpected result of 16 when run by the C# compiler. This result reflects the fact that the .NET Framework uses twos complement representation for negative integers and absolute magnitude representation for positive integers. However, one's complement representation is also in use by some platforms. The following C# code snippet can be incorporated into a larger software application and used to determine the actual complement representation method in use by a particular platform.

```
public class BinaryUtil
{
   public static bool IsTwosComplement()
   {
      return Convert.ToSByte("FF", 16) == -1;
   }

   public static bool IsOnesComplement()
   {
      return Convert.ToSByte("FE", 16) == -1;
   }
}
```

Performing the AND operation with integers that have different signs then requires that we use a common method to represent their values. The most common method is a sign and magnitude representation, which uses a variable to store the absolute value of a number and a separate Boolean variable to store its sign. Using this method of representation, we can define the AND operation as follows:

```
public static int PerformBitwiseAND(int operand1,int operand2)
{
    // Set flag if a parameter is negative.
    bool sign1 = Math.Sign(operand1) == -1;
    bool sign2 = Math.Sign(operand2) == -1;

    // Convert two's complement to its absolute magnitude.
    if (sign1)
        operand1 = ~operand1 + 1;
    if (sign2)
        operand2 = ~operand2 + 1;

    if (sign1 & sign2)
        return -1 * (operand1 & operand2);
    else
        return operand1 & operand2;
}
```

Likewise, we can define the OR bitwise operation for integers of different signs as follows:

```
public static int PerformBitwiseOR(int operand1,int operand2)
{
    // Set flag if a parameter is negative.
    bool sign1 = Math.Sign(operand1) == -1;
    bool sign2 = Math.Sign(operand2) == -1;

    // Convert two's complement to its absolute magnitude.
    if (sign1)
        operand1 = ~operand1 + 1;
    if (sign2)
        operand2 = ~operand2 + 1;
    if (sign1 & sign2)
        return -1 * (operand1 | operand2);
    else
        return operand1 | operand2;
}
```

Similarly, we can define the XOR bitwise operation for integers of different signs as follows:

```
public static int PerformBitwiseXOR(int operand1,int operand2)
{
    // Set flag if a parameter is negative.
    bool sign1 = Math.Sign(operand1) == -1;
    bool sign2 = Math.Sign(operand2) == -1;

    // Convert two's complement to its absolute magnitude.
    if (sign1)
        operand1 = ~operand1 + 1;
    if (sign2)
        operand2 = ~operand2 + 1;
    if (sign1 & sign2)
        return -1 * (operand1 ^ operand2);
    else
        return operand1 ^ operand2;
}
```

Finally, as for the issues raised when converting the string representation of a non-decimal number to a numeric value, the root of the problem seems to be that at the time of its creation, the string representation of a number is effectively disassociated from its underlying numeric value thus making it virtually impossible to determine the sign of that numeric string representation when it is converted back to a number. Nevertheless, that problem of restoring a non-decimal value from its string representation can be resolved by defining a structure that includes a field to indicate the sign of the decimal value. For example, the following structure includes a Boolean field, Negative, that is set to true when the numeric value from which a non-decimal string representation is derived is negative. It also includes a Value field that stores the non-decimal string representation of a number.

```
struct NumericString
{
   public bool Negative;
   public string Value;
}
```

By storing a sign flag together with the string representation of a non-decimal number, the tight coupling between the string representation of a number and its sign can be preserved. This feature is particularly useful if the string is later converted back to a numeric value. For example, the following code defines a static method named ConvertToSignedInteger that takes a single parameter (an instance of the NumericString structure defined previously) and returns an integer.

```
public static int ConvertToSignedInteger(NumericString stringValue)
{
   // Convert the string to an Int32.
   try
   {
      int number = Convert.ToInt32(stringValue.Value, 16);
      // Throw an exception if sign flag is positive but
      // the number is interpreted as negative.
      if ((!stringValue.Negative) && ((number & 0x80000000) ==
            0x80000000))
        throw new OverflowException(String.Format("0x{0}
        cannot be converted to an Int32.",stringValue.Value));
      else
        return number;
   }
   // Handle legitimate overflow exceptions.
   catch (OverflowException e)
   {
      throw new OverflowException(String.Format("0x{0} cannot
        be converted to an Int32.",stringValue.Value),e);
   }
}
```

As the reader may remember, the initial code example returned an erroneous result when we incremented Int32.MaxValue by 1, converted it to a hexadecimal string, and then converted the string back to an integer value. However, by using the NumericString structure and the ConvertToSignedInteger method just described, the output result is an OverflowException and the signs of numbers are correctly handled as illustrated in the following code:

```
public static void Main()
{
   // Define a number.
   Int64 number = (long)Int32.MaxValue + 1;
   // Define its hexadecimal string representation.
   NumericString stringValue;
   stringValue.Value = Convert.ToString(number, 16);
   stringValue.Negative = (Math.Sign(number) < 0);
   ShowConversionResult(stringValue);
   NumericString stringValue2;
   stringValue2.Value = Convert.ToString(Int32.MaxValue, 16);
   stringValue2.Negative = Math.Sign(Int32.MaxValue) < 0;
   ShowConversionResult(stringValue2);
   NumericString stringValue3;
   stringValue3.Value = Convert.ToString(-16, 16);
   stringValue3.Negative = Math.Sign(-16) < 0;
   ShowConversionResult(stringValue3);
}
```

```
private static void ShowConversionResult(NumericString stringValue)
{
   try
   {
      Console.WriteLine(ConversionLibrary.ConvertToSignedInteger(
          stringValue).ToString("N0"));
   }
   catch (OverflowException e)
   {
      Console.WriteLine("{0}: {1}", e.GetType().Name, e.Message);
   }
}

OUTPUT:
OverflowException: 0x80000000 cannot be converted to an Int32.
2,147,483,647
-16
```

6.4 Assorted Bits and Bytes

Developing fast, accurate and reliable code for handling bitwise operations is an important topic in both scientific and engineering software applications, particularly for those encountered when passing data to and from computer hardware devices. It seems therefore appropriate to end this chapter with a collection of bit manipulation routines aimed at providing my readers with a substantial set of tools which they can customize to fit their own individual needs. I will not pretend to have invented any of these bit handling recipes or tricks that follow. In fact, some are considered *general knowledge* and have been well known in the industry for years after having existed in one form or another in the more established C/C++ computer languages [41, 42, 43]. However, with the increasing popularity of C#, I felt it was important to collect as many of these bit handling recipes as possible and then lay them down into one cohesive unit, like this book, which can serve as a reliable reference source for my readers for years to come.

CIRCULAR SHIFT or BITWISE ROTATION: A circular shift or bitwise rotation is a bitwise operation that shifts all bits of its operand so that the vacant bit positions are not filled in with zeros but instead are filled with the bits that are shifted out of the sequence. For example, when we rotate bits left, we shift them off the right end of the value and reinsert them on the left. When we rotate bits right, we shift them off the left end of the value and reinsert them on the right. For example, if the original number is 0x2345 then a circular shift to the left by 4 bits gives 0x5234 and a circular shift to the right by 4 bits = 0x3452. Before introducing the actual bit shift routines, we must first come up with a way to determine which bits will be impacted upon rotation. This is accomplished by using a mask to preserve the bits which will be rotated. Then the left or right rotation is performed and finally, the bits are added back into their proper location. The following routine can be used to mask off the bits which will be rotated out of the variable.

```
enum RotateDirection {LEFT,RIGHT};

private static int CalcRotationMask(int bitstorotate, int direction)
{
   //Returns the mask for masking out the bits which will be
   //displaced in RotateBits

   int c;
   int mask = 0;
   const int maxbits = 32;

   if (bitstorotate == 0) return 0;

   c = Int32.MinValue;
   //c = 0x80000000 in hex or
   //c = 10000000000000000000000000000000 in binary
   mask = (c >> bitstorotate);
   if (direction == (int)RotateDirection.LEFT)
   {
      mask = (c >> (maxbits - bitstorotate));
      mask = ~mask;
   }
   else
      mask = (c >> bitstorotate);
   return mask;
}

private static int RotateBits(int value, int bitstorotate,
                              int sizet, int direction)
{
   //value = value to be rotated
   //bitstorotate = number of bits to rotate
   //sizet = size of the resultant value (8, 16, or 32)
   //returns the value rotated left/right by the number
   //of bitstorotate bits

   int tmprslt =0;
   int mask=0;
   int target=0;
   const int maxbits = 32;

   bitstorotate %= sizet;

   // perform the actual rotatation depending on direction
   if (direction == (int)RotateDirection.LEFT)
   {
      target = value;
      // determine which bits will be impacted by the rotate
      mask = CalcRotationMask(bitstorotate, direction);
      // save off the bits which will be impacted
      tmprslt = value & mask;
      // perform the actual rotatation depending on direction
      target = (value  >> bitstorotate);
      // now rotate the saved off bits so they are in their
      // proper place
      tmprslt <<= (sizet - bitstorotate);
   }
   else if (direction == (int)RotateDirection.RIGHT)
   {
      // determine which bits will be impacted by the rotate
      mask = CalcRotationMask(bitstorotate, direction);
      // shift mask into the correct place
      mask >>= (maxbits - sizet);
      // save off the bits which will be affected
      tmprslt = value & mask;
      // perform the actual rotatation depending on direction
      target = (value << bitstorotate);
      // now shift the saved off bits
      tmprslt >>= (sizet - bitstorotate);
   }

   // add the rotated bits back into their proper location)
   target |= tmprslt;

   // and return the result
   return target;
}

Test Program:
int a = 0x2345;
Console.WriteLine("Original Number To Test: "+DisplayBits(a,16));
Console.WriteLine("Rotate  LEFT by  4 bits: " +
DisplayBits(RotateBits(a,4,16,(int)RotateDirection.LEFT),16));
Console.WriteLine("Rotate RIGHT by  4 bits: " +
DisplayBits(RotateBits(a,4,16,(int)RotateDirection.RIGHT),16));

OUTPUT:
Original Number To Test: 00100011 01000101
Rotate LEFT by   4 bits: 01010010 00110100
Rotate RIGHT by  4 bits: 00110100 01010010
```

EXTRACTING HIGH/LOW WORD OF A NUMBER: Suppose you have a 32 bit integer value that contains information that you would like to extract from both its upper and lower 16 bits. By ANDing the number to another number with all of the high word set to 1, you zero out all of the low word bits leaving the high word intact. Similarly, by ANDing the number to another number with all of the low word set to 1, you zero out all the high word bits leaving the low word bits intact. The following code illustrates this kind of operation.

```
public static int GetHighWord(int value)
{ return (value & (0xFFFF << 16)); }

public static int GetLowWord(int value)
{ return (value & 0x0000FFFF); }
```

EXTRACTING HIGH/LOW BYTE OF A NUMBER: Although the methods presented above just accept 32 bit integer values, you can overload the methods above to also accept any other data types that you may require. For example, if you need to acquire the low or high byte of a 16-bit integer you can use the same tactics that were discussed above but in the slightly adjusted manner as shown below.

```
public static short GetHighByte(short shortValue)
{
   return (short)(shortValue & (0xFF << 8));
}

public static short GetLowByte(short shortValue)
{
   return (short)(shortValue & (short)0xFF);
}
```

MISCELLANEOUS BIT MANIPULATION UTILITIES: The following collection of bit manipulation routines is supposed to be pretty much self explanatory and is meant to provide a set of tools from which parts can be extracted according to individual needs.

```
//Returns x with the n bits that begin at position
//p inverted leaving the others bits unchanged.
private static uint Invert(uint x, int p, int n)
{
   return x ^ (~(~0U << n) << p);
}

// Reverse the bits in a single byte
public static byte ReverseBitsInAByte(byte inByte)
{
   byte result = 0x00;
   byte mask = 0x00;

   for (mask = 0x80; Convert.ToInt32(mask) > 0; mask >>= 1)
   {
      result >>= 1;
      byte tempbyte = (byte)(inByte & mask);
      if (tempbyte != 0x00)
         result |= 0x80;
   }
   return (result);
}

//Reverses the byte order in an int value (version 1)
public static int reverseBytes1(int value)
{
   int b1 = (value >> 0) & 0xff;
   int b2 = (value >> 8) & 0xff;
   int b3 = (value >> 16) & 0xff;
   int b4 = (value >> 24) & 0xff;
   return b1 << 24 | b2 << 16 | b3 << 8 | b4 << 0;
}
```

```
//Reverses the byte order in an int value (version 2)
public static int reverseBytes2(int value)
{
   Byte[] bytes = BitConverter.GetBytes(value);
   Array.Reverse(bytes);
   return BitConverter.ToInt32(bytes, 0);
}

//Reverses the bit order in an int value
public static int reverseBits(int value)
{
   int n = 0;
   int i;
   // loop through all the bits
   for (i = 0; i < 32; i++)
   {
      // add bit from value to 1 bit left shifted variable
      n = (n << 1) + (value & 1);
      // right shift bits by 1
      value >>= 1;
   }
   return n;
}

public static string ExtractHexDigits(string input)
{
   // Removes any characters that are not digits (like @)
   Regex isHexDigit =
   new Regex("[abcdefABCDEF\\d]+", RegexOptions.Compiled);
   string newnum = "";
   foreach (char c in input)
   {
      if (isHexDigit.IsMatch(c.ToString()))
        newnum += c.ToString();
   }
   return newnum;
}

//Puts a string into a byte array
public static byte[] StringToByteArr(string str)
{
 System.Text.ASCIIEncoding encoding=new System.Text.ASCIIEncoding();
 return encoding.GetBytes(str);
}

public static int FindPatternInsideBitArray(byte[] ba,string pattern)
{
   //Finds bit patterns inside a bit array
   int temp = 0;
   StringBuilder sb = new StringBuilder(8 * ba.Length);
   foreach (byte b in ba)
   {
      temp = int.Parse(Convert.ToString(b, 2));
      sb.Append(temp.ToString("00000000"));
   }
   string bitArray = sb.ToString()
   //Console.WriteLine("bitArray is {0}", bitArray);
   //Console.WriteLine("pattern  is {0}\n", pattern);
   int index = bitArray.IndexOf(pattern);
   return index;
}

public static string ConvertHexValuesToASCII(string HexValue)
{
  //Converts input data in hex values to readable characters
  string StrValue = "";
  while (HexValue.Length > 0)
  {
   // Use ToChar() to convert each ASCII value
   // (two hex digits) to the actual character
   StrValue += System.Convert.ToChar(
   System.Convert.ToUInt32(HexValue.Substring(0,2),16)).ToString();
   // Remove from the hex object the converted value
   HexValue = HexValue.Substring(2, HexValue.Length - 2);
  }
  return StrValue;
}

public static String ConvertASCIIValuesToHEX(string ASCIIValue)
{
 StringBuilder sBuffer = new StringBuilder();
 for (int i = 0; i < ASCIIValue.Length; i++)
 {
  sBuffer.Append(Convert.ToInt32(ASCIIValue[i]).ToString("x"));
 }
 return sBuffer.ToString().ToUpper();
}
```

Finally, here's one last handy little utility to have. It returns the radix of the given number.

```
//Returns the radix of the given number. For example:
//1st radix of 79981 is 1, 2nd radix of 79981 is 8
public static int getRadix(int number, int radix)
{
   return (int)(number/Math.Pow(10,radix-1)) % 10;
}
```

7

Interpolation

7.1 Introduction

As part of the intricate process of analyzing experimental data, one often has to plot the observed data points on a graph in order to determine whether some kind of mathematical relationship exists that can describe the observed results. The data points are usually plotted as a discrete set of N ordered pairs: $(x_0,y_0),(x_1,y_1),\ldots,(x_N,y_N)$ such that $x_0 < x_1 < \cdots < x_N$. Often the points are equally spaced but that is not always necessarily true. In any event, we know the value of the function $y_i = f(x_i)$ at each data point but we don't have a general analytic expression for $f(x)$ that would allow us to calculate values at an arbitrary point (x,y). It would thus be highly desirable to be able to accurately estimate $f(x)$ for arbitrary x. In some cases, this process will also allow us to draw a smooth curve through, and perhaps even beyond, the given set of data points. If the desired x value lies between the smallest and largest values of the given set of points (x_i,y_i), then the process is called *interpolation*. If x being sought is outside that given range, then the process is called *extrapolation* and the results are much less reliable, as many former stock market analysts can very likely attest.

Good interpolation methods should be able to provide their own error estimate. However, flawless error estimates do not really exist. For example, we could have a function that, for no apparent reason, suddenly takes off and oscillates wildly between any two tabulated points. Because of this possibility, interpolation schemes always presumes some degree of smoothness for the interpolated function. In addition, most practical interpolation schemes start at a nearby point and, as more information from other points is incorporated, then add a sequence of hopefully decreasing corrections. If the interpolation process goes well without any unpleasant surprises leading to incorrect results, then the last correction will be the smallest and can be used as an informal, though not rigorous, bound on the error. The number of points (minus one) used in an interpolation scheme is called the order of the interpolation. However, increasing the order does not necessarily increase the accuracy, especially in polynomial interpolation. In any event, the goal of this chapter is to briefly describe how the most popular interpolation schemes work and then provide C# routines to illustrate their use in practical applications.

7.2 Linear Interpolation

Linear interpolation is perhaps the easiest interpolation method to understand and to implement. With this method any two points, denoted by (x_0, y_0) and (x_1, y_1), are simply joined together by a straight line segment. The desired interpolation point, denoted by (x, y) and lying between (x_0, y_0) and (x_1, y_1), should therefore lie on this same line segment. The slope between either (x_0, y_0) and (x, y) or between (x, y) and (x_1, y_1) should therefore be equal to the slope between (x_0, y_0) and (x_1, y_1). Setting any of these two slopes equal to each other and solving for the unknown $y = f(x)$ given x will provide us with the desired result. More formally, suppose we have a finite set S of ordered pairs $(x_1, y_1), \ldots, (x_n, y_n)$ of real numbers such that $x_1 < x_2 < \cdots < x_n$. The linear interpolation function of S is a real-valued function f defined on $[x_1, x_n]$ such that, for $i = 1, \ldots, n-1$ we have

$$f(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) \quad \text{where} \quad x \in [x_i, x_{i+1}]$$

The implementation of this linear interpolation scheme is given below. The first routine calculates single point linear interpolation whereas the second routine calculates multiple point linear interpolation.

```
public static double LinearInterpolation(double[] x, double[] y,
    double xval)
{
    double yval = 0.0;
    for (int i = 0; i < x.Length - 1; i++)
    {
        if (xval >= x[i] && xval < x[i+1])
        {
            yval = y[i]+(xval-x[i])*(y[i+1]-y[i])/(x[i+1]-x[i]);
        }
    }
    return yval;
}

public static double[] LinearInterpolation(double[] x,double[] y,
    double[] xvals)
{
    double[] yvals = new double[xvals.Length];
    for (int i = 0; i < xvals.Length; i++)
        yvals[i] = LinearInterpolation(x, y, xvals[i]);
    return yvals;
}

static void TestLinear()
{
    double[] xdata = new double[] { 1, 3, 5, 7, 9 };
    double[] ydata = new double[] { 2, 6, 10, 14, 18 };
    double[] x = new double[] { 2, 4, 6, 8 };
    double[] y = LinearInterpolation(xdata, ydata, x);
    RVector xvec = new RVector(x);
    RVector yvec = new RVector(y);
    Console.Clear();
    Console.WriteLine("Running Linear Interpolation Test\n\n");
    Console.WriteLine(" x = " + xvec.ToString());
    Console.WriteLine(" y = " + yvec.ToString() + "\n\n");
    Console.WriteLine("Press ENTER key to continue...");
    Console.ReadLine();
}
```

```
OUTPUT: Running Linear Interpolation Test
x = (2, 4, 6, 8)
y = (4, 8, 12, 16)
```

7.3 Bilinear Interpolation

The bilinear interpolation scheme is just an extension of the linear interpolation method for interpolating functions of two variables, $z = f(x,y)$, on a rectangular grid. Bilinear interpolation uses four vertex values, one on each edge of a rectangular cell, in order to obtain an approximate value anywhere inside that same cell. The key idea is to perform linear interpolation first in one direction, and then again in the other direction. For example, suppose that we want to get an estimate of the value at the point (x,y), and the vertices of a rectangular cell are located at $z_{i,i} = f(x_i,y_i)$, $z_{i,i+1} = f(x_i,y_{i+1})$, $z_{i+1,i} = f(x_{i+1},y_i)$, and $z_{i+1,i+1} = f(x_{i+1},y_{i+1})$. We first do linear interpolation in the x-direction which gives

$$f(x,y_i) = \frac{x_{i+1}-x}{x_{i+1}-x_i} f(x_i,y_i) + \frac{x-x_i}{x_{i+1}-x_i} f(x_{i+1},y_i)$$

$$f(x,y_{i+1}) = \frac{x_{i+1}-x}{x_{i+1}-x_i} f(x_i,y_{i+1}) + \frac{x-x_i}{x_{i+1}-x_i} f(x_{i+1},y_{i+1})$$

Then we proceed by interpolating in the y-direction.

$$f(x,y) = \frac{y_{i+1}-y}{y_{i+1}-y_i} f(x,y_i) + \frac{y-y_i}{y_{i+1}-y_i} f(x,y_{i+1}).$$

Combining these three equations results in the desired estimate for $z = f(x,y)$

$$\begin{aligned} f(x,y) = {} & \frac{(x_{i+1}-x)(y_{i+1}-y)}{(x_{i+1}-x_i)(y_{i+1}-y_i)} f(x_i,y_i) + \frac{(x-x_i)(y_{i+1}-y)}{(x_{i+1}-x_i)(y_{i+1}-y_i)} f(x_{i+1},y_i) \\ & + \frac{(x_{i+1}-x)(y-y_i)}{(x_{i+1}-x_i)(y_{i+1}-y_i)} f(x_i,y_{i+1}) + \frac{(x-x_i)(y-y_i)}{(x_{i+1}-x_i)(y_{i+1}-y_i)} f(x_{i+1},y_{i+1}) \end{aligned}$$

The most striking feature of this bilinear interpolation equation for obtaining the function value $f(x,y)$ at the point (x,y) is that we essentially end up with a polynomial with four coefficients with powers of both x and y no greater than 1:

$$f(x,y) = p_0 + p_1 x + p_2 y + p_3 xy$$

Since these four coefficients are determined by four values, $z_{i,i}, z_{i,i+1}, z_{i+1,i}, z_{i+1,i+1}$, they are uniquely determined by the given data. This feature indicates that a comparable procedure of first interpolating along the *y*-axis and then interpolating the results in the *x*-direction will give the same unique result. In addition, we can see that the term *bilinear* comes from the process of linear interpolation twice in one direction and then once in the perpendicular direction and not from the formula for $f(x,y)$ which involves the non-linear term xy.

The implementation in C# of this bilinear interpolation scheme is given below. The first routine calculates single point bilinear interpolation, whereas the second routine calculates multiple point bilinear interpolation.

```
public static double BilinearInterpolation(double[] x, double[] y,
    double[,] z, double xval, double yval)
{
 double zval = 0.0;
 for (int i = 0; i < x.Length - 1; i++)
 {
  for (int j = 0; j < y.Length - 1; j++)
  {
   if (xval>=x[i] && xval<x[i+1] && yval>=y[j] && yval<y[j+1])
   {
zval=z[i,j]*(x[i+1]-xval)*(y[j+1]-yval)/(x[i+1]-x[i])/(y[j+1]-y[j]) +
     z[i+1,j]*(xval-x[i])*(y[j+1]-yval)/(x[i+1]-x[i])/(y[j+1]-y[j]) +
     z[i,j+1]*(x[i+1]-xval)*(yval-y[j])/(x[i+1]-x[i])/(y[j+1]-y[j]) +
     z[i+1,j+1]*(xval-x[i])*(yval-y[j])/(x[i+1]-x[i])/(y[j+1]-y[j]);
   }
  }
 }
 return zval;
}

public static double[] BilinearInterpolation(double[] x, double[] y,
    double[,] z, double[] xvals, double[] yvals)
{
 double[] zvals = new double[xvals.Length];
 for (int i = 0; i < xvals.Length; i++)
   zvals[i] = BilinearInterpolation(x,y,z,xvals[i],yvals[i]);
 return zvals;
}

static void TestBilinear()
{
    double[] xdata = new double[] { 0, 2 };
    double[] ydata = new double[] { 0, 2 };
    double[,] zdata = new double[,] { { 0, 20 }, { 20, 10 } };
    double[] x = new double[18];
    double[] y = new double[18];
    RMatrix z = new RMatrix(18, 18);
    for (int i = 0; i < 9; i++)
    {
        x[i] = (i + 2.0) / 20.0;
        y[i] = (i + 2.0) / 20.0;
    }
    RVector xvec = new RVector(x);
    RVector yvec = new RVector(y);

    for (int i = 0; i < 18; i++)
    {
      for (int j = 0; j < 18; j++)
      {
        z[i,j]=BilinearInterpolation(xdata,ydata,zdata,x[i],y[j]);
      }
    }
    Console.Clear();
    Console.WriteLine("Running Bilinear Interpolation Test\n\n");
    Console.WriteLine("x = " + xvec.ToString());
    Console.WriteLine("y = " + yvec.ToString());
    Console.WriteLine("\nResults z = \n" + z.ToString() + "\n\n");
    Console.WriteLine("Press ENTER key to continue...");
    Console.ReadLine();
}
```

```
OUTPUT: Running Bilinear Interpolation Test

x = (0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5,
        0, 0, 0, 0, 0, 0, 0, 0, 0)
y = (0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5,
        0, 0, 0, 0, 0, 0, 0, 0, 0)

Results z =
(1.925, 2.3875, 2.85, 3.3125, 3.775, 4.2375, 4.7, 5.1625, 5.625,
        1, 1, 1, 1, 1, 1, 1, 1, 1
 2.3875, 2.83125, 3.275, 3.71875, 4.1625, 4.60625, 5.05, 5.49375,
        5.9375, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5
 2.85, 3.275, 3.7, 4.125, 4.55, 4.975, 5.4, 5.825, 6.25, 2, 2, 2,
        2, 2, 2, 2, 2, 2
 3.3125, 3.71875, 4.125, 4.53125, 4.9375, 5.34375, 5.75, 6.15625,
        6.5625, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5
 3.775, 4.1625, 4.55, 4.9375, 5.325, 5.7125, 6.1, 6.4875, 6.875,
        3, 3, 3, 3, 3, 3, 3, 3, 3
 4.2375, 4.60625, 4.975, 5.34375, 5.7125, 6.08125, 6.45, 6.81875,
        7.1875, 3.5, 3.5, 3.5, 3.5, 3.5, 3.5, 3.5, 3.5, 3.5
 4.7, 5.05, 5.4, 5.75, 6.1, 6.45, 6.8, 7.15, 7.5, 4, 4, 4, 4, 4,
        4, 4, 4, 4
 5.1625, 5.49375, 5.825, 6.15625, 6.4875, 6.81875, 7.15, 7.48125,
        7.8125, 4.5, 4.5, 4.5, 4.5, 4.5, 4.5, 4.5, 4.5, 4.5
 5.625, 5.9375, 6.25, 6.5625, 6.875, 7.1875, 7.5, 7.8125, 8.125,
        5, 5, 5, 5, 5, 5, 5, 5, 5
 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0
 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0
 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0
 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0
 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0
 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0
 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0
 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0
 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

7.4 Polynomial Interpolation

Unfortunately, linear interpolation has some major drawbacks and is therefore not sufficiently well suited for use in all types of interpolation calculations, particularly when one seeks a smoother and more accurate fit of the individual data points. The most obvious and intuitive way to improve on the linear interpolation scheme is to replace the linear function with a polynomial of higher degree. The main idea behind polynomial interpolation is that given $n+1$ discrete data points there exits a unique smooth and infinitely differentiable polynomial of order n that goes through all those data points. Also, a polynomial of lower order is not generally possible and a polynomial of higher order is not unique.

However, polynomial interpolation also has some disadvantages which require some attention. For example, calculating the interpolating polynomial can be computationally expensive compared to linear interpolation. More importantly, in some cases, particularly with high degree polynomials near the endpoints, polynomial interpolation may not always be as accurate as one might hope or expect. This problem arises when the data points used to calculate the polynomial are somewhat distant from the interpolating point of interest. In such cases, the resulting higher-order polynomial, with its additional constrained points, tends to oscillate wildly between the tabulated values. In the end, this oscillation may have no relation at all to the behavior of the *true* fitting function and can thus lead to misleading or even completely erroneous results. Consequently, unless there is solid evidence that the interpolating function is close in form to the actual true function, it is a good idea to be cautious about high-order interpolation. In fact, Press et al. [22] enthusiastically recommend doing polynomial interpolations with no more than 3 or 4 points but frowns on those having as much as 5 or 6 points, and strongly discourages going any higher unless errors are closely monitored.

In spite of there being some significant shortcomings every now and then, the polynomial interpolation techniques discussed here still have enough desirable features provided one monitors their behavior properly, especially near the endpoints. In general, polynomial interpolation assumes that the data points are in some sense correct and lie on an underlying but unknown curve, and the goal is to be able to estimate the values of the curve at any position between the known points.

7.4.1 Lagrange Interpolation

Lagrange polynomial interpolation is an old popular interpolation method used to approximate a function $f(x)$ at an arbitrary point x which can be fitted for both equally and unequally spaced data. In a nutshell, given a set of $n+1$ data points

$$(x_0,y_0),(x_1,y_1),\dots,(x_n,y_n)$$

where no two x_i are the same, the interpolation polynomial in the Lagrange form is a linear combination

$$y = f(x) = \sum_{i=0}^{n} \ell_i(x) f(x_i)$$

where

$$\ell_i(x) = \prod_{j=0,\, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} = \frac{(x-x_0)}{(x_i-x_0)} \cdots \frac{(x-x_{i-1})}{(x_i-x_{i-1})} \frac{(x-x_{i+1})}{(x_i-x_{i+1})} \cdots \frac{(x-x_n)}{(x_i-x_n)}.$$

From this last expression we see that $\ell_i(x)$ is an n-th order polynomial with zeros at all of the sample points except for the n-th one.

The implementation in C# of this Lagrange interpolation scheme is given below. The first routine calculates single point Lagrange interpolation whereas the second routine calculates multiple point Lagrange interpolation.

```
public static double LagrangeInterpolation(double[] x, double[] y,
    double xval)
{
    double yval = 0.0;
    double Products = y[0];
    for (int i = 0; i < x.Length; i++)
    {
        Products = y[i];
        for (int j = 0; j < x.Length; j++)
        {
            if (i != j)
            {
                Products *= (xval-x[j]) / (x[i]-x[j]);
            }
        }
        yval += Products;
    }
    return yval;
}

public static double[] LagrangeInterpolation(double[] x, double[] y,
    double[] xvals)
{
    double[] yvals = new double[xvals.Length];
    for (int i = 0; i < xvals.Length; i++)
        yvals[i] = LagrangeInterpolation(x, y, xvals[i]);
    return yvals;
}

static void TestLagrangian()
{
    double[] xdata = new double[5] { 2, 4, 6, 8, 10 };
    double[] ydata = new double[5] { 2, 8, 18, 32, 50 };
    double[] x = new double[3] { 5.0, 7.0, 3.0 };

    double[] y = LagrangeInterpolation(xdata, ydata, x);
    RVector xvec = new RVector(x);
    RVector yvec = new RVector(y);
    Console.Clear();
    Console.WriteLine("Running Lagrangian Interpolation Test\n\n");
    Console.WriteLine(" x = " + xvec.ToString());
    Console.WriteLine(" y = " + yvec.ToString() + "\n\n");
    Console.WriteLine("Press ENTER key to continue...");
    Console.ReadLine();
}
```

```
OUTPUT: Running Lagrangian Interpolation Test
 x = (5, 7, 3)
 y = (12.5, 24.5, 4.5)
```

Unfortunately, the Lagrange polynomial interpolation method is widely regarded as being of mainly theoretical instead of practical interest, as reference to almost any numerical analysis textbook will reveal. Acton [44], for example, goes even further claiming that Lagrangian interpolation should be praised for its analytic utility and beauty but *deplored* for actual numerical practice. A short list highlighting a few of its main disadvantages include

- Each evaluation of $f(x)$ requires a complete computation of all the terms in the interpolation formula which means doing another $O(n^2)$ additions and multiplications.
- Adding a new data point (x_{N+1}, y_{N+1}) requires a complete computation of all the terms of the interpolation formula.
- Sometimes the computation may be unstable particularly near the endpoints.

However, other authors, such as Salzer [45], Werner [46] and Winrich [47] have noted that certain variants of the Lagrange formula are indeed useful in practice. As it turns out, the Lagrange representation of the interpolating polynomial can be rewritten in two more computationally attractive forms: a barycentric modified Lagrange form and a modified Lagrange form.

7.4.2 Barycentric Interpolation

Barycentric interpolation is a variant of Lagrange polynomial interpolation that is both fast and stable. As such, it is rapidly becoming the preferred method for doing practical calculations involving polynomial interpolation. Using the quantity

$$\ell(x) = (x - x_0)(x - x_1)\cdots(x - x_n)$$

we can rewrite the Lagrange basis polynomials as

$$\ell_i(x) = \frac{\ell(x)}{x - x_i}\frac{1}{\prod_{j=0, j\neq i}^{n}(x_i - x_j)}$$

or, by defining the barycentric weight functions

$$w_i = \frac{1}{\prod_{j=0, j\neq i}^{n}(x_i - x_j)}$$

we can simply write

$$\ell_i(x) = \ell(x)\frac{w_i}{x - x_i}$$

which is commonly referred to as the first form of the barycentric interpolation formula.
The advantage of this representation is that the interpolation polynomial may now be evaluated as

$$y = f(x) = \ell(x)\sum_{i=0}^{n}\frac{w_i}{x - x_i}f(x_i)$$

which, if the weights w_i have been pre-computed, requires only $O(n)$ operations as opposed to $O(n^2)$ for evaluating the Lagrange basis polynomials $\ell_i(x)$ individually.

The barycentric interpolation formula can also easily be updated to incorporate a new node x_{n+1} by dividing each of the $w_i, i = 0 \ldots n$ by $(x_i - x_n + 1)$ and constructing the new w_{n+1} as above.

We can further simplify the first form by first considering the barycentric interpolation of the constant function $g(x) \equiv 1$:

$$g(x) = \ell(x)\sum_{i=0}^{n}\frac{w_i}{x - x_i}$$

Dividing L(x) by g(x) does not modify the interpolation, yet yields

$$y = f(x) = \frac{\sum_{i=0}^{n}\frac{w_i}{x-x_i}f(x_i)}{\sum_{i=0}^{n}\frac{w_i}{x-x_i}}$$

which is referred to as the second form or true form of the barycentric interpolation formula. This second form has the advantage, that $\ell(x)$ need not be evaluated for each evaluation of $f(x)$.
The implementation in C# of this barycentric interpolation scheme is given below. The first routine calculates single point barycentric interpolation whereas the second routine calculates multiple point barycentric interpolation.

```
public static double BarycentricInterpolation(double[] x, double[] y,
     double xval)
{
    double product;
    double deltaX;
    double bc1 = 0;
    double bc2 = 0;

    int size = x.Length;
    double[] weights = new double[size];

    for (int i = 0; i < size; i++)
    {
        product = 1;
        for (int j = 0; j < size; j++)
        {
            if (i != j)
            {
                product *= (x[i] - x[j]);
                weights[i] = 1.0 / product;
            }
        }
    }

    for (int i = 0; i < size; i++)
    {
        deltaX = weights[i] / (xval - x[i]);
        bc1 += y[i] * deltaX;
        bc2 += deltaX;
    }
    return bc1 / bc2;
}

public static double[] BarycentricInterpolation(double[] x, double[]
    y, double[] xvals)
{
    double[] yvals = new double[xvals.Length];
    for (int i = 0; i < xvals.Length; i++)
        yvals[i] = BarycentricInterpolation(x, y, xvals[i]);
    return yvals;
}

static void TestBarycentric()
{
    double[] xdata = new double[] { 0, 4, 8, 12, 16 };
    double[] ydata = new double[] { 0, 8, 32, 72, 128 };
    double[] x = new double[] { 2, 6, 10, 14 };
    double[] y = BarycentricInterpolation(xdata, ydata, x);
    RVector xvec = new RVector(x);
    RVector yvec = new RVector(y);
    Console.Clear();
    Console.WriteLine("Running Barycentric Interpolation Test\n\n");
    Console.WriteLine(" x = " + xvec.ToString());
    Console.WriteLine(" y = " + yvec.ToString() + "\n\n");
    Console.WriteLine("Press ENTER key to continue...");
    Console.ReadLine();
}

OUTPUT: Running Barycentric Interpolation Test
 x = (2, 6, 10, 14)
 y = (2, 18, 50, 98)
```

7.4.3 Newton's Divided Differences Interpolation

In general, if you have $n+1$ data points $\{(x_0,y_0),(x_1,y_1),\ldots,(x_n,y_n)\}$ then Newton's interpolation polynomial can be written as

$$\begin{aligned}P_n(x) &= \sum_{k=0}^{n} a_k n_k(x) \\ &= a_0 + a_1(x-x_0) + a_2(x-x_0)(x-x_1) \\ &\quad + \ldots + a_n(x-x_0)(x-x_1)\cdots(x-x_{n-1})\end{aligned}$$

where the statement that $p(x)$ interpolates the given set of data points means that $P_n(x_i) = f(x_i)$, $\forall i = 0,\ldots,n$.

The Newton basis polynomial function, $n_k(x)$, is defined by:

$$n_k(x) = \prod_{i=0}^{k-1}(x-x_i) = (x-x_0)(x-x_1)\cdots(x-x_{k-1}), \quad k = 1,\ldots,n.$$

and the first few terms of $n_k(x)$ are given by

$$\begin{aligned}
n_0(x) &= 1 \\
n_1(x) &= (x-x_0) \\
n_2(x) &= (x-x_0)(x-x_1) \\
n_3(x) &= (x-x_0)(x-x_1)(x-x_2) \\
\vdots \;\; &\;\; \vdots \\
n_n(x) &= (x-x_0)(x-x_1)\cdots(x-x_{n-1})
\end{aligned}$$

Newton's interpolation polynomial, $P_n(x)$, of degree n evaluated at x_0, gives:

$$P_n(x_0) = \sum_{k=0}^{n} a_k n_k(x_0) = a_0 = f(x_0) = f[x_0]$$

Using the following notation to denote the i-th order divided difference

$$f[x_i] = f(x_i), \quad \forall i = 0,\ldots,n$$

we see that $f[x_0]$ is called the zero-order divided difference.

Similarly, Newton's interpolation polynomial, $P_n(x)$, of degree n evaluated at x_1, gives

$$\begin{aligned}
P_n(x_1) &= \sum_{k=0}^{n} a_k n_k(x_1) \\
&= a_0 + a_1(x_1-x_0) \\
&= f[x_0] + a_1(x_1-x_0) \\
&= f[x_1]
\end{aligned}$$

from which we can extract the first order divided difference

$$a_1 = \frac{f[x_1]-f[x_0]}{x_1-x_0} = f[x_0,x_1]$$

where $f[x_0,x_1]$ is called the first order divided difference.

Likewise, Newton's interpolation polynomial, $P_n(x)$, of degree n evaluated at x_2, gives

$$\begin{aligned}
P_n(x_2) &= \sum_{k=0}^{n} a_k n_k(x_2) \\
&= a_0 + a_1(x_2-x_0) + a_2(x_2-x_0)(x_2-x_1) \\
&= f[x_0] + f[x_0,x_1](x_2-x_0) + a_2(x_2-x_0)(x_2-x_1) \\
&= f[x_2]
\end{aligned}$$

from which we can solve for a_2 giving

$$\begin{aligned}
a_2(x_2-x_0)(x_2-x_1) &= f[x_2]-f[x_0]-f[x_0,x_1](x_2-x_0)\\
a_2 &= \frac{f[x_2]-f[x_0]-f[x_0,x_1](x_2-x_0)}{(x_2-x_0)(x_2-x_1)}\\
a_2 &= \frac{f[x_2]-f[x_0]}{(x_2-x_0)(x_2-x_1)}-\frac{f[x_0,x_1]}{x_2-x_1}\\
a_2 &= \frac{f[x_0,x_2]-f[x_0,x_1]}{x_2-x_1}
\end{aligned}$$

where $f[x_0,x_1,x_2]$ is called the second order divided difference.

In general, we obtain:

$$a_k = \frac{f[x_1,\ldots,x_k]-f[x_0,\ldots,x_{k-1}]}{x_k-x_0} = f[x_0,\ldots,x_k]$$

where $f[x_0,\ldots,x_k]$ is thus called a k^{th}-order divided difference.
In practice, when we want to determine, say the third order divided difference $f[x_0,x_1,x_2,x_3]$, we need the following quantities

$$\begin{bmatrix}
x_0 \ f[x_0] & & & \\
x_1 \ f[x_1] & f[x_0,x_1] & & \\
x_2 \ f[x_2] & f[x_1,x_2] & f[x_0,x_1,x_2] & \\
x_3 \ f[x_3] & f[x_2,x_3] & f[x_1,x_2,x_3] & f[x_0,x_1,x_2,x_3]
\end{bmatrix}$$

Hence

$$f[x_0,x_1,x_2,x_3] = \frac{f[x_1,x_2,x_3]-f[x_0,x_1,x_2]}{x_3-x_0}$$

Finally, Newtons interpolation polynomial of degree n is obtained via the successive divided differences as shown below

$$P_n(x) = f[x_0]+\sum_{k=1}^{n} f[x_0,\ldots,x_k]n_k(x)$$

The implementation in C# of Newton's divided difference interpolation scheme is given below. The first routine calculates single point Newton's divided difference interpolation whereas the second routine calculates multiple point Newton's divided difference interpolation.

```
public static double NewtonDividedDifferenceInterpolation(double[] x,
     double[] y, double xval)
{
  double yval;
  int size = x.Length;
  double[] tarray = new double[size];
  for (int i = 0; i < size; i++)
  {
    tarray[i] = y[i];
  }
  for (int i = 0; i < size - 1; i++)
  {
    for (int j = size - 1; j > i; j--)
    {
      tarray[j]=(tarray[j-1]-tarray[j])/(x[j-1-i]-x[j]);
    }
  }

  yval = tarray[size - 1];

  for (int i = size - 2; i >= 0; i--)
  {
    yval = tarray[i] + (xval-x[i])*yval;
  }
  return yval;
}

public static double[] NewtonDividedDifferenceInterpolation(double[]
    x, double[] y, double[] xvals)
{
  double[] yvals = new double[xvals.Length];
  for (int i = 0; i < xvals.Length; i++)
    yvals[i]=NewtonDividedDifferenceInterpolation(x,y,xvals[i]);
  return yvals;
}

static void TestNewtonDividedDifference()
{
    double[] xdata = new double[] { 50, 60, 70, 80, 90 };
    double[] ydata = new double[] { 75.0, 150.0, 200.0, 225.0, 250.0
        };
    double[] x = new double[] { 55, 65, 75, 85 };
    double[] y = NewtonDividedDifferenceInterpolation(xdata, ydata, x
        );
    RVector xvec = new RVector(x);
    RVector yvec = new RVector(y);
    Console.Clear();
    Console.WriteLine("Running Newton Divided Difference
        Interpolation Test\n\n");
    Console.WriteLine(" x = " + xvec.ToString());
    Console.WriteLine(" y = " + yvec.ToString() + "\n\n");
    Console.WriteLine("Press ENTER key to continue...");
    Console.ReadLine();
}

OUTPUT: Running Newton Divided Difference Interpolation Test
 x = (55, 65, 75, 85)
 y = (114.6484375, 178.7109375, 214.6484375, 234.9609375)
```

7.5 Cubic Spline Interpolation

Arbitrary functions on closed intervals may be approximated by the use of various different kinds of polynomials. However, because of the oscillatory nature of high-degree polynomials and the property that a small fluctuation over a small portion of the interval of interest can at times induce large fluctuations over the entire range, makes this method sometimes unreliable. An alternative approach is to divide the desired interval into a collection of subintervals and construct a different approximating polynomial on each subinterval. Approximation by functions of the this type is called *piecewise-polynomial approximation.*

The simplest piecewise-polynomial approximation is piecewise-linear interpolation which consists of joining a set of data points $(x_0,y_0),\dots,(x_n,y_n)$ by a series of straight lines. Unfortunately, this approach often results in no differentiability at the endpoints of the subintervals, which, in a geometrical context, means that the interpolating function is not *smooth.* Since physical conditions may at times require smoothness, the approximating function must therefore be made to be continuously differentiable.

The most common piecewise-polynomial approximation on an interval $[x_0,x_n]$ uses cubic polynomials between each successive pair of nodes and is called *cubic spline interpolation.* A general cubic polynomial consists of four constant coefficients and so there is sufficient flexibility in a cubic spline procedure to ensure that the interpolant is not only continuously differentiable on the interval, but also has a continuous second derivative. These coefficients bend the line just enough so that it passes through each of the data points without generating unwanted wild oscillations or breaks in continuity. Cubic splines are the most popular procedure for polynomial interpolation because they produce an interpolated function that is both smooth and continuous through the second derivative. The basic idea behind using a cubic spline is to fit a piecewise function of the form

$$S(x) = \begin{cases} S_1(x),\ x \in [x_1,x_2) \\ S_2(x),\ x \in [x_2,x_3) \\ \quad\cdots \\ S_{n-1}(x),\ x \in [x_{n-1},x_n) \end{cases}$$

where $S_i(x)$ is a third degree polynomial with coefficients a_i,b_i,c_i and d_i defined by

$$S_i(x) = a_i + b_i(x-x_i) + c_i(x-x_i)^2 + d_i(x-x_i)^3 \text{ for } x \in [x_i,x_{i+1}] \text{ and } i = 1,2,\dots,n-1$$

More formally, given a function $f(x)$ defined on an interval $[a,b]$ and a set of nodes $a = x_0 < x_1 < \dots < x_n = b$, a cubic spline interpolant $S(x)$ for $f(x)$ is a function that satisfies the following conditions:

(a) $S(x)$ is a cubic polynomial, denoted by $S_i(x)$, on the subinterval $[x_i,x_{i+1}]$ for each $i = 0,1,\dots,n-1$.

(b) $S_i(x_i) = f(x_i)$ and $S_i(x_{i+1}) = f(x_{i+1})$ for each $i = 0, 1, \ldots, n-1$.

(c) $S_{i+1}(x_{i+1}) = S_i(x_{i+1})$ for each $i = 0, 1, \ldots, n-2$.

(d) $S'_{i+1}(x_{i+1}) = S'_i(x_{i+1})$ for each $i = 0, 1, \ldots, n-2$.

(e) $S''_{i+1}(x_{i+1}) = S''_i(x_{i+1})$ for each $i = 0, 1, \ldots, n-2$.

(f) and one of the following set of boundary conditions is also satisfied:

$S''(x_0) = S''(x_n) = 0$ (free or natural boundary)

$S'(x_0) = f'(x_0)$ and $S'(x_n) = f'(x_n)$ (clamped boundary)

To construct the cubic spline polynomial for a given function $f(x)$, the conditions (a)-(f) given above are simply applied to the cubic polynomials

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \quad \text{for each} \quad i = 0, 1, \ldots, n-1.$$

Since $S_i(x_i) = a_i = f(x_i)$, condition (c) can be applied to obtain

$$a_{i+1} = S_{i+1}(x_{i+1}) = S_i(x_{i+1}) = a_i + b_i(x_{i+1} - x_i) + c_i(x_{i+1} - x_i)^2 + d_i(x_{i+1} - x_i)^3$$

for each $i = 0, 1, \ldots, n-2$. Since the terms $x_{i+1} - x_i$ are used repeatedly in this derivation, it is convenient to define $h_i = x_{i+1} - x_i$ for each $i = 0, 1, \ldots, n-1$. If we also define $a_n = f(x_n)$ then the equation

$$a_{i+1} = a_i + b_i h_i + c_i h_i^2 + d_i h_i^3$$

holds for each $i = 0, 1, \ldots, n-1$.

In a similar way, $b_n = S'(x_n)$ and observe that

$$S'_i(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2$$

implies that $S'_i(x_i) = b_i$ for each $i = 0, 1, \ldots, n-1$. Applying condition (d) gives

$$b_{i+1} = b_i + 2c_i h_i + 3d_i h_i^2$$

for each $i = 0, 1, \ldots, n-1$.

Another relationship between the coefficients of S_i can be obtained by observing that $S''(x_n) = 2c_n$ and applying condition (e). Then, for each $i = 0, 1, \ldots, n-1$ we have

$$c_{i+1} = c_i + 3d_i h_i$$

Solving for d_i and substituting this value into the equations for a_{i+1} and b_{i+1} gives, for each $i = 0, 1, \ldots, n-1$, the new equations

$$a_{i+1} = a_i + b_i h_i + \frac{h_i^2}{3}(2c_i + c_{i+1})$$

and

$$b_{i+1} = b_i + h_i(c_i + c_{i+1})$$

The final relationship involving the coefficients is obtained by solving the previous two equations above, first for b_i,

$$b_i = \frac{1}{h_i}(a_{i+1} - a_i) - \frac{h_i}{3}(2c_i + c_{i+1})$$

and then, with a reduction of the index, for b_{i-1} which gives

$$b_{i-1} = \frac{1}{h_{i-1}}(a_i - a_{i-1}) - \frac{h_{i-1}}{3}(2c_{i-1} + c_i)$$

Substituting these values into the equation derived from the earlier equation for b_{i+1},

$$b_{i+1} = b_i + h_i(c_i + c_{i+1})$$

with the index reduced by one, gives the linear system of equations:

$$h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_i c_{i+1} = \frac{3}{h_i}(a_{i+1} - a_i) - \frac{3}{h_{i-1}}(a_i - a_{i-1})$$

for each $i = 1, 2, \ldots, n-1$. This system involves only the $\{c_i\}_{i=0}^{n}$ as unknowns since the values of $\{h_i\}_{i=0}^{n-1}$ and $\{a_i\}_{i=0}^{n}$ are given, respectively, by the spacing of the nodes $\{x_i\}_{i=0}^{n}$ and the values of $f(x)$ at the nodes. Once the values of $\{c_i\}_{i=0}^{n}$ are determined, it is a simple matter to find the remainder of the constants $\{b_i\}_{i=0}^{n-1}$ and $\{d_i\}_{i=0}^{n-1}$from the relationships derived above and then construct the cubic polynomials $\{S_i(x)\}_{i=0}^{n-1}$.

7.5.1 Natural Cubic Splines

The natural cubic splines satisfy the condition: $S''(x_0 = a) = S''(x_n = b) = 0$. This means that $S''(x_n)/2 = c_n = 0$ and that $S''(x_0) = 2c_0 + 6d_0(x_0 - x_0) = 0$ so that $c_0 = 0$. The two equations $c_0 = 0$ and $c_n = 0$ together with the linear system of equations

$$h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_i c_{i+1} = \frac{3}{h_i}(a_{i+1} - a_i) - \frac{3}{h_{i-1}}(a_i - a_{i-1})$$

derived earlier produce a linear system described by the matrices $Ax = b$ where A is the $(n+1) \times (n+1)$ matrix

$$A = \begin{pmatrix} 1 & 0 & 0 & \ldots & \ldots & 0 \\ h_0 & 2(h_0 + h_1) & h_1 & 0 & \ldots & 0 \\ 0 & h_1 & 2(h_1 + h_2) & h_2 & 0\ldots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & h_{n-2} & 2(h_{n-2} + h_{n-1}) & h_{n-1} \\ 0 & 0 & 0 & \ldots & 0 & 1 \end{pmatrix}$$

and b and x are vectors given by

$$b = \begin{pmatrix} 0 \\ \frac{3}{h_1}(a_2 - a_1) - \frac{3}{h_0}(a_1 - a_0) \\ \vdots \\ \frac{3}{h_{n-1}}(a_n - a_{n-1}) - \frac{3}{h_{n-2}}(a_{n-1} - a_{n-2}) \\ 0 \end{pmatrix} \quad \text{and} \quad x = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix}$$

The C# implementation below constructs the cubic spline interpolant $S(x)$ for the function $f(x)$, defined at the numbers $x_0 < x_1 < \ldots < x_n$ satisfying the natural boundary conditions $S''(x_0) = S''(x_n) = 0$.

```
public static void NaturalCubicSpline(double[] x, double[] y, double
    xvalue)
{
  int i, j, m;
  double S = 0.0;
  double delta = 0.0;

  int n = 4; //max number of coefficients: A,B,C,D for cubic spline

  double[] A = new double[n + 1];
  double[] B = new double[n + 1];
  double[] C = new double[n + 1];
  double[] D = new double[n + 1];
  double[] H = new double[n + 1];
  double[] XA = new double[n + 1];
  double[] XL = new double[n + 1];
  double[] XU = new double[n + 1];
  double[] XZ = new double[n + 1];

  for (i = 0; i < n; i++) A[i] = y[i];

  m = n - 1;
  for (i = 0; i <= m; i++)
    H[i] = x[i + 1] - x[i];

  for (i = 1; i <= m; i++)
    XA[i] = 3 * (A[i+1] * H[i-1] - A[i] * (x[i+1] - x[i-1]) +
                 A[i-1] * H[i]) / (H[i] * H[i-1]);

  XL[0] = 1; XU[0] = 0; XZ[0] = 0;
  for (i = 1; i <= m; i++)
  {
    XL[i] = 2 * (x[i + 1] - x[i - 1]) - H[i - 1] * XU[i - 1];
    XU[i] = H[i] / XL[i];
    XZ[i] = (XA[i] - H[i - 1] * XZ[i - 1]) / XL[i];
  }
  XL[n] = 1; XZ[n] = 0; C[n] = 0;

  for (i = 0; i <= m; i++)
  {
    j = m - i;
    C[j] = XZ[j] - XU[j] * C[j + 1];
    B[j] = (A[j+1] - A[j]) / H[j] - H[j] * (C[j+1] + 2*C[j])/3;
    D[j] = (C[j + 1] - C[j]) / (3 * H[j]);
  }

  Console.WriteLine("\n\nTesting natural cubic splines");
  Console.WriteLine("Input data stored in (x[i],y[i]).
                   Value to interpolate = {0}\n", xvalue);
  Console.WriteLine("i\t"+"x[i]\t"+"y[i]\t"+"A[i]\t"+"B[i]\t"+
                  "C[i]\t"+"D[i]\n");
  for (i = 0; i < n; i++)
  {
    Console.WriteLine(i.ToString() + "\t" + x[i].ToString("0.0000") +
    "\t" + y[i].ToString("0.0000") + "\t" + A[i].ToString("0.0000") +
    "\t" + B[i].ToString("0.0000") + "\t" + C[i].ToString("0.0000") +
    "\t" + D[i].ToString("0.0000") + "\n");
  }

  for (i = 0; i <= m; i++)
  {
   if (xvalue >= x[i] && xvalue < x[i + 1])
   {
     delta = xvalue - x[i];
     S = A[i]+B[i]*delta+C[i]*delta*delta+D[i]*delta*delta*delta;
     Console.WriteLine("Interpolated x value=f({0})={1}\n",xvalue,S);
   }
  }
}

public static void NaturalCubicSplineTest()
{
    int size = 26;
    double[] xdata = new double[size];
    double[] ydata = new double[size];

    xdata[0] = 0.1;
    xdata[1] = 0.2;
    xdata[2] = 0.3;
    xdata[3] = 0.4;

    ydata[0] = -0.62049958;
    ydata[1] = -0.28398668;
    ydata[2] = 0.00660095;
    ydata[3] = 0.24842440;

    NaturalCubicSpline(xdata, ydata, 0.25);
    Console.ReadLine();
}

static void Main(string[] args)
{
  NaturalCubicSplineTest();
}
```

```
OUTPUT: Testing natural cubic splines
Input data stored in (x[i],y[i]). Value to interpolate = 0.25

i   x[i]       y[i]      A[i]    B[i]      C[i]     D[i]

0   0.1000   -0.6205   -0.6205   3.4370   0.0000  -7.1897
1   0.2000   -0.2840   -0.2840   3.2213  -2.1569  -9.9769
2   0.3000    0.0066    0.0066   2.4906  -5.1500  44.2584
3   0.4000    0.2484    0.2484   2.7884   8.1275   6.7729

Interpolated x value = f(0.25) = -0.129559271569149
```

7.5.2 Clamped Cubic Splines

Clamped cubic splines satisfy the condition: $S'(x_0) = f'(x_0)$ and $S'(x_n) = f'(x_n)$. Since $S'(a) = f'(a) = S'(x_0) = b$ then by the equation

$$b_i = \frac{1}{h_i}(a_{i+1} - a_i) - \frac{h_i}{3}(2c_i + c_{i+1})$$

which was derived earlier and now with $i = 0$ implies

$$f'(a) = \frac{1}{h_0}(a_1 - a_0) - \frac{h_0}{3}(2c_0 - c_1)$$

As a result,

$$2h_0c_0 + h_0c_1 = \frac{3}{h_0}(a_1 - a_0) - 3f'(a)$$

Similarly, $f'(b) = b_n = b_{n-1} + h_{n-1}(c_{n-1} + c_n)$ and by this same equation above for b_i but now with $i = n-1$ implies that

$$f'(b) = \frac{a_n - a_{n-1}}{h_{n-1}} - \frac{h_{n-1}}{3}(2c_{n-1} + c_n) + h_{n-1}(c_{n-1} + c_n) = \frac{a_n - a_{n-1}}{h_{n-1}} + \frac{h_{n-1}}{3}(c_{n-1} + 2c_n)$$

and so

$$h_{n-1}c_{n-1} + 2h_{n-1}c_n = 3f'(b) - \frac{3}{h_{n-1}}(a_n - a_{n-1})$$

Then the system of linear equations

$$h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_ic_{i+1} = \frac{3}{h_i}(a_{i+1} - a_i) - \frac{3}{h_{i-1}}(a_i - a_{i-1})$$

derived earlier together with

$$2h_0c_0 + h_0c_1 = \frac{3}{h_0}(a_1 - a_0) - 3f'(a)$$

and

$$h_{n-1}c_{n-1} + 2h_{n-1}c_n = 3f'(b) - \frac{3}{h_{n-1}}(a_n - a_{n-1})$$

determine a linear system of the form $Ax = b$ where now

$$A = \begin{pmatrix} 2h_0 & h_0 & 0 & \dots & \dots & 0 \\ h_0 & 2(h_0+h_1) & h_1 & 0 & \dots & 0 \\ 0 & h_1 & 2(h_1+h_2) & h_2 & 0\dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & 0 & 0 & \dots & h_{n-1} & 2h_{n-1} \end{pmatrix}$$

and b and x are vectors given by

$$b = \begin{pmatrix} \frac{3}{h_0}(a_1 - a_0) - 3f'(a) \\ \frac{3}{h_1}(a_2 - a_1) - \frac{3}{h_0}(a_1 - a_0) \\ \vdots \\ \frac{3}{h_{n-1}}(a_n - a_{n-1}) - \frac{3}{h_{n-2}}(a_{n-1} - a_{n-2}) \\ 3f'(b) - \frac{3}{h_{n-1}}(a_n - a_{n-1}) \end{pmatrix} \quad \text{and} \quad x = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix}$$

The C# implementation below constructs the cubic spline interpolant $S(x)$ for the function $f(x)$, defined at the numbers $x_0 < x_1 < \dots < x_n$ satisfying the clamped boundary conditions $S'(x_0) = f'(x_0)$ and $S'(x_n) = f'(x_n)$.

```
public static void ClampedCubicSpline(double[] x, double[] y, double
    fp0, double fpn, double xvalue)
{
  int i, j, m;
  double S = 0.0;
  double delta = 0.0;

  int n = 4; //max number of coefficients: A,B,C,D for cubic spline

  double[] A = new double[n + 1];
  double[] B = new double[n + 1];
  double[] C = new double[n + 1];
  double[] D = new double[n + 1];
  double[] H = new double[n + 1];
  double[] XA = new double[n + 1];
  double[] XL = new double[n + 1];
  double[] XU = new double[n + 1];
  double[] XZ = new double[n + 1];

  for (i = 0; i < n; i++) A[i] = y[i];

  m = n - 1;
  for (i = 0; i <= m; i++)
     H[i] = x[i + 1] - x[i];
  XA[0] = 3 * (A[1] - A[0]) / H[0] - 3 * fp0;
  XA[n] = 3 * fpn - 3 * (A[n] - A[n - 1]) / H[n - 1];

  for (i = 1; i <= m; i++)
     XA[i]=3*(A[i+1]*H[i-1]-A[i]*(x[i+1]-x[i-1]) +
             A[i-1]*H[i])/(H[i]*H[i-1]);

  XL[0] = 2 * H[0]; XU[0] = 0.5; XZ[0] = XA[0] / XL[0];
  for (i = 1; i <= m; i++)
  {
    XL[i] = 2 * (x[i+1] - x[i-1]) - H[i-1] * XU[i-1];
    XU[i] = H[i] / XL[i];
    XZ[i] = (XA[i] - H[i - 1] * XZ[i - 1]) / XL[i];
  }
  XL[n] = H[n - 1] * (2 - XU[n - 1]);
  XZ[n] = (XA[n] - H[n - 1] * XZ[n - 1]) / XL[n];
  C[n] = XZ[n];

  for (i = 1; i <= n; i++)
  {
    j = n - i;
    C[j] = XZ[j]-XU[j]*C[j+1];
    B[j] = (A[j+1]-A[j])/H[j]-H[j]*(C[j+1] + 2*C[j])/3;
    D[j] = (C[j+1]-C[j])/(3*H[j]);
  }

  Console.WriteLine("\n\nTesting clamped cubic splines");
  Console.WriteLine("Input data stored in (x[i],y[i]). Value to
      interpolate = {0}\n", xvalue);
  Console.WriteLine("i\t"+"x[i]\t"+"y[i]\t"+"A[i]\t"+"B[i]\t"+"C[i]\t
      "+"D[i]\n");
  for (i = 0; i < n; i++)
  {
    Console.WriteLine(i.ToString()+"\t"+x[i].ToString("0.0000") +
      "\t"+y[i].ToString("0.0000")+"\t"+A[i].ToString("0.0000")+
      "\t" + B[i].ToString("0.0000")+"\t"+C[i].ToString("0.0000")+
      "\t"+D[i].ToString("0.0000")+"\n");
  }

  for (i = 0; i <= m; i++)
  {
   if (xvalue >= x[i] && xvalue < x[i + 1])
   {
     delta = xvalue - x[i];
     S = A[i]+B[i]*delta+C[i]*delta*delta+D[i]*delta*delta*delta;
     Console.WriteLine("Interpolated x value=f({0})={1}\n",xvalue,S);
   }
  }
}

public static void ClampedCubicSplineTest()
{
    int size = 26;
    double[] xdata = new double[size];
    double[] ydata = new double[size];
    xdata[0] = 0;
    xdata[1] = 1;
    xdata[2] = 2;
    xdata[3] = 3;

    ydata[0] = 1;
    ydata[1] = 2.718281828;
    ydata[2] = 7.389056099;
    ydata[3] = 20.08553692;

    double fp_0 = 1.0;
    double fp_n = 20.0855369;

    ClampedCubicSpline(xdata, ydata, fp_0, fp_n, 2.0);
    Console.ReadLine();
}

static void Main(string[] args)
{
  ClampedCubicSplineTest();
}

OUTPUT: Testing clamped cubic splines
Input data stored in (x[i],y[i]). Value to interpolate = 2

i    x[i]     y[i]     A[i]     B[i]     C[i]     D[i]

0   0.0000   1.0000   1.0000   1.0000   0.1958   0.5225
1   1.0000   2.7183   2.7183   2.9591   1.7633  -0.0515
2   2.0000   7.3891   7.3891   6.3310   1.6087   4.7569
3   3.0000  20.0855  20.0855  23.8189  15.8792   3.3904

Interpolated x value = f(2) = 7.389056099


static void Main(string[] args)
{
    TestLinear();
    TestBilinear();
    TestLagrangian();
    TestBarycentric();
    TestNewtonDividedDifference();
    NaturalCubicSplineTest();
    ClampedCubicSplineTest();
}
```

8

Linear Equations

8.1 Introduction

Linear equations, including their theory and various methods of solution, have many applications in both pure and applied mathematics, the natural sciences, and engineering [48]. In image processing, for example, digital images are displayed on the screen by a fairly large number of tiny little squares called pixels which store critical information about tiny individual picture elements. This information in turn can be treated as an array of numbers called a matrix which assigns a whole number to each pixel. For example, in the case of a 256×256 pixel gray scale image, the image is stored as a 256×256 matrix, with each element of the matrix being a whole number ranging from 0 for black to 256 for white. One can then use some linear algebra techniques to manipulate and enhance the image and also compress it for storage.

A general system of m linear equations with n unknowns can be written as

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
\vdots \qquad \vdots \qquad\qquad \vdots \quad &\ \ \vdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m.
\end{aligned}$$

Here $x_1, x_2, \ldots, x_n$ are the unknowns, $a_{11}, a_{12}, \ldots, a_{mn}$ are the coefficients of the system, and $b_1, b_2, \ldots, b_m$ are the constant terms. The coefficients and unknowns can be integers, real or complex numbers. One very popular and useful approach is to take each unknown as a weight for a column vector in a linear combination as shown below.

$$x_1 \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} + x_2 \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} + \cdots + x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

This method allows all the language and theory of vector spaces to be brought to bear. For example, the collection of all possible linear combinations of the vectors on the left-hand side is called their span, and the equations have a solution just when the right-hand vector is within that span. If every vector within that span has exactly one expression as a linear combination of the given left-hand vectors, then any solution is unique. In any event, the span has a basis of linearly independent vectors that do guarantee exactly one expression and the number of vectors in that basis, also known as its dimension, cannot be larger than m or n, but it can be smaller. This is important because if we have m independent vectors then a solution is guaranteed regardless of the right-hand side. Otherwise it is not guaranteed.

The vector equation is equivalent to a matrix equation of the form $A\mathrm{x} = \mathrm{b}$ where A is an $m \times n$ matrix, x is a column vector with n entries, and b is a column vector with m entries.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathrm{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathrm{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

The number of vectors in a basis for the span is now expressed as the rank of the matrix. A solution of a linear system is an assignment of values to the variables $x_1, x_2, \ldots, x_n$ such that each of the equations is satisfied. The set of all possible solutions is called the solution set. A linear system may behave in any one of three possible ways:

- The system has infinitely many solutions.
- The system has a single unique solution.
- The system has no solutions.

In general, the behavior of a linear system is determined by the relationship between the number of equations and the number of unknowns:

- Usually, a system with fewer equations than unknowns has infinitely many solutions.
- Usually, a system with the same number of equations and unknowns has a single unique solution.
- Usually, a system with more equations than unknowns has no solution.

In the first case, the dimension of the solution set is usually equal to $n - m$, where n is the number of variables and m is the number of equations. The equations of a linear system are consistent if they possess a common solution, and inconsistent otherwise. When the equations are inconsistent, it is possible to derive a contradiction from the equations, such as in the statement that $0 = 1$. The equations of a linear system are independent if none of the equations can be derived algebraically from the others. When the equations are independent, each equation contains new information about the variables, and removing any of the equations increases the size of the solution set. Two linear systems using the same set of variables are equivalent if each of the equations in the second system can be derived algebraically from the equations in the first system, and vice-versa. Equivalent systems convey precisely the same information about the values of the variables. In particular, two linear systems are equivalent if and only if they have the same solution set. When the solution set is finite, it is usually described in set notation. However, it can be difficult to describe a set with infinite solutions. Typically, some of the variables are designated as free, or independent, or as parameters, meaning that they are allowed to take any value, while the remaining variables are dependent on the values of the free variables.

The simplest, but perhaps also the longest and most tedious, method for solving a system of linear equations is to repeatedly eliminate variables. This method can be summarized as follows:

- In the first equation, solve for the one of the variables in terms of the others.
- Plug this expression into the remaining equations. This yields a system of equations with one less equation and one less unknown.
- Continue until you have reduced the system to a single linear equation.
- Solve this equation, and then back-substitute until the entire solution is found.

8.2 Gaussian Elimination

Gaussian elimination is an efficient algorithm for solving systems of linear equations, producing both the solution of the equations and the inverse of the coefficient matrix. Elementary row operations are used to reduce a matrix to row echelon form. An extension of this algorithm, Gauss-Jordan elimination, reduces the matrix further to reduced row echelon form. However, Gaussian elimination alone is sufficient for many applications [48]. A matrix is said to be in row-echelon form if it satisfies the following two conditions: (1) Each row contains only zeros until the first non-zero element, which must be 1. (2) As the rows are followed from top to bottom, the first non-zero element, also called the leading coefficient or pivot, occurs further to the right than in the previous row above it. A matrix is in reduced row echelon form, also called row canonical form, if it satisfies all the conditions above and, in addition, if every leading coefficient is 1 and the only nonzero entry in its column. That is, the entries above and below the first 1 in each row must all be 0.

The process of Gaussian elimination has two parts. The first part, called forward elimination, reduces a given system to either triangular or echelon form, or results in a degenerate equation meaning that the system has no solution. This step is accomplished through the use of elementary row operations. The second part uses back substitution to find the solution of the system of equations. Stated equivalently for matrix formalism, the first part reduces a matrix to row echelon form using elementary row operations while the second part simplifies it even further to reduced row echelon form, or row canonical form. Another point of view, which turns out to be very useful to analyze the algorithm, is that Gaussian elimination computes a matrix decomposition. The three elementary row operations used in Gaussian elimination, multiplying rows, switching rows, and adding multiples of rows to other rows, amount to multiplying the original matrix with invertible matrices from the left. The first part of the algorithm computes an *LU* decomposition, while the second part writes the original matrix as the product of a uniquely determined invertible matrix and a uniquely determined reduced row-echelon matrix.

In summary, Gaussian Elimination is considered the workhorse of computational methods for the solution of a system of linear equations. Gaussian Elimination consists of a systematic application of elementary row operations to a system of linear equations in order to convert it to upper triangular form. Once the coefficient matrix is in upper triangular form, we use back substitution to find a solution. The general procedure for Gaussian Elimination can be summarized in the following steps:

- Write the augmented matrix for the system of linear equations.
- Use elementary row operations on the augmented matrix $[A|b]$ to transform A into upper triangular form. If a zero is located on the diagonal, switch the rows until a nonzero is in that place. If you are unable to do so, then stop because the system has either infinite or no solutions.
- Use back substitution to find the solution of the problem.

8.3 Gauss-Jordan Elimination

Gauss-Jordan Elimination is a variant of Gaussian Elimination. Again, we are transforming the coefficient matrix into another matrix that is much easier to solve, and the system represented by the new augmented matrix has the same solution set as the original system of linear equations. In Gauss-Jordan Elimination, the goal is to produce a triangular matrix of coefficients with all zero elements below the principal diagonal. The general procedure for Gauss-Jordan Elimination can be summarized in the following three steps:

- Write the augmented matrix for the system of linear equations.
- Use elementary row operations on the augmented matrix $[A|b]$ to transform A into diagonal form. If a zero or a very small number is located on the diagonal, switch the rows until a nonzero is in that place. If you are unable to do so, then stop because the system has either infinite or no solutions.
- By dividing the diagonal element and the right-hand-side element in each row by the diagonal element in that row, make each diagonal element equal to one.

The following code illustrates the Gauss-Jordan elimination process in more detail.

```
const double epsilon = 1.0e-500;
// Gauss-Jordan elimination to solve Ax = b for x
public static RVector GaussJordan(RMatrix A, RVector b)
{
  Triangulate(A, b);
  int bSize = b.GetVectorSize;
  RVector x = new RVector(bSize);
  for (int i = bSize - 1; i >= 0; i--)
  {
    double Aii = A[i, i];
    if (Math.Abs(Aii) < epsilon)
      throw new Exception("Diagonal element is too small!");
    x[i] = (b[i]-RVector.DotProduct(A.GetRowVector(i),x))/Aii;
  }
  return x;
}

// Triangulate matrix A
private static void Triangulate(RMatrix A, RVector b)
{
  int nRows = A.GetnRows;
  for (int i = 0; i < nRows - 1; i++)
  {
    double diagonalElement = pivotGaussJordan(A, b, i);
    if (Math.Abs(diagonalElement) < epsilon)
      throw new Exception("Diagonal element is too small!");
    for (int j = i + 1; j < nRows; j++)
    {
      double w = A[j, i] / diagonalElement;
      for (int k = i + 1; k < nRows; k++)
      {
        A[j, k] -= w * A[i, k];
      }
      b[j] -= w * b[i];
    }
  }
}

private static double pivotGaussJordan(RMatrix A, RVector b, int q)
{
  int bSize = b.GetVectorSize;
  int c = q;
  double d = 0.0;
  for (int j = q; j < bSize; j++)
  {
    double w = Math.Abs(A[j, q]);
    if (w > d)
    {
        d = w;
        c = j;
    }
  }
  if (c > q)
  {
    A.SwapMatrixRow(q, c);
    b.SwapVectorEntries(q, c);
  }
  return A[q, q];
}

static void TestGaussJordan()
{
  RMatrix A =
        new RMatrix(new double[3,3] {{2,4,-6 },{6,-4,2 },{4,2,6}});
  RVector b = new RVector(new double[3] {6,-2,4});
  RVector x = GaussJordan(A, b);
  Console.WriteLine("Solution x = {0}",x);
}

static void Main(string[] args)
{
  TestGaussJordan();
  Console.ReadLine();
}

OUTPUT: Solving A x = b for x
Solution x=(0.476190476190476,1.19047619047619,-0.0476190476190476)
```

8.4 LU Decomposition

A matrix decomposition is a factorization of a matrix into some canonical form. A triangular matrix is a special kind of square matrix where the entries either below or above the main diagonal are zero. There are many different kinds of matrix decompositions and each of these finds use among a particular class of problems. In particular, *LU* decomposition gives an algorithm to decompose any invertible matrix A into a normed lower triangular matrix L and an upper triangular matrix U so that $A = LU$. Therefore, given a matrix equation

$$Ax = LUx = b$$

we want to solve the equation for a given A and b. This can be done in two logical steps:

- First, we solve the equation $Ly = b$ for y
- Second, we solve the equation $Ux = y$ for x

where L and U are lower and upper triangular matrices of the same size, respectively. This means that L has only zeros above the diagonal and U has only zeros below the diagonal. For a 3×3 matrix, this becomes:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Note that in both cases we have triangular matrices, lower and upper, which can be solved directly using forward and backward substitution without using the Gaussian elimination process. However, we need this process or its equivalent to compute the *LU* decomposition itself. Thus the *LU* decomposition is computationally efficient only when we have to solve a matrix equation multiple times for different *b*. It is faster in this case to do an *LU* decomposition of the matrix *A* once and then solve the triangular matrices for the different *b*, than to use Gaussian elimination each time. The Doolittle method sets the lower diagonal elements of *L* to 1 and the Crout method sets the upper diagonal elements of *U* to 1. This chapter uses the Crout method to perform the LU decomposition. The following code illustrates the LU decomposition process in more detail.

```
// LU decomposition using the Crout algorithm with pivoting
public static double LUCrout(RMatrix A, RVector b)
{
  LUDecompose(A);
  return LUSubstitute(A, b);
}

private static void LUDecompose(RMatrix matrix)
{
  int nRows = matrix.GetnRows;
  for (int i = 0; i < nRows; i++)
  {
    for (int j = 0; j < nRows; j++)
    {
      double w = matrix[i, j];
      for (int k = 0; k < Math.Min(i, j); k++)
      {
        w -= matrix[i, k] * matrix[k, j];
      }
      if (j > i)
      {
        double s = matrix[i, i];
        if (Math.Abs(w) < epsilon)
          throw new Exception("Diagonal element is too small!");
        w /= s;
      }
      matrix[i, j] = w;
    }
  }
}

private static double LUSubstitute(RMatrix matrix, RVector vec)
{
  int size = vec.GetVectorSize;
  double det = 1.0;
  for (int i = 0; i < size; i++)
  {
    double w = vec[i];
    for (int j = 0; j < i; j++)
    {
      w -= matrix[i, j] * vec[j];
    }
    double p = matrix[i, i];
    if (Math.Abs(w) < epsilon)
      throw new Exception("Diagonal element is too small!");
    w /= p;
    vec[i] = w;
    det *= matrix[i, i];
  }
  for (int i = size - 1; i >= 0; i--)
  {
    double s = vec[i];
    for (int j = i + 1; j < size; j++)
    {
      s -= matrix[i, j] * vec[j];
    }
    vec[i] = s;
  }
  return det;
}

static void TestLU()
{
  RMatrix A = new RMatrix(new double[3,3]
      {{2,4,-6},{6,-4,2},{4,2,6}});
  RVector b = new RVector(new double[3] {4,4,8});
  RMatrix Anew = A.Clone();
  RMatrix Bnew = A.Clone();
  double d = LUCrout(A,b);
  RMatrix inv = LUInverse(Anew);
  Console.WriteLine("\nInverse of A = \n {0}", (inv));
  Console.WriteLine("\nSolution of the equations Ax = b is x={0}",b);
  Console.WriteLine("\nDeterminant of A = {0}", d);
  Console.WriteLine("\nTest Inverse: A*Inverse = \n {0}",Bnew*inv);
}

static void Main(string[] args)
{
  TestLU(); Console.ReadLine();
}

OUTPUT:
 Inverse of A =
 (0.0833333333333333, 0.107142857142857, 0.0476190476190476
 0.0833333333333333, -0.107142857142857, 0.119047619047619
 -0.0833333333333333, -0.0357142857142857, 0.0952380952380952)

Solution of the equations Ax = b is
x = (1.14285714285714, 0.857142857142857, 0.285714285714286)

Determinant of A = -336

Test Inverse: A*Inverse =
 (1, 0, 0
 -1.38777878078145E-16, 1, 0
 -5.55111512312578E-17, 0, 1)
```

Note that the matrices L and U can be used to compute the determinant of the matrix A very quickly, because $\det(A) = \det(L)\det(U)$ and the determinant of a triangular matrix is simply the product of its diagonal entries. In particular, if L is a unit triangular matrix, then

$$\det(A) = \det LU = \det(L)\det(U) = \det L = \prod_{i=1}^{n} L_{ii}$$

The matrices L and U can therefore be used to calculate the inverse matrix of A by:

$$A^{-1} = U^{-1}L^{-1}$$

Computer implementations that invert matrices, as shown below, sometimes use this approach.

```
//LU Matrix Inverse
public static RMatrix LUInverse(RMatrix matrix)
{
    int nRows = matrix.GetnRows;
    RMatrix u = matrix.IdentityMatrix();
    LUDecompose(matrix);
    RVector uv = new RVector(nRows);
    for (int i = 0; i < nRows; i++)
    {
        uv = u.GetRowVector(i);
        LUSubstitute(matrix, uv);
        u.ReplaceRow(uv, i);
    }
    RMatrix inverse = u.GetTranspose();
    return inverse;
}
```

8.5 Iteration Methods

An iterative method attempts to solve a problem, such as an equation or system of equations, by finding successive approximations to the solution starting from an initial guess. This approach is in contrast to direct methods, which attempt to solve a problem by a finite sequence of operations, and, in the absence of rounding errors, could deliver an exact solution.

8.5.1 Gauss-Jacobi Iteration

The Jacobi method is an algorithm in linear algebra for determining the solutions of a system of linear equations with largest absolute values in each row and column dominated by the diagonal element. Each diagonal element is solved for, and an approximate value plugged in. The process is then iterated until it converges. This algorithm is a stripped-down version of the Jacobi transformation method of matrix diagonalization.

We seek the solution to a set of linear equations, written in matrix form as

$$Ax = b$$

Let $A = D + (L + U)$, where D, L, and U represent the diagonal, lower triangular, and upper triangular parts of the coefficient matrix A. Then the equation above can be rephrased as:

$$Dx + (L + U)x = b$$

If $a_{ii} \neq 0$ for each i then we can solve directly for x using the equation.

$$x = D^{-1}\left[b - (L + U)x\right]$$

By iterative rule, the definition of the Jacobi method can be expressed as:

$$x^{(k+1)} = D^{-1}\left[b - (L + U)x^{(k)}\right]$$

where k is the iteration count. Often an element-based approach is used so that:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\right) \quad \text{where} \quad i = 1, 2, \ldots, n$$

Note that the computation of $x_i^{(k+1)}$ requires an element in $x^{(k)}$ except itself. The method will always converge if the matrix A is strictly or irreducibly diagonally dominant. Strict row diagonal dominance means that for each row, the absolute value of the diagonal term is greater than the sum of absolute values of other terms

$$|a_{ii}| > \sum_{i \neq j} |a_{ij}|$$

The Jacobi method sometimes converges even if this condition is not satisfied. It is necessary, however, that the diagonal terms in the matrix are greater in magnitude than the other terms. Thus, two parameters can be used to control the number of iterations: one is a tolerance factor imposed by this convergence condition and the other is the maximum number of iterations imposed by the user. Failure to properly set any of these factors can result in an infinite or nearly infinite loop or the program may terminate early thus yielding a false or erroneous result.

```
// Gauss-Jacobi method
public static RVector GaussJacobi(RMatrix A, RVector b, int
    MaxIterations, double tolerance)
{
  int bSize = b.GetVectorSize;
  RVector x = new RVector(bSize);
  for (int nIteration=0; nIteration<MaxIterations;nIteration++)
  {
    RVector xOld = x.Clone();
    for (int i = 0; i < bSize; i++)
    {
      double entry = b[i];
      double diagonal = A[i, i];
      if (Math.Abs(diagonal) < epsilon)
        throw new Exception("Diagonal element is too small!");
      for (int j = 0; j < bSize; j++)
      {
        if (j != i)
        {
          entry -= A[i, j] * xOld[j];
        }
      }
      x[i] = entry / diagonal;
    }
    RVector dx = x - xOld;
    if (dx.GetNorm() < tolerance)
    {
      return x;
    }
  }
  return x;
}

static void TestGaussJacobiIteration()
{
    RMatrix A =
        new RMatrix(new double[3, 3] {{4,0,1},{0,3,2},{1,2,4}});
    RVector b = new RVector(new double[3] {2,1,3} );
    RMatrix A1 = A.Clone();
    RVector b1 = b.Clone();

    RVector x = GaussJacobi(A, b, 10, 1.0e-4);
    Console.WriteLine("\nSolution from the Gauss-Jacobi iteration:");
    Console.WriteLine(" x[0] = {0}", x[0]);
    Console.WriteLine(" x[1] = {0}", x[1]);
    Console.WriteLine(" x[2] = {0}", x[2]);
}

static void Main(string[] args)
{
  TestGaussJacobiIteration();
  Console.ReadLine();
}

OUTPUT: Solution from the Gauss-Jacobi iteration:
 x[0] = 0.310397736820174
 x[1] = -0.17227270181287
 x[2] = 0.751248669722443
```

8.5.2 Gauss-Seidel Iteration

The Gauss-Seidel method is a technique used to solve a linear system of equations which is actually an improved version of the Jacobi method. The goal of the Gauss-

Seidel method is to find a solution to a set of linear equations, expressed in matrix terms as

$$Ax = b$$

The Gauss-Seidel iteration is expressed by

$$x^{(k+1)} = (D+L)^{-1}\left(b - Ux^{(k)}\right)$$

where $A = D + L + U$. The matrices D, L, and U represent the diagonal, strictly lower triangular, and strictly upper triangular parts of the coefficient matrix A and k is the iteration count. This matrix expression is mainly used to analyze the method. This time, however, we cannot overwrite $x_i^{(k)}$ with $x_i^{(k+1)}$, as that value will be needed by the rest of the computation. This is the most meaningful difference between the Jacobi and Gauss-Seidel methods. Instead, the minimum amount of storage is two vectors of size n.

When implementing the Gauss-Seidel method, an explicit entry-by-entry approach is used:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right) \quad \text{where} \quad i = 1, 2, \ldots, n$$

Note that the computation of $x_i^{(k+1)}$ uses only those elements of $x^{(k+1)}$ that have already been computed and only those elements of $x^{(k)}$ that have yet to be advanced to iteration $k+1$. This means that no additional storage is required, and the computation can be done in place so that $x^{(k+1)}$ replaces $x^{(k)}$. While this might seem like a rather minor concern, for large systems it is unlikely that every iteration can be stored. Thus, unlike the Jacobi method, this process does not have to do any vector copying should one want to use only one storage vector. The iteration is generally continued until the changes made by an iteration are below some tolerance factor.

Gauss-Seidel is guaranteed to converge for an arbitrary symmetric positive definite matrix A. If A is unsymmetric, it will always converge if the matrix A is strictly or irreducibly diagonally dominant. Strict row diagonal dominance means that for each row, the absolute value of the diagonal term is greater than the sum of absolute values of other terms:

$$|a_{ii}| > \sum_{i \neq j} |a_{ij}|.$$

The Gauss-Seidel method sometimes converges even if this condition is not satisfied. It is necessary, however, that the diagonal terms in the matrix are greater in magnitude than the other terms. The following code illustrates the Gauss-Seidel iteration process in more detail.

```
// Gauss-Seidel method
public static RVector GaussSeidel(RMatrix A, RVector b, int
    MaxIterations, double tolerance)
{
  int size = b.GetVectorSize;
  RVector x = new RVector(size);
  for (int nIteration = 0; nIteration < MaxIterations; nIteration++)
  {
    RVector xOld = x.Clone();
    for (int i = 0; i < size; i++)
    {
      double entry = b[i];
      double diagonal = A[i, i];
      if (Math.Abs(diagonal) < epsilon)
        throw new Exception("Diagonal element is too small!");
      for (int j = 0; j < i; j++)
      {
        entry -= A[i, j] * x[j];
      }
      for (int j = i + 1; j < size; j++)
      {
        entry -= A[i, j] * xOld[j];
      }
        x[i] = entry / diagonal;
      }
      RVector dx = x - xOld;
      if (dx.GetNorm() < tolerance)
      { return x; }
    }
  }
  return x;
}

static void TestGaussSeidelIteration()
{
   RMatrix A = new RMatrix(new double[3, 3]
        {{4,0,1},{0,3,2},{1,2,4}});
   RVector b = new RVector(new double[3] {2,1,3});
   RMatrix A1 = A.Clone();
   RVector b1 = b.Clone();

   RVector x1 = GaussSeidel(A1, b1, 10, 1.0e-4);
   Console.WriteLine("\n Solution from the Gauss-Seidel iteration:");
   Console.WriteLine(" x1[0] = {0}", x1[0]);
   Console.WriteLine(" x1[1] = {0}", x1[1]);
   Console.WriteLine(" x1[2] = {0}", x1[2]);
}

static void Main(string[] args)
{
  TestGaussSeidelIteration();
  Console.ReadLine();
}

OUTPUT: Solution from the Gauss-Seidel iteration:
 x1[0] = 0.310390073145789
 x1[1] = -0.172293138277897
 x1[2] = 0.758549050852501
```

8.6 Eigenvalues and Jacobi's Algorithm

In general, an n-dimensional vector x is called an eigenvector of a square $n \times n$ matrix A if and only if it satisfies the following linear equation

$$Ax = \lambda x$$

where λ is a scalar referred to as the eigenvalue corresponding to x. The equation above can be rewritten as

$$Ax - \lambda Ix = 0$$

where I is the identity matrix. This equation can be rearranged to

$$(A - \lambda I)x = 0$$

If there exists an inverse $(A - \lambda I)^{-1}$ then both sides can be left multiplied by the inverse to obtain the trivial solution: x = 0. Thus we require there to be no inverse by assuming from linear algebra that the determinant equals zero:

$$\det(A - \lambda I) = 0$$

The determinant requirement is called the characteristic equation of A, and the left-hand side is called the characteristic polynomial. The roots λ_i for $i = 0, 1, 2, \ldots$ are called the eigenvalues of the matrix A and the solution x where $(A - \lambda_i I)x = 0$ are known as the eigenvectors of matrix A.

From the matrix equations shown above, the calculation of both eigenvalues and eigenvectors at first appears to be a conceptually simple process. However, in practice the actual calculation of eigenvalues and eigenvectors is a fairly complicated business, especially for very large matrices, and remains a very active area of research. There are many numerical methods available for calculating eigenvalues but they often come with a substantial number of restrictions and limitations as to the size and kind of matrices that a particular method can be successfully applied [22].

The Jacobi eigenvalue algorithm is a numerical procedure for calculating all the eigenvalues and eigenvectors of a real symmetric matrix. Moreover, it is a reliable method that produces uniformly accurate results. For matrices of order up to 10×10, the algorithm is competitive with more sophisticated ones. If speed is not a major consideration, it is quite acceptable for matrices up to order 20×20. A solution is guaranteed for all real symmetric matrices when Jacobi's method is used. This limitation is not severe since many practical problems of applied mathematics and engineering involve symmetric matrices. From a theoretical viewpoint, the method also embodies techniques that are found in more sophisticated algorithms. For pedagogical and practical purposes, it is therefore worthwhile to investigate the details of Jacobi's method.

The idea behind Jacobi's eigenvalue algorithm is conceptually simple. From linear algebra, it is an established fact that all eigenvalues of a real symmetric matrix A are real [22]. As a result, there exists a real orthogonal matrix S such that $S^{-1}AS$ is a diagonal matrix D. As D and A are similar matrices, the diagonal elements of D are therefore also the eigenvalues of A. However, the computation of matrix S is not a simple task. It is obtained by a series of orthogonal transformations $S_1S_2,\ldots,S_n$ as discussed below.

Let $|a_{ij}|$ be the largest element among the off-diagonal elements of A. We construct an orthogonal matrix S_1 whose elements are defined as

$$s_{ij}=-\sin\theta,\quad s_{ji}=\sin\theta,\quad s_{ii}=\cos\theta,\quad s_{jj}=\cos\theta$$

All other off-diagonal elements are zero and all other diagonal elements are unity. Therefore, S_1 is of the form

$$S_1=\begin{bmatrix}1\;0\cdots & 0 & \cdots & 0 & \cdots\;0\\ 0\;1\cdots & 0 & \cdots & 0 & \cdots\;0\\ \vdots\;\vdots & \vdots & & \vdots & \vdots\\ 0\;0\cdots & \cos\theta & \cdots & -\sin\theta & \cdots\;0\\ \vdots\;\vdots & \vdots & & \vdots & \vdots\\ 0\;0\cdots & \sin\theta & \cdots & \cos\theta & \cdots\;0\\ \vdots\;\vdots & \vdots & & \vdots & \vdots\\ 0\;0\cdots & 0 & \cdots & 0 & \cdots\;1\end{bmatrix}$$

where $\cos\theta$, $-\sin\theta$, $\sin\theta$, and $\cos\theta$ are at positions (i,i), (i,j), (j,i), and (j,j) respectively.

Let $A_1=\begin{bmatrix}a_{ii} & a_{ij}\\ a_{ji} & a_{jj}\end{bmatrix}$ be a sub-matrix of A formed by the elements a_{ii},a_{ij},a_{ji}, and a_{jj}. To reduce A_1 to a diagonal matrix, an orthogonal transformation is applied which is defined as $\overline{S}_1=\begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}$, where θ is an unknown quantity and it will be selected in such a way that A_1 becomes diagonal. Now,

$$\begin{aligned}\overline{S}_1^{-1}A_1\overline{S}_1&=\begin{bmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{bmatrix}\begin{bmatrix}a_{ii} & a_{ij}\\ a_{ji} & a_{jj}\end{bmatrix}\begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}\\ &=\begin{bmatrix}a_{ii}\cos^2\theta+a_{ij}\sin 2\theta+a_{jj}\sin^2\theta & (a_{jj}-a_{ii})\sin\theta\cos\theta+a_{ij}\cos 2\theta\\ (a_{jj}-a_{ii})\sin\theta\cos\theta+a_{ij}\cos 2\theta & a_{ii}\sin^2\theta-a_{ij}\sin 2\theta+a_{jj}\cos^2\theta\end{bmatrix}\end{aligned}$$

This matrix becomes a diagonal matrix if $(a_{jj}-a_{ii})\sin\theta\cos\theta+a_{ij}\cos 2\theta=0$. That is, if

$$\tan 2\theta=\frac{2a_{ij}}{a_{ii}-a_{jj}}$$

The value of θ can then be obtained from the following relation

$$\theta=\frac{1}{2}\tan^{-1}\left(\frac{2a_{ij}}{a_{ii}-a_{jj}}\right)$$

This expression gives four values of θ, but to get the smallest rotation, θ should lie in the region given by $-\pi/4 \le \theta \le \pi/4$. This equation is valid for all i, j such that $a_{ii} \neq a_{jj}$. If $a_{ii} = a_{jj}$ then

$$\theta = \begin{cases} \dfrac{\pi}{4} & \text{if} \quad a_{ij} > 0 \\ -\dfrac{\pi}{4} & \text{if} \quad a_{ij} < 0 \end{cases}$$

Therefore, the off-diagonal elements s_{ij} and s_{ji} of $\overline{S}_1^{-1} A_1 \overline{S}_1$ vanish and the diagonal elements are modified. The first diagonal matrix is obtained by computing $D_1 = S_1^{-1} A_1 S_1$. In the next step, the largest off-diagonal element is selected from the matrix D_1 and the above process is repeated to generate another orthogonal matrix S_2 to compute D_2. That is,

$$D_2 = S_2^{-1} D_1 S_2 = S_2^{-1}(S_1^{-1} A S_1) S_2 = (S_1 S_2)^{-1} A (S_1 S_2)$$

In this way, a series of two-dimensional rotations are performed. At the end of a k transformations the matrix D_k is obtained as

$$D_k = (S_1 S_2 \cdots S_k)^{-1} A (S_1 S_2 \cdots S_k) = S^{-1} A S$$

where $S = S_1 S_2 \cdots S_k$.

As $k \to \infty$, D_k tends to a diagonal matrix. The diagonal elements of D_k are the eigenvalues and the columns of S are the corresponding eigenvectors.

Unfortunately, like all eigenvalue algorithms, the Jacobi algorithm also has a drawback. The elements which are transfered to zero during the diagonalization process may not necessarily remain zero during subsequent rotations. Therefore the value of θ must be periodically verified for its accuracy by checking whether $|\sin^2\theta + \cos^2\theta - 1|$ remains sufficiently small.

Below is an implementation of the Jacobi eigenvalue algorithm in C#. A 4×4 symmetric matrix was used as input data for testing. The results are subsequently displayed.

```
public static void JacobiEigenValVec(double[,] a, int maxsize, int n,
     double epsilon, out double[,] eigenval, out double[,] eigenvec)
{
    int i, j, p, q, flag;
    double[,] d = new double[maxsize, maxsize];
    double[,] s = new double[maxsize, maxsize];
    double[,] s1 = new double[maxsize, maxsize];
    double[,] s1t = new double[maxsize, maxsize];
    double[,] temp = new double[maxsize, maxsize];
    double theta, max;

    //Initialization of matrix d and s
    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= n; j++)
        {
            d[i, j] = a[i, j];
            s[i, j] = 0.0;
        }
    }
    for (i = 1; i <= n; i++) s[i, i] = 1.0;

    do
    {
        flag = 0;
        //Find largest off-diagonal element
        i = 1;
        j = 2;
        max = Math.Abs(d[1, 2]);
        for (p = 1; p <= n; p++)
        {
            for (q = 1; q <= n; q++)
            {
                if (p != q) //off diagonal element
                {
                    if (max < Math.Abs(d[p, q]))
                    {
                        max = Math.Abs(d[p, q]);
                        i = p;
                        j = q;
                    }
                }
            }
        }

        if (d[i, i] == d[j, j])
        {
            if (d[i, j] > 0) theta = Math.PI / 4.0;
            else theta = -Math.PI / 4.0;
        }
        else
        {
            theta = 0.5*Math.Atan(2.0*d[i,j]/(d[i,i]-d[j,j]));
        }

        //Construction of the matrix s1 and s1t
        for (p = 1; p <= n; p++)
        {
            for (q = 1; q <= n; q++)
            {
                s1[p, q] = 0.0;
                s1t[p, q] = 0.0;
            }
        }

        for (p = 1; p <= n; p++)
        {
            s1[p, p] = 1.0;
            s1t[p, p] = 1.0;
        }

        s1[i, i] = Math.Cos(theta); s1[j, j] = s1[i, i];
```

```
s1[j, i] = Math.Sin(theta); s1[i, j] = -s1[j, i];
s1t[i, i] = s1[i, i]; s1t[j, j] = s1[j, j];
s1t[i, j] = s1[j, i]; s1t[j, i] = s1[i, j];

//Product of s1t and d
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= n; j++)
    {
        temp[i, j] = 0.0;
        for (p = 1; p <= n; p++)
            temp[i, j] += s1t[i, p] * d[p, j];
    }
}

//Product of temp and s1: d = s1t * d * s1
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= n; j++)
    {
        d[i, j] = 0.0;
        for (p = 1; p <= n; p++)
            d[i, j] += temp[i, p] * s1[p, j];
    }
}

//Product of s and s1: s = s*s1
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= n; j++)
    {
        temp[i, j] = 0.0;
        for (p = 1; p <= n; p++)
            temp[i, j] += s[i, p] * s1[p, j];
    }
}

for (i = 1; i <= n; i++)
{
    for (j = 1; j <= n; j++)
    {
        s[i, j] = temp[i, j];
    }
}

//check to see if d is a diagonal matrix
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= n; j++)
    {
        if (i != j)
            if (Math.Abs(d[i, j]) > epsilon)
                flag = 1;
    }
}
```

```
    } while (flag == 1);
    //copy results to output matrices
    eigenval = d;
    eigenvec = s;
}

public static void TestingJacobiEigenValVec()
{
  int nCol = 4;              //size of input 4x4 matrix
  int maxMatrixSize = 10;    //max matrix size
  double Epsilon = 1.0e-04; //tolerance check if close enough to 0.
  double[,] A = new double[maxMatrixSize,maxMatrixSize];
  double[,] Eigenvalues = new double[maxMatrixSize,maxMatrixSize];
  double[,] Eigenvectors = new double[maxMatrixSize,maxMatrixSize];

  A[1, 1] = 1; A[1, 2] = 2; A[1, 3] = 3; A[1, 4] = 4;
  A[2, 1] = 2; A[2, 2] =-3; A[2, 3] = 3; A[2, 4] = 4;
  A[3, 1] = 3; A[3, 2] = 3; A[3, 3] = 4; A[3, 4] = 5;
  A[4, 1] = 4; A[4, 2] = 4; A[4, 3] = 5; A[4, 4] = 0;

  Console.WriteLine("\n\nTesting Jacobi's Method For Finding
      Eigenvalues and Eigenvectors\n");
  Console.WriteLine("The input matrix is given by\n");
  for (int i = 1; i <= nCol; i++)
  {
    for (int j = 1; j <= nCol; j++)
    {
      if (j != nCol)
        Console.Write(A[i, j].ToString("0.000000") + "\t");
      else
        Console.WriteLine(A[i, j].ToString("0.000000"));
    }
  }

  //Calculate eigenvalues and eigenvectors using Jacobi method
  JacobiEigenValVec(A, maxMatrixSize, nCol, Epsilon, out Eigenvalues,
       out Eigenvectors);

  //Output results
  Console.WriteLine("\nThe eigenvalues are:\n");
  for (int i = 1; i <= nCol; i++)
  { Console.WriteLine(Eigenvalues[i, i].ToString("0.00000")); }

  Console.WriteLine("\nThe corresponding eigenvectors are\n");
  for (int j = 1; j <= nCol; j++)
  {
    for (int i = 1; i <= nCol; i++)
    {
      if (i != nCol)
       Console.Write(Eigenvectors[i, j].ToString("0.00000")+"\t\t");
      else
       Console.WriteLine(Eigenvectors[i, j].ToString("0.00000"));
    }
  }
}
```

```
static void Main(string[] args)
{
  TestingJacobiEigenValVec();
  Console.ReadLine();
}

OUTPUT:
Testing Jacobi’s Method For Finding Eigenvalues and Eigenvectors

The input matrix is given by

1.000000        2.000000        3.000000        4.000000
2.000000       -3.000000        3.000000        4.000000
3.000000        3.000000        4.000000        5.000000
4.000000        4.000000        5.000000        0.000000

The eigenvalues are:

-0.73369
-5.88321
11.78254
-3.16564

The corresponding eigenvectors are

 0.74263        0.04635        -0.65234         0.14421
 0.13467        0.74460         0.06235        -0.65081
 0.43846        0.33395         0.64097         0.53422
-0.48797        0.57611        -0.39965         0.51987
```

9

Nonlinear Equations

9.1 Introduction

Nonlinear equations are of particular interest in mathematics, engineering and the natural sciences because many physical phenomena are inherently nonlinear in nature. Predicting future weather events is an excellent example of a nonlinear system. Here, simple changes in one part of the system can produce complex effects throughout other parts of the system. Other examples of nonlinear systems include the NavierStokes differential equations in fluid dynamics, the LotkaVolterra equations in biology, and the BlackScholes partial differential equation in finance. Nonlinear systems can also give rise to some other interesting physical phenomena such as chaos. Unfortunately, most nonlinear equations are so difficult to solve that they cannot be solved analytically but instead can only be solved by numerical approximations [22].

In general, nonlinear systems can be classified into three broad categories:

- Indeterministic systems where the behavior of a system cannot be predicted
- Multistabilistic systems where solutions alternate between two or more exclusive states
- Aperiodic oscillatory systems, also known as chaotic systems, where solutions do not repeat values after some unspecified period

In mathematics, a linear function (or map) $f(x)$ is one which satisfies the following properties:

- Additivity, $f(x+y) = f(x) + f(y)$
- Homogeneity, $f(\alpha x) = \alpha f(x)$

where α is just a scalar constant. A function $f(x)$ that does not satisfy the criteria for linear functions as listed above is said to be *nonlinear*. The most common types of nonlinear equations may include polynomial, transcendental, exponential, logarithmic, trigonometric, hyperbolic equations or some combination thereof.

Many nonlinear root finding problems involve finding one or more values of x that satisfy one of the following three basic forms of equations:

- $f(x) = 0$
- $g(x) = C \rightarrow f(x) = g(x) - C = 0$
- $g(x) = h(x) \rightarrow f(x) = g(x) - h(x) = 0$

As difficult and intimidating as some nonlinear equations may appear, it sometimes very helpful to think of them in graphical terms before attempting to solve them. For example, solutions to equations of the form $f(x) = 0$ can be seen as places where the graph of $f(x)$ crosses or touches the x axis. Similarly, solutions to equations of the form $f(x) = g(x)$ can be seen as places where the graphs of $f(x)$ and $g(x)$ intersect. Fortunately, most nonlinear equations can be solved through repeated iteration of some approximated expression until an answer is achieved to the desired degree of accuracy or tolerance.

The most popular nonlinear root finding algorithms use bracketing methods to arrive at approximate solutions and so it seems logical to start by briefly describing what they are and how they work. A root is said to be *bracketed* in the interval given by $a \leq x \leq b$ if $f(a)$ and $f(b)$ have opposite signs so that $f(a)f(b) < 0$. In that case and if the function is continuous, then at least one root must lie in that interval. If the function is discontinuous but bounded, then instead of a root, there might be a step discontinuity that crosses zero. In such cases, the behavior is indistinguishable from the case of a continuous function whose zero crossing occurs somewhere in that interval. Only for functions with singularities, such as in the case for

$$f(x) = \frac{1}{x - c}$$

is there a possibility that a bracketed root is not really there. With all these basic concepts in mind, let us start this chapter which covers the topic of nonlinear equations.

9.2 Linear Incremental Method

The linear incremental method is perhaps the simplest incremental procedure that does not use any iteration. After making an initial guess of the value of a suspected root and evaluating the function at that point, another point is selected slightly higher or lower than the initial point and the function is evaluated again at that new point. If there is a change in the sign of the evaluated functions, then there must be a root between the last two points and the result is then calculated by doing a linear interpolation of the points to extract the approximate value of the root. The success of this method depends on guessing an initial value that is close enough to the actual root *and* choosing a step size Δx that is small enough to avoid skipping over roots. The outline of the linear incremental method algorithm is shown below.

- Pick a starting point x_n and a step size Δx such that the next point can be found by simply adding Δx to x_n as shown here: $x_{n+1} = x_n + \Delta x$. Use a positive Δx if you want to search in the positive $+x$ direction, and a negative Δx if you want to search in the negative $-x$ direction.

- Using $x_{n+1} = x_n + \Delta x$, calculate $f(x_n)$ and $f(x_{n+1}) = f(x_n + \Delta x)$.

- If $f(x_n)f(x_{n+1}) < 0$, indicating that the sign of $f(x)$ has changed while stepping from x_n to x_{n+1}, then there must be a root of $f(x)$ in the interval (x_n, x_{n+1}). The solution x is then approximated by a linear interpolation

$$x = x_{n+1} - \frac{f(x_{n+1})\Delta x}{f(x_{n+1}) - f(x_{n+1} - \Delta x)}$$

- If $f(x_n)f(x_{n+1}) > 0$, then the sign of $f(x_n)$ did not change in stepping from x_n to x_{n+1}. As a result, calculate the location of the next point: $x_{n+1} = x_n + \Delta x$ and repeat the process.

Unfortunately, the incremental search method has some important drawbacks to keep in mind:

- It only finds real-valued roots of $f(x)$. It cannot find complex roots of polynomials.

- It only finds roots where $f(x)$ crosses the x axis. It cannot find roots where $f(x)$ is tangent to the x axis.

- It may be fooled by singularities in $f(x)$, such as in the tangent and cotangent functions.

- If the step size Δx is too large, it may miss closely-spaced roots by skipping over them.

In spite of these limitations, the incremental search method is a good method to get started in learning to handle problems involving nonlinear equations. The code below shows how one might implement the incremental search method for solving nonlinear equations in C#.

```
//Used in all nonlinear root finding examples
public delegate double Function(double x);

//Used in all nonlinear root finding examples
static double F(double x)
{
   return x*x*x - 5.0*x + 3.0;
}

public static double LinearIncrementalSearch(Function f, double
     xstart, double deltaX, int nMaxInc)
{
   double x = xstart;
   double fstart = f(xstart);
   double fx = fstart;
   double fProd = 0;
   for (int i = 0; i < nMaxInc; i++)
   {
      x = xstart + i * deltaX;
      fx = f(x);
      fProd = fstart * fx;
      if (fProd < 0)
         break;
   }
   if (fProd > 0)
      throw new Exception("Solution not found!");
   else
   {
      return x = x - (deltaX*fx)/(fx - f(x-deltaX));
   }
}

static void Main(string[] args)
{
  Console.WriteLine("\nTesting Testing Linear Incremental Method\n");
  double deltaX = 0.01;
  int n = 500;
  double x = -4.0;
  for (int i = 1; i <= 3; i++)
  {
    x = LinearIncrementalSearch(F, x, deltaX, n);
    Console.WriteLine("\nSolution " + i.ToString() + " = " + x.
        ToString());
    Console.WriteLine("Solution confirmation: f(x) = " + F(x).
        ToString());
  }
  Console.ReadLine();
}

OUTPUT:
Testing Linear Incremental Method
Solution 1 = -2.49085929901792
Solution confirmation: f(x) = 5.87591228651263E-05
Solution 2 = 0.656630398082894
Solution confirmation: f(x) = -3.69431091167272E-05
Solution 3 = 1.83422345770804
Solution confirmation: f(x) = -0.000100472251910233
```

9.3 Bisection Method

Suppose $f(x)$ is a continuous function and that we want to solve the equation $f(x) = 0$. The bisection method starts by selecting two points a and b such that $f(a)$ and $f(b)$ have opposite signs so that $f(a)f(b) < 0$. By the intermediate value theorem, if

$f(a)$ and $f(b)$ have opposite signs then $f(x)$ must have at least one root in the interval $[a,b]$. If we then divide the interval $[a,b]$ in half, we end up with two possibilities. Either the interval $[a,(a+b)/2]$ has opposite signs: $f(a)f((a+b)/2) < 0$, or the interval $[(a+b)/2,b]$ has opposite signs: $f((a+b)/2)f(b) < 0$. If both intervals have opposite signs, then the initial interval chosen was too big and we need to go back to the beginning of this process and select a smaller interval closer to the location where there might be a root. The bisection algorithm is then applied recursively to each new sub-interval where the sign change occurs until a solution is found up to the desired accuracy or, as it is sometimes called, a tolerance factor is reached.

If we let a_n and b_n be the endpoints at the n-th iteration such that $a_1 = a$ and $b_1 = b$ and let r_n be the n-th approximate solution then the number of iterations n that is required to obtain an error smaller than some tolerance factor ε can be estimated by noting that

$$b_n - a_n = \frac{b-a}{2^{(n-1)}}$$

and that r_n is defined by

$$r_n = \frac{a_n + b_n}{2}$$

In order for the error to be $< \varepsilon$,

$$|r_n - r| \leq \frac{(b_n - a_n)}{2} = \frac{b-a}{2^n} < \varepsilon.$$

Taking the natural logarithm of both sides then gives

$$-n \ln 2 < \ln \varepsilon - \ln(b-a)$$

from which we can solve for n

$$n > \frac{\ln(b-a) - \ln \varepsilon}{\ln 2}$$

The code below shows how one might implement the bisection method for solving nonlinear equations in C#.

```
public static double Bisection(Function f, double a, double b, double
     epsilon)
{
   double x1 = a; double x2 = b;
   double fb = f(b);
   while (Math.Abs(x2-x1) > epsilon)
   {
      double midpt = 0.5*(x1+x2);
      if (fb*f(midpt) > 0)
         x2 = midpt;
      else
         x1 = midpt;
   }
   return x2-(x2-x1)*f(x2)/(f(x2)-f(x1));
}
```

```
static void Main(string[] args)
{
  Console.WriteLine("\n\n Testing Bisection Method\n");
  x = Bisection(F, 1.0, 2.0, 0.0001);
  Console.WriteLine("Solution from bisection method:"+x.ToString());
  Console.WriteLine("Solution confirmation:f(x) ="+F(x).ToString());
  Console.ReadLine();
}

OUTPUT:
Testing Bisection Method
Solution from the bisection method: 1.83422345770804
Solution confirmation: f(x) = -3.74249431445151E-09
```

9.4 The Secant Method

To improve the slow convergence of the bisection 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 the new reference point. The next iteration starts from evaluating the function at the new reference point which is then used to form another line. This process is repeated until the root is found. Note that the secant method retains only the most recent estimate, and so the root does not necessarily remain bracketed. Since the secant method does not always bracket the root, the algorithm may not converge for functions that are not sufficiently smooth. The recurrence relation for the secant method can be derived as follows.

Given x_{n-1} and x_n, we construct the line through the points $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$. Note that this line is a secant or chord of the graph of the function $f(x)$. In point-slope form, the secant line can be defined as

$$y - f(x_n) = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}(x - x_n)$$

We now choose x_{n+1} to be the root of this line. Then the equation above can be written as

$$f(x_n) + \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}(x_{n+1} - x_n) = 0.$$

Solving this last equation for x_{n+1} then gives the recurrence relation for the secant method

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

The code below shows how one might implement the secant method for solving nonlinear equations in C#.

```
public static double SecantMethod(Function f, double a, double b,
    double epsilon)
{
  double x1 = a;
  double x2 = b;
  double fb = f(b);
  while (Math.Abs(f(x2)) > epsilon)
  {
    double mpoint = x2-(x2-x1)*fb/(fb-f(x1));
    x1 = x2;
    x2 = mpoint;
    fb = f(x2);
  }
  return x2;
}

static void Main(string[] args)
{
  Console.WriteLine("\n\n Testing Secant Method\n");
  x = SecantMethod(F, 1.0, 1.5, 0.0001);
  Console.WriteLine("Solution from the secant method:"+x.ToString());
  Console.WriteLine("Solution confirmation:f(x) = "+F(x).ToString());
  Console.ReadLine();
}

OUTPUT:
Testing Secant Method
Solution from the secant method: 1.83424298458486
Solution confirmation: f(x) = -1.01728863288741E-06
```

9.5 False Positioning Method

Similar to the secant method, the false position method also uses a straight line to approximate the function $f(x)$ in the local region of interest. The only difference between these two methods is that the secant method keeps the most recent two estimates, while the false position method retains the most recent estimate and the next recent one which has an opposite sign in the function value.

Like the bisection method, the false position method starts with two points a_0 and b_0 such that $f(a_0)$ and $f(b_0)$ are of opposite signs, which implies by the intermediate value theorem that the function $f(x)$ has a root in the interval $[a_0, b_0]$. The method proceeds by producing a sequence of shrinking intervals $[a_n, b_n]$ that all contain a root of $f(x)$. The recurrence relation for the false positioning method can therefore be calculated as follows. Given a_n and b_n, we construct the line through the points $(a_n, f(a_n))$ and $(b_n, f(b_n))$. Note that this line is a secant or chord of the graph of the function $f(x)$. In point-slope form, it can be defined as

$$y - f(b_n) = \frac{f(b_n) - f(a_n)}{b_n - a_n}(x - b_n)$$

We now choose c_n to be the root of this line so that $x = c_n$ when $y = 0$ and so the equation above reduces to

$$f(b_n) + \frac{f(b_n) - f(a_n)}{b_n - a_n}(c_n - b_n) = 0$$

Solving this equation for c_n gives the required recurrence relation

$$c_n = \frac{f(b_n)a_n - f(a_n)b_n}{f(b_n) - f(a_n)}$$

Note that c_n is the root of the secant line through $(a_n, f(a_n))$ and $(b_n, f(b_n))$. If $f(a_n)$ and $f(c_n)$ have the same sign, then we set $a_{n+1} = c_n$ and $b_{n+1} = b_n$, otherwise we set $a_{n+1} = a_n$ and $b_{n+1} = c_n$. This process is repeated until the root is approximated sufficiently well. The above formula is also used in the secant method, but the secant method always retains the last two computed points, while the false position method retains two points which certainly bracket a root. On the other hand, the only difference between the false positioning method and the bisection method is that the latter uses $c_n = (a_n + b_n)/2$. The code below shows how one might implement the false positioning method for solving nonlinear equations in C#.

```
public static double FalsePositionMethod(Function f, double a, double
     b, double epsilon)
{
   double x1 = a;
   double x2 = b;
   double fb = f(b);
   while (Math.Abs(x2 - x1) > epsilon)
   {
      double xpoint = x2-(x2-x1)*f(x2)/(f(x2)-f(x1));
      if (fb * f(xpoint) > 0)
         x2 = xpoint;
      else
         x1 = xpoint;
      if (Math.Abs(f(xpoint)) < epsilon)
         break;
   }
   return x2-(x2-x1)*f(x2)/(f(x2)-f(x1));
}

static void Main(string[] args)
{
  Console.WriteLine("\n\n Testing False Position Method\n");
  x = FalsePositionMethod(F, 1.0, 2.0, 0.0001);
  Console.WriteLine("Solution from false pos. method:"+x.ToString());
  Console.WriteLine("Solution confirmation: f(x)="+F(x).ToString());
  Console.ReadLine();
}

OUTPUT:
False Position Method
Solution from the false position method: 1.83424212554915
Solution confirmation: f(x) =-5.39264722032584E-06
```

9.6 Fixed Point Iteration

By definition, a fixed point of a function is a point that is mapped to itself by the function. In other words, x is said to be a fixed point of the function $f(x)$ if and only if $f(x) = x$. A fixed point is therefore *not* the same thing as a root of the equation $f(x) = 0$ but rather, it is a solution of the equation $f(x) = x$. Geometrically, the fixed points of a function $f(x)$ are the points of intersection of the curve $y = f(x)$ with the line $y = x$.

Iteration is an important concept in computer science. As the name suggests, iteration is a process that repeats itself until an answer is achieved to the desired degree of accuracy or tolerance.

The fixed point iteration method is therefore a technique for calculating roots by computing fixed points of iterated functions.

This method can be applied towards finding roots of nonlinear functions as follow. First, take your nonlinear equation $f(x) = 0$ and convert it algebraically into the form $x = g(x)$. Then starting with an initial guess of x_0, iterate through the equation

$$x_{i+1} = g(x_i) \quad \text{where} \quad n = 0, 1, 2, \ldots$$

until some convergence criterion is met. Typical convergence criteria can include fixing apriori the maximum total number of iterations that is to be performed or testing the condition $|x_{i+1} - g(x_i)| \leq \varepsilon$ at each iteration until it meets some tolerance limit ε that was also set apriori. The code below shows how one might implement the fixed point iteration method for solving nonlinear equations in C#.

```
static double Ffixpt(double x)
{
   //Setup for test case: x^2 + 2x - 35 = 0
   //Test function = sqrt(35 - 2x)
   return Math.Sqrt(35.0-2.0*x);
}

public static double FixedPointMethod(Function f, double x0, double
     epsilon, int nMaxIter)
{
   double x1 = x0; double x2 = x0;
   double currEpsilon = 0.0;
   for (int i = 0; i < nMaxIter; i++)
   {
      x2 = f(x1);
      currEpsilon = Math.Abs(x1 - x2);
      x1 = x2;
      if (currEpsilon < epsilon)
         break;
   }
   if (currEpsilon > epsilon)
   { throw new Exception("Solution not found!"); }
   return x1;
}
```

```
static void Main(string[] args)
{
 Console.WriteLine("\n\n Testing Fixed Point Method\n");
 double tol = 0.0001;
 n = 10000;
 double x0 = 1.6;
 x = FixedPointMethod(Ffixpt, x0, tol, n);
 Console.WriteLine("Solution from fixed point method:"+x.ToString());
 Console.WriteLine("Expected solution = 5.00");
 Console.ReadLine();
}

OUTPUT:
Testing Fixed Point Method
Solution from the fixed point method: 4.99999172934641
Expected solution = 5.00
```

9.7 Newton-Raphson Method

Newton's method, also known as the Newton-Raphson method, is a root-finding algorithm that uses the first few terms of the Taylor series of a function $f(x)$ in the vicinity of a suspected root. This method can converge remarkably quickly, especially if the iteration begins sufficiently close to the actual root. Unfortunately, when iteration begins far from the actual root, Newton's method can also easily lead one astray with little warning. Successful implementation of this method therefore requires the user to first make a reasonably good initial estimate of the actual root.

Newton's method can be easily derived by using a Taylor series to expand $f(x)$ about the point $x = x_0 + \varepsilon$ as shown below.

$$f(x_0+\varepsilon) = f(x_0) + f'(x_0)\varepsilon + \frac{1}{2}f''(x_0)\varepsilon^2 + \dots$$

Keeping terms only up to first order results in the following approximation.

$$f(x_0+\varepsilon) \approx f(x_0) + f'(x_0)\varepsilon$$

This expression can be used to estimate the amount of offset ε needed to land closer to the root starting from an initial guess x_0. Setting $f(x_0+\varepsilon) = 0$ and solving the equation above for $\varepsilon = \varepsilon_0$ gives

$$\varepsilon_0 = -\frac{f(x_0)}{f'(x_0)}$$

which is the first-order adjustment to the root's position. By letting $x_1 = x_0 + \varepsilon_0$, calculating a new ε_1, and so on, the process can be repeated until it converges to a fixed point, which is precisely a root, using

$$\varepsilon_n = -\frac{f(x_n)}{f'(x_n)}$$

Unfortunately, this procedure can be unstable near a horizontal asymptote or a local extremum. However, with a good initial choice of the root's suspected position, the algorithm can be applied iteratively to obtain

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad \text{where} \quad n = 1, 2, 3, \ldots$$

The error ε_{n+1} after the $(n+1)$st iteration can be shown to be approximately given by

$$\varepsilon_{n+1} = \varepsilon_n + (x_{n+1} - x_n) = \varepsilon_n - \frac{f(x_n)}{f'(x_n)} \approx -\frac{f''(x_{n-1})}{2f'(x_{n-1})}\varepsilon_n^2$$

Therefore, when the Newton method converges, it does so quadratically. The code below shows how one might implement the Newton-Raphson method for solving nonlinear equations in C#.

```
static double F1(double x)
{ return Math.Sin(x) - x*x*x*x; }
static double F1prime(double x)
{ return Math.Cos(x) - 4.0*x*x*x; }

public static double NewtonRaphsonMethod(Function f, Function fprime,
     double x0, double epsilon)
{
   double f0 = f(x0);
   double x = x0;
   while (Math.Abs(f(x)) > epsilon)
   {
      x -= f0 / fprime(x);
      f0 = f(x);
   }
   return x;
}

static void Main(string[] args)
{
  Console.WriteLine("\n\nTesting Testing Newton-Raphson Method\n");
  double x = NewtonRaphsonMethod(F1, F1prime, 1.0, 0.0001);
  Console.WriteLine("Solution Newton-Raphson method:"+x.ToString());
  Console.WriteLine("Solution confirmation:f(x)="+F1(x).ToString());
  Console.ReadLine();
}

OUTPUT: Testing Newton-Raphson Method
Solution from the Newton-Raphson method: 0.949616692330961
Solution confirmation: f(x) = -1.02824210257424E-08
```

10

Random Numbers

10.1 Introduction

Random numbers, sometimes also referred to as *random variables*, are simply numbers chosen by chance so that there are no correlations of any kind between successive values. As logical and deterministic machines, computers are unable to generate pure random numbers but must instead rely on some kind of mathematical algorithm which only gives the illusion of generating random numbers. Although some algorithms for generating random numbers have a very large periodicity, they are all flawed to some extent and eventually will start repeating their output sequence all over again. Consequently, computer-generated random numbers are sometimes called *pseudo-random numbers* to distinguish them from pure random numbers which are usually extracted as bit sequences from the output of natural unpredictable physical processes such as radioactive decay or lightning strikes. A pure random variable can therefore be thought of as a function mapping the sample space of a random process to the real numbers. On the other hand, pseudo-random numbers are generated by computers through the use of special algorithms called pseudo-random number generators, often abbreviated as PRNG or RNG.

Traditionally, the term *random numbers* applies to pseudo-random numbers that arise from a uniform distribution whereas *random variates* applies to pseudo-random numbers generated from some other distribution. However, Gentle [49] points out that these two terms along with the additional term *random deviates* can all be used interchangeably. Regardless of the terminology that is ultimately used, high-quality random numbers are very important in a wide range of mathematical, statistical, engineering and scientific applications. For example, computer simulation of physical processes use random numbers to study system behavior under different scenarios in a virtual environment that mimics real-world conditions. Such studies often lead to significant improvements and optimization of system performance. Reliable sources of random numbers with statistical properties indistinguishable from true random numbers are therefore a fundamental necessity for successfully carrying out such tasks. In addition, computational resources for obtaining true random numbers from naturally occurring physical processes will also be provided later on in this chapter.

A random number generator is a computational or physical device designed to generate a sequence of numbers or symbols that lack any visible or detectable pattern and thus appear to be truly random in nature. Due to a very high demand for random numbers, many different methods for generating them have appeared and these methods may vary greatly as to how unpredictable or statistically random they are and how quickly they can be produced. Most programming languages, including C#, have a built-in pseudo-random number generator, often abbreviated as RNG or PRNG, which is good enough for low-keyed relatively simple everyday applications. However, such RNGs are often inadequate for use in more serious applications such as cryptography or Monte Carlo simulations. This is because all the algorithms that have been developed for producing random numbers will eventually start repeating themselves after a certain amount of cycles and thus will produce a noticeable numerical pattern. Although a detailed discussion of what makes a RNG really *good* is beyond the scope of this chapter, it suffices to say that a good RNG should at least satisfy the following criteria:

- Good distribution. The points should be distributed according to what one would expect from a truly random distribution. Furthermore a pseudo-random number generator should not introduce artificial correlations between successively generated points.

- Long period. Both pseudo-random and quasi-random generators always have a period, after which they begin to generate the same sequence of numbers over again. To avoid undesired correlations one should in any practical calculation not come anywhere near exhausting the period.

- Repeatability. For testing and development, it may be necessary to repeat a calculation with exactly the same random numbers as in the previous run. Furthermore the generator should allow the possibility to repeat a part of a job without doing the whole thing. This requires the ability to store the state of a generator.

- Long disjoint subsequences. For large problems it is extremely convenient to be able to perform independent subsimulations whose results can later be combined assuming statistical independence.

- Portability. This means not only that the code should be portable but that it should generate exactly the same sequence of numbers on different machines.

- Efficiency. The generation of the pseudo-random numbers should not be too time-consuming. Almost all generators can be implemented in a reasonably efficient way.

10.2 The C# Built-In Random Number Generator

The .NET Framework has implemented a uniform built-in pseudo-random number generator in the `System.Random()` class. These pseudo-random numbers, from now on referred to simply as random numbers, are chosen with equal probability from a finite set of allowed `int32` values. The current implementation of the `Random()` class is based on Knuth's subtractive random number generator algorithm [50] which has an alleged period of about 2^{32}. This period is usually considered good enough for most practical applications but is definitely not recommended for cryptography or other applications that require a substantial degree of security such as password generation or encrypted telecommunications.

The random number generator provided by the .NET Framework starts from a seed value and can be overloaded in two different ways. By default, the parameterless constructor of the `Random()` class uses the computer's internal system clock to generate its own seed value whereas the parameterized constructor can accept specific seed values from within the allowed range of `Int32` integers. Because the system clock is continuously changing in value, using a parameterless constructor will therefore initiate a different randomized numerical sequence every time it is called up. However, since the clock has finite resolution, using the parameterless constructor to create different `Random()` objects in close succession can create random number generators that produce identical randomized numerical sequences. This problem can be avoided by creating one `Random()` object to generate many random numbers over time, instead of repeatedly creating several new `Random()` objects to generate one random number at a time. However, some of these features of the random number generator can sometimes be useful. For example, if you want to repeat the same sequence of numbers, you can set the seed state during instantiation by passing the same integer value to the constructor.

Once this `Random()` object has been created we can call the `Next()` method to produce a random number. The `Next()` method can be overloaded in three different ways. Without any input parameters, the `Next()` method returns a random number anywhere from 0 to the maximum value of `int32.Max` which is $2,147,483,647$. The `Next(int32)` method returns a nonnegative random number from 0 to the `Int32` integer value specified by the single input parameter. Lastly, the `Next(Int32,Int32 )` method returns a random number from anywhere within the allowed `Int32` integer range specified by the two input parameters.

The `Random.NextBytes()` method fills the elements of a specified array of bytes with random bytes selected anywhere from 0 to 255. Therefore, the `Random()` class can also be used to populate a byte array with random bytes.

Finally, the `Random.NextDouble()` method is not overloaded and only returns a double-precision floating point from the interval $[0,1]$. However, by properly adjusting the call to the `Random.NextDouble()` method this way

```
a + (b - a)*rnd.NextDouble();
```

where rnd is an object of the `Random()` class, we can obtain random numbers from anywhere within the numerical interval specified by $[a,b)$. Some actual examples are listed below in order to illustrate the implementation of all these ideas.

```
static void TestRNG()
{
  int seed = 234;
  int  min = -55;
  int  max =  25;

  Console.WriteLine("Testing 6 random int32 from 0 to int32.MAX");
  Random randObj1 = new Random(seed);
  for (int j = 0; j < 6; j++)
    Console.Write("{0,11} ", randObj1.Next());

  Console.WriteLine("Testing 6 random int32 from 0 to 25");
  Random randObj2 = new Random(seed);
  for (int j = 0; j < 6; j++)
    Console.Write("{0,11} ", randObj2.Next(max));

  Console.WriteLine("Testing 6 random int32 from -55 to 25");
  Random randObj3 = new Random(seed);
  for (int j = 0; j < 6; j++)
    Console.Write("{0,11} ", randObj3.Next(min, max));

  Console.WriteLine("Testing 5 random bytes: 0 to 255");
  Random randObj4 = new Random();
  Byte[] b = new Byte[10];
  randObj4.NextBytes(b);
  Console.WriteLine("The Random bytes are: ");
  for (int i = 0; i < 10; i++)
  {
    Console.Write(i);
    Console.Write(":");
    Console.WriteLine(b[i]);
  }

  Console.WriteLine("Testing 6 random doubles from 0 to 1");
  Random randObj5 = new Random(seed);
  for (int j = 0; j < 6; j++)
  Console.WriteLine("{0,11} ",randObj5.NextDouble());

  Console.WriteLine("Testing 6 random doubles from -55 to 25");
  Random randObj6 = new Random(seed);
  for (int j = 0; j < 6; j++)
  Console.WriteLine("{0,11}",(min+(max-min)*randObj6.NextDouble()));
}
```

A useful application of the random number generator class is a utility that pertains to generating passwords that can contain random letters or characters in addition to random numbers. Since random numbers are generated by default, we must create a new method that will contain this functionality. The implementation of a random password generator is given below.

```
static string CreateRandomPassword(int passwordLength)
{
  string allowedChars =
  "abcdefghijkmnopqrstuvwxyzABCDEFGHJKLMNOPQRSTUVWXYZ0123456789";
  char[] chars = new char[passwordLength];
  Random rd = new Random();
  for (int i = 0; i < passwordLength; i++)
  {
    chars[i] = allowedChars[rd.Next(0,allowedChars.Length)];
  }
  return new string(chars);
}

static void TestRandomPassword()
{
 Console.WriteLine("Testing for generation of random passwords");
 for (int i = 0; i < 6; i++)
 {
   Console.WriteLine("Password {0}={1}",i,CreateRandomPassword(10));
   Thread.Sleep(2000);
 }
 Console.ReadLine();
}
```

Microsoft also provides an encryption class, called `RNGCryptoServiceProvider`, for use in developing secure applications, including allegedly secure random number generators. However, at the time of this writing there was considerable debate over the Internet as to the level of security that this class actually is capable of providing. Since technology advances so rapidly, you might want to double check the current reports on this topic when you read this to see what the current security level status of this is really like. In any event, below are two additional examples illustrating the use of this Microsoft encryption class.

```
public static void BetterRandomString()
{
  // create a stronger hash code using RNGCryptoServiceProvider
  byte[] random = new byte[64];
  RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
  // populate with random bytes
  rng.GetBytes(random);
  // convert random bytes to string
  string randomBase64 = Convert.ToBase64String(random);
  // display
  Console.WriteLine("Random string: {0}\r\n ",randomBase64);
}

public static string CreateRandomEncryptedPassword(int PasswordLen)
{
  String allowedChars =
  "abcdefghijkmnopqrstuvwxyzABCDEFGHJKLMNOPQRSTUVWXYZ0123456789";
  Byte[] randomBytes = new Byte[PasswordLen];
  RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
  rng.GetBytes(randomBytes);
  char[] chars = new char[PasswordLen];
  int allowedCharCount = allowedChars.Length;
  for (int i = 0; i < PasswordLen; i++)
  {
   chars[i] = allowedChars[(int)randomBytes[i]%allowedCharCount];
  }
  return new string(chars);
}
```

```
static void TestRandomEncryptedPassword()
{
    for (int i = 0; i < 6; i++)
    {
        Console.WriteLine("Encrypted Password {0} = {1}",
                    i, CreateRandomEncryptedPassword(10));
        Thread.Sleep(2000);
    }
    Console.ReadLine();
}
```

By default, the random number methods within the `Random()` class generate numbers with replacement. This means that a particular random number may get generated repeatedly. If you do not want a set of randomly generated numbers to contain any duplication of elements, you need to code the process of generating random numbers accordingly. There are a number of ways to obtain a list of unique random numbers containing no repetition. For pedagogical reasons, I thought it would be instructive to present at least three different methods to generate random numbers having no duplicate values.

In the first and perhaps simplest method, you can just put the values you want randomized (numbers, strings, objects,...) in an array and then shuffle the array by randomly exchanging the array elements. This procedure, using the Knuth-Fisher-Yates shuffle algorithm, will produce a randomized list of unique numerical values as illustrated in the example below.

```
static int[] UniqueRandom1(int max)
{
  Random rand = new Random();
  int[] array = new int[max];

  // Initialize the array to integers from 0 to max
  for (int i = 0; i < array.Length; i++)
        { array[i] = i; }

  for (int i = array.Length - 1; i > 0; i--)
  {
      int randomPosition = rand.Next(i+1);
      int temp = array[i];
      array[i] = array[randomPosition];
      array[randomPosition] = temp;
  }
  return array;
}

static void TestUniqueRandom1()
{
  int size = 15;
  int[] myData = UniqueRandom1(size);
  Console.WriteLine("Testing unique random numbers version 1\n");
  for (int i = 0; i < myData.Length; i++)
        { Console.WriteLine("myData[{0}] = {1}", i, myData[i]); }
}
```

As an alternate method for obtaining a list of unique random numbers you can also start by first creating an `ArrayList` object to hold the numbers. Next, generate a random number within range of the desired values and add this number to the list. Then loop repeatedly generating another random numbers and checking whether each of them is already contained in the list. If the number is not found in the list, then add that number to the list. Otherwise, if the number is found in the list then ignore it and get another random number. Repeat this process until the list has been completely filled. The random numbers left in the list should all be unique without any repetition. The code below implements this idea.

```
static ArrayList UniqueRandom2(int max)
{
  // Create an ArrayList object that will hold the numbers
  ArrayList lstNumbers = new ArrayList();
  // Create an object from the Random() class that
  // will be used to generate random numbers
  Random rndNumber = new Random();

  // Get a random number between 0 and the max requested value
  int number = rndNumber.Next(0, max);
  // Add this first random number to the list
  lstNumbers.Add(number);
  // Setup a number counter to keep track of the
  // numbers being added to the list
  int count = 0;
  do // Repeatedly...
  {
    // generate a random number between 0 and the max
    number = rndNumber.Next(0, max);

    // If the newly generated number in not yet in the list...
    if (!lstNumbers.Contains(number))
    {
      lstNumbers.Add(number); // ... add it
      count++;                // and increase the counter
    }
  } while (count <= max);
  // Once the list is built, return it
  return lstNumbers;
}

static void TestUniqueRandom2()
{
  const int Total = 15;
  ArrayList lstNumbers = UniqueRandom2(Total);

  for (int i = 0; i < lstNumbers.Count; i++)
    Console.WriteLine("x[{0}] = {1}", i, lstNumbers[i]);
  Console.ReadLine();
}
```

As a final, but perhaps much subtler, example of a method for obtaining a list of unique random numbers consider the following algorithm and its subsequent implementation shown below. First, a generic list is created and loaded with the values you want randomized (numbers, strings, objects,...). Individual elements from this list are then randomly selected and added to another list. After an element has been selected and added to the second list, it is removed from the original list so that it cannot be selected again. This process is repeated until there are no more elements to select from in the original list. An implementation of this algorithm in C# is given below.

```
public static List<int> GetUniqueRandomList(int size)
{
  List<int> list = new List<int>();
  Random randomGenerator = new Random();

  List<int> range = new List<int>();
  for (int i = 0; i < size; i++)
     range.Add(i);

  while (range.Count > 0)
  {
    int item = range[randomGenerator.Next(range.Count)];
    list.Add(item);
    range.Remove(item);
  }
  return list;
}

static void TestUniqueRandom3()
{
  List<int> iList = new List<int>();
  iList = GetUniqueRandomList(15);

  foreach (int i in iList)
  {
    Console.WriteLine(i);
  }
  Console.ReadLine();
}
```

10.3 Other Random Number Generators

There are many different types of random number generators [51] but the *Linear Congruential Generator* (LCG) represents one of the oldest and best-known pseudorandom number generator algorithms [50]. The linear congruential generator is defined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \mod m$$

where X_n is the sequence of pseudorandom values, $m > 0$ is called the modulus, a where $0 < a < m$ is called the multiplier, c where $0 \leq c < m$ is called the increment, and X_0 where $0 \leq X_0 < m$ is called the *seed* or *start* value. Together these values specify the LCG. The period of the LCG is at most m, and largely depends on the value chosen for the other variables. The LCG will have a full period if and only if c and m are relatively prime, $a-1$ is divisible by all prime factors of m and $a-1$ is a multiple of 4 if m is a multiple of 4. While capable of producing decent pseudorandom numbers, LCGs are extremely sensitive to the choice of the coefficients c, m, and a. The most efficient linear congruent generators, including the one built into Visual C#, reportedly have an m equal to a power of 2, most often around $m = 2^{32}$.

Perhaps the most famous and controversial one of all the random number generators is the *Mersenne Twister* published in 1998 by Matsumoto and Nishimura [52, 53], which has a reportedly proven period of $2^{19937}-1$. In practice, however, there is little reason to use anything larger as most applications do not require 2^{19937} unique combinations ($2^{19937} \approx 4.3 \times 10^{6001}$). Nevertheless, the Mersenne Twister algorithm has received some criticism in the computer science field. Critics claim that while the Mersenne Twister is good at generating random numbers, it is not very elegant and is overly complex to implement. As an alternative to the Mersenne Twister, critics have proposed a simpler complementary multiply-with-carry generator [54] with an alleged period of 10^{33000} which claims to be significantly faster and maintains better or equal randomness. In any event, a C# implementation of the Mersenne Twister random number generator algorithm is given below along with sample code to demonstrate how to use it. Those readers wishing to learn more about how this algorithm works are referred to the original paper [52]. Those readers looking for an alternative to either the Visual C# or the Mersenne Twister random number generators are referred to another website [55] that maintains a very good list of random number generators including code in C/C++ and Fortran that can be downloaded directly to your computer.

```
public class MersenneTwister
{
    // Class MersenneTwister generates random numbers
    // from a uniform distribution using the Mersenne
    // Twister algorithm.
    private const int N = 624;
    private const int M = 397;
    private const uint MATRIX_A = 0x9908b0dfU;
    private const uint UPPER_MASK = 0x80000000U;
    private const uint LOWER_MASK = 0x7fffffffU;
    private const int MAX_RAND_INT = 0x7fffffff;
    private uint[] mag01 = {0x0U, MATRIX_A};
    private uint[] mt = new uint[N];
    private int    mti = N+1;

    public MersenneTwister()
    { init_genrand( (uint)DateTime.Now.Millisecond); }

    public MersenneTwister( int seed )
    {
      init_genrand( (uint)seed );
    }
```

```
public MersenneTwister( int[] init )
{
  uint[] initArray = new uint[init.Length];
  for ( int i = 0; i < init.Length; ++i )
    initArray[i] = (uint)init[i];
  init_by_array( initArray, (uint)initArray.Length);
}

public static int MaxRandomInt
{ get { return 0x7fffffff; } }

public int Next()
{ return genrand_int31(); }

public int Next( int maxValue )
{ return Next( 0, maxValue ); }

public int Next( int minValue, int maxValue )
{
  if ( minValue > maxValue )
  {
    int tmp = maxValue;
    maxValue = minValue;
    minValue = tmp;
  }
  return (int)(Math.Floor((maxValue-minValue+1)*genrand_real1()+
      minValue));
}

public float NextFloat()
{ return (float) genrand_real2(); }

public float NextFloat( bool includeOne )
{
  if ( includeOne )
  {
    return (float) genrand_real1();
  }
  return (float) genrand_real2();
}

public float NextFloatPositive()
{ return (float) genrand_real3(); }

public double NextDouble()
{ return genrand_real2(); }

public double NextDouble( bool includeOne )
{
  if ( includeOne )
  {
    return genrand_real1();
  }
  return genrand_real2();
}
```

```
public double NextDoublePositive()
{ return genrand_real3(); }

public double Next53BitRes()
{ return genrand_res53(); }

public void Initialize()
{ init_genrand((uint)DateTime.Now.Millisecond); }

public void Initialize( int seed )
{ init_genrand( (uint)seed ); }

public void Initialize( int[] init )
{
  uint[] initArray = new uint[init.Length];
  for ( int i = 0; i < init.Length; ++i )
    initArray[i] = (uint)init[i];
  init_by_array( initArray, (uint)initArray.Length );
}

private void init_genrand( uint s)
{
  mt[0]= s & 0xffffffffU;
  for (mti=1; mti<N; mti++)
  {
    mt[mti] =
      (uint)(1812433253U*(mt[mti-1]^(mt[mti-1]>>30))+mti);
    mt[mti] &= 0xffffffffU;
  }
}

private void init_by_array(uint[] init_key, uint key_length)
{
  int i, j, k;
  init_genrand(19650218U);
  i=1; j=0;
  k = (int)(N>key_length ? N : key_length);
  for (; k>0; k--)
  {
    mt[i] = (uint)((uint)(mt[i]^((mt[i-1]^(mt[i-1]>>30))*1664525U
        ))+init_key[j]+j);
    mt[i] &= 0xffffffffU;
    i++; j++;
    if (i>=N) { mt[0] = mt[N-1]; i=1; }
    if (j>=key_length) j=0;
  }
  for (k=N-1; k>0; k--)
  {
    mt[i] = (uint)((uint)(mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) *
        1566083941U))- i);
    mt[i] &= 0xffffffffU;
    i++;
    if (i>=N) { mt[0] = mt[N-1]; i=1; }
  }
  mt[0] = 0x80000000U;
}
```

```
    uint genrand_int32()
    {
      uint y;
      if (mti >= N)
      {
        int kk;
        if (mti == N+1)
          init_genrand(5489U);
        for (kk=0;kk<N-M;kk++)
        {
          y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
          mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1U];
        }
        for (;kk<N-1;kk++)
        {
          y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
          mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1U];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1U];
        mti = 0;
      }
      y = mt[mti++];
      y ^= (y >> 11);
      y ^= (y << 7) & 0x9d2c5680U;
      y ^= (y << 15) & 0xefc60000U;
      y ^= (y >> 18);
      return y;
    }

    private int genrand_int31()
    { return (int)(genrand_int32()>>1); }

    double genrand_real1()
    { return genrand_int32()*(1.0/4294967295.0); }

    double genrand_real2()
    { return genrand_int32()*(1.0/4294967296.0); }

    double genrand_real3()
    {return (((double)genrand_int32())+0.5)*(1.0/4294967296.0);}

    double genrand_res53()
    {
      uint a=genrand_int32()>>5, b=genrand_int32()>>6;
      return(a*67108864.0+b)*(1.0/9007199254740992.0);
    }
}

static void TestMersenneTwister()
{
  MersenneTwister randGen = new MersenneTwister();
  Console.WriteLine( "100 uniform random integers in [0,{0}]:",
      MersenneTwister.MaxRandomInt);
  int i;
  for (i = 0; i < 100; ++i)
  {
    Console.Write("{0} ",randGen.Next());
    if ( i%5 == 4 ) Console.WriteLine("");
  }

  Console.WriteLine("100 uniform random doubles in [0,1]:");
  for ( i = 0; i < 100; ++i )
  {
    Console.Write("{0} ",randGen.NextDouble().ToString("F8"));
    if ( i%5 == 4 ) Console.WriteLine("");
  }
  Console.WriteLine("Press ENTER to quit");
  Console.ReadLine();
}
```

10.4 True Random Number Generators

As we have seen, C# comes with its own built-in pseudo-random number generator that has an alleged period of about 2^{32} which is adequate for most practical but non-critically secure applications. More recently, however, far more robust pseudo-random number generators have been developed. The most prominent among these are perhaps the Mersienne Twister [52, 53] with an alleged period of $2^{19937} - 1$ and the multiply-with-carry generator [54] with an reported period of about 10^{33000}. With such huge periods, these pseudo-random number generators have been very useful in a wide range of advanced applications, such as encrypted telecommunications, where security issues are a major concern.

True random numbers, however, can only be obtained from observing and recording naturally occurring random physical phenomena. As a result, a few websites with links and interfaces to certain natural events have appeared on the Internet and can therefore generate true random numbers for us. At the time of this writing, I have compiled a list of some of these websites below and included a brief description of the nature with which they can collect true random numbers from naturally occurring physical events.

- Fourmilab is located in Switzerland [56] and offers free genuine random numbers by timing successive pairs of radioactive decays detected by a Geiger-Müller tube interfaced to a computer. Because of the radioactive nature of these random numbers, they call them *Hot Bits*. Since the HotBits generation hardware produces data at a modest rate of about 100 bytes per second, requests are filled from an inventory of pre-built HotBits. You order up your serving of HotBits by filling out a request form on their webpage and specifying how many random bytes you want and in which format you would like them delivered. Your request is relayed to the HotBits server, which flashes the random bytes back to you over the Web. An alternative to downloading HotBits for later use is also provided by their downloadable software package in Java called randomX. A program developed with randomX can select from a variety of pseudo-random sequence generators or genuine random data from HotBits, obtained on demand across the Internet.

- LavaRnd [57] is located in Sunnyvale, California U.S.A. and offers a random number generator that works by measuring noise from a CCD camera such as an inexpensive webcam. The CCD is enclosed in a light-proof container, and operated at a high gain. The resulting images are not perfectly black and instead contain some noise. The LavaRnd system takes noisy data from the CCD and runs it through an algorithm called the Digital Blender to produce data that is more uniformly random and which you can download free of charge directly from their website.

- Araneus Information Systems [58] is a company located in Finland that sells a random number generating electronic device, called Alea I, that fits directly into the USB port of your computer. The Alea I uses a reverse biased semiconductor junction to generate wide-band Gaussian white noise. This noise is amplified and digitized using an analog-to-digital converter. The raw output bits from the A/D converter are then further processed by an embedded microprocessor to combine the entropy from multiple samples into each final random bit and remove any bias caused by imperfections in the noise source and A/D converter.

- Quantis [59] is a physical random number generator that was developed at the University of Geneva, Switzerland. This device works by exploiting the randomness of an elementary quantum optics process. Photons, better known as light particles, are sent one by one onto a semi-transparent mirror and detected. The exclusive events, reflection and transmission, are associated to either 0 or 1 bit values. Users can download batches of random numbers directly from their website. The operation of Quantis is continuously monitored to ensure immediate detection of a failure and disabling of the random bit stream.

- Located in Zagreb, Croatia, the Quantum Random Bit Generator [60] is a fast non-deterministic random number, actually bit, generator whose randomness relies on intrinsic randomness of the quantum physical process of photonic emission in semiconductors and subsequent detection by photoelectric effect. In this process photons are detected at random, one by one independently of each other. Timing information of detected photons is used to generate random binary digits or bits which, of course, can then easily be converted to numbers. The unique feature of this method is that it uses only one photon detector to produce both zeros and ones which results in a very small bias and high immunity to components variation and aging. Furthermore, detection of individual photons is made by a photo-multiplier tube (PMT). Compared to solid state photon detectors the PMT's have drastically superior signal to noise performance and much lower probability generating after-pulses which could be a source of unwanted correlations.

- RANDOM.ORG [61], located in Dublin, Ireland, offers true random numbers free of charge to anyone on the Internet. The randomness comes from measuring atmospheric noise. Because of the high demand for this service, they keep track of your IP address and users are limited to download only a certain number of random bits per month. However, more bits can be purchased for cash if so desired. This organization has very graciously provided the public with a nice detailed information page to guide their clients on how to write computer programs to interface directly to their server through Hyper-Text Transfer Protocol (HTTP). There is also the HTTP client archive, which contains clients that other people have written. An example of a C# HTTP interface application to request a download of some true random numbers from their website is given below.

```
// Add these extra Microsoft .NET Framework Class
// Library namespaces to the top of your program page

using System.IO;
using System.Net;

//Returns an array of random integers
//between two numbers, both inclusive

public static int[] GetRandomInts(int min,int max,int trials)
{
 //Build the url string to www.random.org
 string url="http://www.random.org/integers/?num="+trials.ToString();
 url += "&min=" + min.ToString();
 url += "&max=" + max.ToString();
 url += "&col=1&base=10&format=html&rnd=new";

 string data = DownloadData(url);

 if (data != string.Empty)
 {
  //Parse the data
  string startMarker="<pre class="+'"'+"data"+'"'+">";
  int j = data.IndexOf(startMarker);
  if (j != -1)
  {
    int k = data.IndexOf("</pre>", j);
    if (k != -1)
    {
      string intString =
      data.Substring(j+startMarker.Length,k-j-startMarker.Length);
      intString = intString.Trim();

      //Read each line
      List<int> integers = new List<int>();
      StringReader readLines = new StringReader(intString);
      while (readLines.Peek() != -1)
      {
        integers.Add(int.Parse(readLines.ReadLine()));
      }
      return integers.ToArray();
      }
    }
  }
  return new int[] { -1 };
}

//Connects to URL to download data
private static string DownloadData(string url)
{
  try
  {
    //Get a data stream from the url
    WebRequest req = WebRequest.Create(url);
    WebResponse response = req.GetResponse();
    Stream stream = response.GetResponseStream();

    //Download in chuncks
    byte[] buffer = new byte[1024];

    //Get Total Size
    int dataLength = (int)response.ContentLength;

    //Download to memory
    //Note: adjust the streams here to download
    //directly to the hard drive
    MemoryStream memStream = new MemoryStream();
    while (true)
    {
      //Try to read the data
      int bytesRead = stream.Read(buffer,0,buffer.Length);

      if (bytesRead == 0) break;
      else memStream.Write(buffer,0,bytesRead);
    }

    //Convert the downloaded stream to a byte array
    string downloadedData =
    System.Text.ASCIIEncoding.ASCII.GetString(memStream.ToArray());

    //Clean up
    stream.Close();
    memStream.Close();

    return downloadedData;
  }
  catch (Exception)
  {
    return string.Empty;
  }
}
```

```
//Sample driver program for testing purposes only
static void Main(string[] args)
{
  //Sets up amount of random numbers to download
  int NToDownload = 10;
  int min = 10; //Sets up the range of values
  int max = 100;
  string numberArray = "";
  Console.WriteLine("\nAttempting to download true random numbers
      from RANDOM.ORG server. Please wait..");
  //Download numbers into integer array
  int[] randomNumbers = GetRandomInts(min, max, NToDownload);
  foreach (int rnd in randomNumbers)
  {
    numberArray += rnd.ToString() + Environment.NewLine;
  }
  Console.WriteLine(numberArray);
  Console.WriteLine("\nDone!");
  Console.ReadLine();
}
```

10.5 Random Variate Generation Methods

Studies have shown that a significant number of naturally occurring physical phenomena tend to follow some particular type of probability distribution [49]. For example, nuclear decay of atoms can be modeled using the Poisson distribution. Consequently, in order to effectively model the behavior of many physical systems in the laboratory, it would be highly desirable to program computers to generate random variates according to specific probability distributions. Although there are many different methods for carrying out such calculations [49], some may be more computationally efficient than others. Regardless of the method chosen, it would be pedagogically prudent to start this section by first reviewing a few useful and well known fundamental concepts from the theory of probability and statistics including the four most popular methods for generating random variates according to specific probability distributions. This material is followed by a set of short summaries and C# code implementations of these methods for the most common probability distributions.

A probability distribution is a statistical function that describes the range of all possible values that a random variable can attain and the probability that the value of the random variable is within a measurable subset of that range. Random variables, along with their associated probability distributions can be classified as being either *discrete* or *continuous*.

A discrete random variable is one which can take on only a countable number of distinct positive integral values. Conversely, if a random variable can take only a finite number of distinct values, then it must be discrete. For example, if you toss a coin, you are going to get either heads or tails, but never something like 1.254374 heads. The probability function for discrete random variables is called the *probability mass function* (PMF) $f_X(x)$ and gives the probability that a discrete random variable X in some well defined sample space is exactly equal to some specific value x. That is, suppose that $X : S \to \mathbb{R}$ is a discrete random variable defined in a sample space S that is mapped onto the set of real numbers $\mathbb{R}$. Then the probability mass function $f_X(x)$ for X is defined as $f_X(x) = P(X = x)$. Consequently, $f_X(x) = 0$ for all $x \notin X(S)$. Therefore, the probability distribution, $P(x)$, of a discrete random variable is simply a list of probabilities associated with each of its possible values. More formally, a discrete probability distribution, $P(x)$, is a function that satisfies the following properties:

- The probability that a discrete random variable x can take a specific value x_i is given by $P(x_i)$ where $i = 1, 2, \ldots, \infty$
- $0 \leq P(x_i) \leq 1$ for all discrete x_i.
- $\sum_{i=1}^{\infty} P(x_i) = 1$.

As a result, a function that allows negative values or values greater than one is not a discrete probability distribution function. The condition that the sum of all the probabilities that a discrete random variable x can take equals one means that at least one of the possible values that x can take has to occur.

A continuous random variable is one which can take on an infinite number of possible values. Continuous random variables are usually measurements expressed as real numbers. Typical examples include height, weight, or the time required to run a mile. The probability function for continuous random variables is called the *probability density function* (PDF) $f(x)$ and describes the probability density at each point in the associated sample space S. The probability of a random variable falling within a given set is therefore given by the integral of the PDF over that set. Since a continuous probability function is defined for an infinite number of points over a continuous interval, the measurement of probability at any single point in that interval is always zero. Instead, continuous probabilities are measured over intervals, not single points and the area under the curve between two distinct points therefore defines the probability for that interval. More formally, the probability distribution, $P(x)$, of a continuous random variable X is a function that satisfies the following properties:

- The probability that a random variable X is between two points a and b is given by: $P(a \leq X \leq b) = \int_a^b f(x)\,dx$.
- $0 \leq P(X) \leq 1$ for all real X.
- The integral of the probability function is one, *i.e.* $\int_{-\infty}^{+\infty} f(x)\,dx = 1$.

Note that a PMF differs from a PDF in that the values of a PDF, defined only for continuous random variables, are not individual probabilities. Instead, it is the integral of a PDF over a range of possible values that gives the probability of the random variable falling within that range. In addition, not every probability distribution has a density function. The distributions of discrete random variables, for example, do not.

All random variables, whether they are discrete or continuous, have a *cumulative distribution function* (CDF) $F_X(x)$, which gives the probability that a random variable X is less than or equal to x, for every value x. More formally, for every real number x, the CDF of X is given by $F_X(x) = P(X \leq x)$. If X is a discrete random variable, then it attains values $x_1, x_2, \ldots$ with corresponding probabilities $P(x_1), P(x_2), \ldots$ so that the CDF $F_X(x)$ can be written as:

$$F_X(x) = P(X \leq x) = \sum_{x_i \leq x} P(X = x_i) = \sum_{x_i \leq x} P(x_i)$$

However, if X is a continuous random variable with a corresponding continuous PDF $f(x)$ then the CDF $F_X(x)$ can be written as:

$$F_X(x) = P(X \leq x) = \int_{-\infty}^{x} f(t)dt$$

where t is just a dummy variable used for integration purposes. As a result, the CDF for discrete random variables is the sum of its discrete probabilities whereas the CDF for continuous random variables is the integral of its probability density function. For continuous variables, the CDF $F_X(x)$, and PDF $f(x)$, are therefore related by

$$f(x) = \frac{d}{dx} F_X(x) \text{ so that } p[a \leq X \leq b] = \int_a^b f(x)\,dx = F_X(b) - F_X(a)$$

Other important statistical functions of interest include the mean, variance and standard deviation. The mean or expected value of a probability distribution is given by

$$\bar{x} = E(x) = \sum_{i=1}^{n} p_i x_i = \int_{-\infty}^{\infty} x f(x)\,dx$$

where summation is used for discrete distributions and integration is used for continuous distributions respectively. The quantity $(x - \bar{x})^2$ represents the square of distance between x and its mean $\bar{x}$. The expected value of this quantity is called the variance of x and is given by

$$\sigma^2 = Var(x) = E((x - \bar{x})^2) = \sum_{i=1}^{n} p_i (x - \bar{x})^2 = \int_{-\infty}^{\infty} (x - \bar{x})^2 f(x)\,dx$$

The square root of the variance is called the standard deviation and is denoted by σ. A possible implementation of these basic three statistical functions in C# using raw, discrete data values is shown below.

```
public static void stats(double[] data, out double mean, out double
    variance, out double sigma)
{
 double sum = 0.0;
 mean = 0.0;
 variance = 0.0;
 sigma = 0.0;

 for (int i = 0; i < data.Length; i++)
 {
   sum += data[i];
 }
 mean = sum / data.Length;

 for (int i = 0; i < data.Length; i++)
 {
   variance += Math.Pow(data[i] - mean, 2);
 }
 variance /= data.Length;
 sigma = Math.Sqrt(variance);
 return;
}

static void TestBasicStats()
{
 //Generate 10000 random points following a normal distribution
 //with the following parameters:
 int nPoints = 10000; double mu = 2.0; double sigma = 0.5;
 double[] randObj = NextNormal(mu, sigma, nPoints);

 //Calculate the basic statistical functions:
 //mean, variance and standard deviation.
 double mean = 0.0, variance = 0.0, CalcSigma = 0.0;
 stats(randObj, out mean, out variance, out CalcSigma);

 Console.WriteLine("\nTesting Stats Using a Normal Distribution");
 Console.WriteLine("\nInput parameters: number of points = 10,000,
                    mu = 2.0, sigma = 0.5");
 Console.WriteLine("\nResults (calculated):\n");
 Console.WriteLine("Mean={0}",Math.Round(mean, 4));
 Console.WriteLine("Variance={0}",Math.Round(variance, 4));
 Console.WriteLine("Standard Deviation={0}",Math.Round(CalcSigma,4));
 Console.WriteLine("\nResults (expected):\n");
 Console.WriteLine("Mean={0}",Math.Round(mu, 4));
 Console.WriteLine("Variance={0}",Math.Round(sigma*sigma,4));
 Console.WriteLine("Standard Deviation={0}",Math.Round(sigma,4));
 Console.WriteLine("\nPress ENTER key to continue...");
 Console.ReadLine();
 Console.Clear();
}
```

```
Output:

Testing Basic Stats Using a Normal Distribution
Input parameters: number of points = 10,000, mu = 2.0, sigma = 0.5

Results (calculated):           Results (expected):
Mean = 1.9958                   Mean = 2
Variance = 0.246                Variance = 0.25
Standard Deviation = 0.496      Standard Deviation = 0.5
```

The uniform distribution over the unit interval $[0,1]$ is usually the default distribution generated by pseudorandom numbers in a computer. Its PDF is given by

$$f(x) = \begin{cases} 1 & \text{if } 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

This distribution is usually denoted by $u[0,1]$. More generally, the notation $u[a,b]$ is used to denote the absolutely continuous uniform distribution over the interval $[a,b]$. The uniform distribution is particularly convenient because there are many simple techniques to transform uniform samples into samples from other distributions of interest. The basic idea is to first generate a random variate uniformly distributed on $u[0,1]$. Then, by using an appropriate transformation procedure, convert this random variate obtained to one from the desired distribution. Since probability distributions can be discrete or continuous, any transformation procedure must also account for the proper handling of discrete and continuous variables.

The four most common methods for generating univariate random variables are:

1. the inverse transform method,
2. the acceptance-rejection method,
3. the composition method, and
4. the convolution method.

Inverse transform sampling, also known as the *inverse probability integral transform* or the *inverse transformation method*, is a method for sampling random numbers from any probability distribution provided its CDF can be analytically or numerically inverted. Because of this limitation, for some probability distributions this method may be computationally expensive, impractical or even impossible to implement in practice. Nevertheless, the inverse transform method has been successfully applied to generate random variates from many important probability distributions and so it merits considerable attention.

Suppose X is a discrete random variable with a PMF given by

$$X = \begin{cases} x_1, & \text{with probability } p_1 \\ x_2, & \text{with probability } p_2 \\ x_3, & \text{with probability } p_3 \end{cases}$$

where $p_1 + p_2 + p_3 = 1$. We would like to generate a value of X and we can do this by using our uniform distribution generator $u[0,1]$:

1. Generate $u[0,1]$.
2. Set

$$X = \begin{cases} x_1, & \text{if } 0 \leq u \leq p_1 \\ x_2, & \text{if } p_1 < u \leq p_1 + p_2 \\ x_3, & \text{if } p_1 + p_2 < u \leq 1 \end{cases}$$

We can check that this result is correct by making the observation that

$$P(X = x_1) = P(0 \leq u \leq p_1) = p_1$$

since u is a random variate drawn from the uniform distribution $u[0,1]$. The same conclusion holds true for $P(X = x_2)$ and $P(X = x_3)$.

More generally, if X is a discrete random variable and can take on n distinct values $x_1 < x_2 < \ldots < x_n$ with

$$P(X = x_i) = p_i \ \text{ for } i = 1,2,\ldots,n$$

then to generate a sample value of X we

1. Generate $u[0,1]$.
2. Set $X = x_j$ if $\sum_{i=1}^{j-1} p_i < u \leq \sum_{i=1}^{j} p_i$
3. or equivalently, set $X = x_j$ if $F_X(x_{j-1}) < u \leq F_X(x_j)$.

Example: Suppose that X belongs to a geometric distribution with probability of success parameter p, $q = 1 - p$ and with a PMF given by

$$P(X = i) = p(1-p)^{i-1}$$

where $i \geq 1$. The CDF of the random variable is given by

$$\begin{aligned} F(i) &= P(X \leq i) \\ &= 1 - P(X > i) \\ &= 1 - P(\text{ first } i \text{ trials are failures }) \\ &= 1 - q^i \end{aligned}$$

Set $X = j$ if

$$\begin{aligned} F(j-1) \leq u[0,1] < F(j) &\Leftrightarrow 1 - q^{j-1} \leq u < 1 - q^j \\ &\Leftrightarrow q^j < 1 - u \leq q^{j-1} \end{aligned}$$

Therefore,

$$\begin{aligned} X &= \min\{j \,|\, q^j < 1 - u\} \\ &= \min\{j \,|\, j\log(q) < \log(1-u)\} \\ &= \min\{j \,|\, j > \frac{\log(1-u)}{\log(q)}\} \\ &= \min\{j \,|\, j > \frac{\log(u)}{\log(1-p)}\} \text{ where } 0 < u \leq 1.0 \\ &= \lceil \frac{\log(u)}{\log(1-p)} \rceil \end{aligned}$$

where $\lceil x \rceil$ is the smallest integer larger than x.

The general procedure for sampling variates from a distribution of continuous random variables is perhaps more easily understood if we first consider the resulting implications of the following theorem and its associated corollary.

Theorem: Consider a continuous random variable X with a CDF given by $F_X(x)$. Define a new variable $Y = F_X(x)$. Then $Y \sim u[0,1]$.

Proof:

$$\begin{aligned} F_Y(y) &= P(Y \leq y) \text{ by the definition of a CDF} \\ &= P(F_X(x) \leq y) \text{ since } Y = F_X(x) \\ &= P(X \leq F_X^{-1}(y)) \text{ since } F_X(x) \text{ is monotone} \\ &= F_X(F_X^{-1}(y)) = y \text{ from which } f(y) = \frac{dF}{dy} = 1 \text{ and therefore } Y \sim u[0,1]. \end{aligned}$$

Corollary: Let $u \sim u[0,1]$. Define a random variable $X = F^{-1}(u)$ where F is a CDF. Then F is the CDF of X.

Proof: $F_X(x) = P(X \leq x) = P(F^{-1}(u) \leq x) = P(u \leq F(x)) = F(x)$.

Given any continuous random variable X with a CDF $F_X(x)$, then the variable $u = F_X(x)$ is uniformly distributed between 0 and 1. Therefore, X can be obtained by first generating $u[0,1]$ and then computing $X = F^{-1}(u)$ provided that the inverse function F^{-1} exists. If X is a continuous random variable, then the following algorithm can be used to generate a sample value of X.

1. Generate $u[0,1]$.
2. Set $X = x$ if $F_X(x) = u$. That is, set $X = F_X^{-1}(u)$.

Proof:

$$\begin{aligned} P(X \leq x) &= P(F_X^{-1}(u) \leq x) \\ &= P(u \leq F_X(x)) \\ &= F_X(x) \end{aligned}$$

If F_X^{-1} does not exist, then we need to use a slightly different algorithm as shown below.

1. Generate $u[0,1]$.
2. Set $X = \min\{x : F_X(x) \geq u\}$

Note that this last algorithm works for both discrete and continuous random variables or mixtures of the two.

Example: Consider the Weibull distribution which has a PDF given by

$$f(x) = \frac{a}{b}\left(\frac{x}{b}\right)^{a-1} \exp\left[-\left(\frac{x}{b}\right)^{a}\right]$$

After calculating its corresponding CDF, then $F^{-1}(u)$ is that value of X for which $1-e^{-(x/b)^a} = u$. Solving for x, we obtain $x = b(-\log(1-u))^{1/a}$. For $u \sim u[0,1]$, u is uniform when $u-1$ is uniform and therefore we may also write $x = b(-\log(u))^{1/a}$.

Rejection sampling, sometimes also called the *acceptance-rejection* method, is an alternative method for sampling random numbers from a specific probability distribution that may be used when the CDF cannot be inverted analytically. If a numerical constant value c can be found such that $f(x) \leq cg(x)$ for all x, then we can use $g(x)$ to obtain a sample from $f(x)$ and then claim that the value X has a PDF given by $f(x)$. The algorithm below demonstrates how this procedure may be implemented.

1. Generate a Y random variate having a PDF given by $g(Y)$.
2. Generate a random number from the uniform distribution $u[0,1]$.
3. If $u \leq \dfrac{f(Y)}{cg(Y)}$ then set $X = Y$ and stop, otherwise return to step 1.

Proof: We first define B to be the event that Y has been accepted in a loop: $u \leq \dfrac{f(Y)}{cg(Y)}$. The goal is to show that $P(X \leq x) = F_X(x)$. First observe that

$$P(X \leq x) = P(Y \leq x|B) = \frac{P((Y \leq x) \cap B)}{P(B)}$$

From our definition of event B and since $\lim_{x \to \infty} P(X \leq x) = 1$ we can then write

$$P(B) = P\left(u \leq \frac{f(Y)}{cg(Y)}\right) = \frac{1}{c}$$

The numerator $P((Y \leq x) \cap B)$ can be simplified further as shown below.

$$\begin{aligned}
P((Y \leq x) \cap B) &= \int_{-\infty}^{\infty} P((Y \leq x) \cap B | Y = y) g(y) dy \\
&= \int_{-\infty}^{\infty} P((Y \leq x) \cap \left(u \leq \frac{f(Y)}{cg(Y)}\right) |Y = y) g(y) dy \\
&= \int_{-\infty}^{x} P\left(u \leq \frac{f(y)}{cg(y)}\right) g(y) dy = \int_{-\infty}^{x} \frac{f(y)}{c} dy = \frac{F_X(x)}{c}
\end{aligned}$$

Therefore, $P(X \leq x) = F_X(x)$, as required.

Note that the efficiency of the rejection technique depends upon the function $g(x)$ and how closely it envelopes $f(x)$. If there is a large area between $cg(x)$ and $f(x)$ then a large percentage of random variates generated in steps 1 and 2 are wasted and rejected. Also, if the generation of random variates with $g(x)$ is particularly difficult then this method may not be very efficient. Although c should be chosen to be as small as possible, in many applications c is hard to compute and one often ends up choosing c conservatively large thereby resulting in very high computational costs.

Example: Consider generating a random variable having a PDF given by

$$f(x) = 20x(1-x)^3 \text{ where } 0 < x < 1$$

by applying the acceptance-rejection method with $g(x) = 1$ where $0 < x < 1$. To determine the constant c such that $f(x)/g(x) \leq c$, we use calculus to determine the maximum value of $f(x)/g(x) = 20x(1-x)^3$. Differentiation of this quantity yields

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = 20((1-x)^3 - 3x(1-x)^2)$$

Setting this quantity equal to 0 shows that the maximal value is attained when $x = 1/4$, and thus

$$\frac{f(x)}{g(x)} \leq 20\left(\frac{1}{4}\right)\left(\frac{3}{4}\right)^3 = \frac{135}{64} = c$$

Hence,

$$\left[\frac{f(x)}{cg(x)}\right] = \frac{256}{27}x(1-x)^3$$

which leads to the following acceptance-rejection algorithm

1. Generate two random numbers, u_1 and u_2 from the uniform distribution $u[0,1]$.
2. If $u_2 \leq \frac{256}{27}u_1(1-u_1)^3$ then set $X = u_1$ and stop. Otherwise return to step 1.

The acceptance-rejection method for discrete random variables can be deduced in an analogous fashion. Suppose we have an efficient method for generating a random variable having a PMF given by $\{g_i, i \geq 0\}$. We can use this PMF as the basis for generating random numbers from another more complex distribution having a PMF given by $\{f_i, i \geq 0\}$. Let c be a constant such that

$$\frac{f_i}{g_i} \leq c \text{ for all } i \text{ such that } f_i \neq 0.$$

Then the following algorithm can be used for generating a random variable X having a PMF given by $f_i = P(X = i)$.

1. Generate a Y random variate having a PMF given by g_i.

2. Generate a random number from the uniform distribution $u[0,1]$.

3. If $u \leq \dfrac{f_Y}{c\, g_Y}$ then set $X = Y$ and stop, otherwise return to step 1.

The proof of this algorithm is analogous to the one given earlier for the acceptance-rejection method for continuous random variables and is therefore left as an exercise for the reader.

The *composition sampling method* is another scheme for generating random deviates from a specific probability distribution. In this technique, the PDF or PMF is expressed as a probability mixture of properly selected density functions. Suppose that X has a CDF given by $F_X(x)$ and we wish to simulate a value of X. Often the CDF can be expressed as a weighted sum of n other CDFs as shown below.

$$F_X(x) = \sum_{j=1}^{n} p_j F_j(x)$$

where the individual $F_j(x)$ terms are also CDFs, $p_j \geq 0$ for all j, and $\sum_{j=1}^{n} p_j = 1$. Equivalently, if the density functions $f_X(x)$ exist, we can also write

$$f_X(x) = \sum_{j=1}^{n} p_j f_j(x)$$

The number of CDF functions n can be finite or infinite. The idea is that n CDFs can be composed together to form the desired CDF thus giving the name to this technique. Another approach is to have the desired CDF decomposed into several other CDFs. In that case, this method is called the decomposition sampling method.

To generate X variates using the composition method we

1. Generate I that is distributed on the non-negative integers so that $P(I = j) = p_j$.

2. If $I = j$, the simulate Y_j from F_j.

3. Set $X = Y_j$.

The claim is that X now has the desired distribution.

Proof:

$$\begin{aligned} P(X \leq x) &= \sum_{j=1}^{n} P(X \leq x | I = j) P(I = j) \\ &= \sum_{j=1}^{n} P(Y_j \leq x) P(I = j) \\ &= \sum_{j=1}^{n} F_j(x) p_j \\ &= F_X(x) \end{aligned}$$

The *convolution sampling* method provides another method for obtaining random deviates from a specific probability distribution. Suppose that X can be expressed as the sum of m random variables $y_1, y_2, \ldots, y_m$ so that

$$X = y_1 + y_2 + \ldots + y_m$$

where $y_i \sim F_i$ and are all independent. Then the following algorithm can be used for generating a random variable X.

1. Generate m random numbers $u_1, u_2, \ldots, u_m$ on the interval $[0, 1]$.
2. Apply the inverse transform method: $y_i = F_i^{-1} u_i$.
3. Set $X = \sum_{i=1}^{m} y_i$.

In this case, X can be generated by simply generating m random variates y_i and then summing them.

For example, if X is a sum of two random variables y_1 and y_2, then the densities of X can be calculated analytically by a convolution of the PDFs of y_1 and y_2. This is the reason why this method is called convolution sampling even though no convolution is actually required to generate the random variates.

Lastly, some distributions have special characteristics which allow their variates to be generated using algorithms specifically customized for them. Such algorithms are classified under the technique called *characterization sampling*. Examples of calculating random variates using this method tend to be rather unusual, bizarre and otherwise not at all obvious. For example, it can be shown that the ratio of two unit normal variates is a Cauchy $(0, 1)$ variate. It can also be shown that the ath smallest number in a sequence of $a + b + u(0, 1)$ uniform variates has a $beta(a, b)$ distribution.

After reviewing all these different methods for generating random variates from specific probability distributions, it is natural to ask which one is the best overall method to use. This is not a trivial question and there is no short simple answer. Instead, only general guidelines have been developed. If the CDF is easily invertible, then the inverse transformation method is the best choice. Otherwise, if either the CDF or PDF can be expressed as a sum of the other CDFs or PDFs, then the composition method can be used. If the variate can be expressed as a sum of other variates, use the convolution method. The characterization method can be used if the distribution has some known properties that can be exploited for random variate generation. Finally, if the PDF is such that a majorizing function can be found, then use the rejection method. If all else fails, you can always use empirical inverse transformation by numerically computing the distribution function.

10.6 Histograms

Working with probability distributions typically involves handling large amounts of data. As a result, it is often very helpful to see such data plotted on a graph in order to potentially identify underlying patterns or detect trends. However, plotting every available data point may not always be desirable or even feasible. Consequently, histograms were developed as a plotting tool to help simplify matters.

In a general sense, a histogram is an approximate graphical representation of the shape of a frequency distribution by means of rectangles whose widths represent class intervals and whose areas are proportional to the corresponding frequencies. A histogram is therefore just a graphical display of tabulated frequencies showing a count of the data points falling within various ranges. The groups of data are formally called classes, but in the context of a histogram they are also known as bins because one can think of them as containers that accumulate data and fill up at a rate equal to the frequency of that data class. In short, histograms are basically useful data summaries that convey the following information listed below.

- The general shape of the frequency distribution (normal, chi-square, etc.).
- Symmetry of the distribution and whether it is skewed.
- Modality - unimodal, bimodal, or multimodal.

The shape of the distribution conveys important information such as the probability distribution of the data. In cases in which the distribution is known, a histogram that does not fit the distribution may provide clues about a process and measurement problem. The shape of the histogram sometimes is particularly sensitive to the number of bins. If the bins are too wide, important information might get omitted. On the other hand, if the bins are too narrow, what may appear to be meaningful information really may be due to random variations in the data that show up because of the small number of data points in a bin. Different bin sizes can therefore reveal different features of the data. Because of the importance of this step, a considerable amount of research effort has been made to find a fast and reliable method for calculating appropriate bin sizes [62, 63].

Unfortunately no magic bullet-proof formula has yet been found for figuring out the *best* way to calculate the number of bins. The usual approach to determine whether the bin width is set to an appropriate size has been through the well established scientific method of trial-and-error. In other words, different bin widths should be tried and the results compared to determine the sensitivity of the histogram shape with respect to bin size. However, as a general rule-of-thumb, bin widths are typically selected so that there are between 5 and 20 groups of data, but the actual appropriate number depends on the situation. In addition, a mathematical formula has also evolved to help people get started in their search for the optimal bin width to use in calculating a histogram. If h represents the suggested bin width of a dataset X consisting of values given by $X = x_1, x_2, \ldots, x_N$ then the number of bins k can be roughly estimated by

$$k = \left\lceil \frac{\max X - \min X}{h} \right\rceil$$

where the braces indicate the ceiling function. Note that the histogram of the frequency distribution can be easily converted to a probability distribution by dividing the tally in each group by the total number of data points to give the relative frequency. The following C# code illustrates one way to calculate a basic frequency histogram of a random set of data points as indicated [64].

```
public static double[] makeHistogram(double[] data,double min,double
    max,int nBins)
{
  double[] Histogram = new double[nBins];
  double BinWidth = (max - min) / nBins;
  for (int i = 0; i < nBins; i++)
  {
    int nCounts = 0;
    for (int j = 0; j < data.Length; j++)
    {
if (data[j] >= min+(i)*BinWidth && data[j]<min+(i+1)*BinWidth)
      {
        nCounts++;
      }
    }
    Histogram[i] = nCounts;
  }
  return Histogram;
}

public static double[] makeHistogram(double[] data, int nBins)
{
  double[] Histogram = new double[nBins];
  for (int i = 0; i < nBins; i++)
  {
    int nCounts = 0;
    for (int j = 0; j < data.Length; j++)
    {
      if (data[j] == i)
      {
        nCounts++;
      }
    }
    Histogram[i] = nCounts;
  }
  return Histogram;
}

public static double dataMax(double[] data)
{
  double max = data[0];
  for (int i = 1; i < data.Length; i++)
  { max = Math.Max(max, data[i]); }
  return max;
}
public static double dataMin(double[] data)
{
  double min = data[0];
  for (int i = 1; i < data.Length; i++)
  { min = Math.Min(min, data[i]); }
  return min;
}
```

10.7 Random Variate Generation

After reviewing some of the most important fundamental concepts pertaining to the topic of this chapter, we are now ready to look at some examples of how to sample variates from the most common probability distributions. Since both the formulas and corresponding derivations that follow can be easily found in almost any introductory textbook, we will simply quote the results and then focus more of our attention on how to implement them in C#. For practical reasons the internal random number generator provided by C# was chosen as our default source for random numbers. If the quality of randomness is an important consideration, then one can easily upgrade to a more robust random number generator. Accordingly, one must first remember to instantiate an object from the `Random` number generator class as in the example shown below.

```
private static Random randObj = new Random();
```

Once an object of the random number class has been instantiated, one needs to replace its associated default `Next()` method with a customized version of it specifically designed to generate the desired probability distribution. In the examples that follow, the customized `Next()` method for each probability distribution has been overloaded in order to allow calculations for both single as well as multi-value array entries.

To evaluate the quality of the random variate generation algorithms for a particular probability distribution, a small standardized test routine was created capable of producing 10,000 random data points which are subsequently normalized on a scale from 0 to 100% and then spread into a histogram data structure consisting of 20 bins. The resulting output was then plotted directly onto the computer screen using stars, "*", to represent a certain number of data points contained in each bin. In order to enhance the evaluation of the resulting output even further, both the generated and the calculated probability distribution values are enclosed by brackets, `[...]`, and displayed on the screen for each bin.

10.7.1 Discrete Distributions

Bernoulli Distribution

The Bernoulli distribution, named after the Swiss mathematician Jacob Bernoulli, is perhaps the simplest discrete distribution. A Bernoulli trial is an experiment whose outcome is random and can be one of only two possible results, *success* or *failure*. A Bernoulli variate can therefore take only two values, which are usually denoted as $x = 0$ (failure) and $x = 1$ (success). The Bernoulli distribution can be used only if the trials are independent and identical so that the probability of success in each trial is given by p and is not affected or influenced by the outcomes of any past trials. The key characteristics of the Bernoulli distribution are summarized below [65, 66, 49].

$$\text{Parameters: } p = \text{probability of success in a trial, } 0 \le p \le 1$$

$$\text{PMF: } f(x) = \begin{cases} p^x(1-p)^{1-x} & \text{for } x = 0,1 \\ 0 & \text{otherwise} \end{cases}$$

Using the inverse transformation method, Bernoulli variates can be created by first generating $u \sim u(0,1)$. Then if $u \le p$, return 0 else return 1. This process is repeated until the desired number of Bernoulli variates has been obtained. An implementation of this procedure in C# is given below along with the resulting output.

```
public static double BernoulliPMF(int x, double p)
{
    if ((x == 0) || (x == 1))
        return (Math.Pow(p, x) * Math.Pow(1 - p, 1 - x));
    else
        return 0.0;
}

public static double[] BernoulliPMF(int[] x, double p)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    {
        tempArr[i] = BernoulliPMF(x[i], p);
    }
    return tempArr;
}

public static double NextBernoulli(double p)
{
    if (randObj.NextDouble() <= p)
        return 1.0;
    else
        return 0.0;
}

public static double[] NextBernoulli(int nLen, double p)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextBernoulli(p);
    }
    return tempArr;
}
```

```
static void TestBernoulli()
{
    int nBins = 20;
    int nPoints = 10000;
    double probSuccess = 0.60;
    double displayScaleFactor = 500.0;

    double[] randObj = NextBernoulli(nPoints, probSuccess);
    double[] HistogramValues = makeHistogram(randObj, nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];
    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = i;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i] = BernoulliPMF(i, probSuccess);
    }

    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);

    Console.WriteLine("\nTesting the Bernoulli Probability
        Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
         [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
        Console.Write("Bin {0,2:n0} = ", i + 1);
        for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
        Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                 ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                 yCalcDistributionMax,0));
    }
    Console.WriteLine("\nPress ENTER key to continue...");
    Console.ReadLine();
    Console.Clear();
}
```

```
Results: Bernoulli Probability Distribution (p = 0.60)
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 = *****************  [68] [67]
Bin  1 = *************************  [100] [100]
Bin  2 =   [0] [0]
Bin  3 =   [0] [0]
Bin  4 =   [0] [0]
Bin  5 =   [0] [0]
Bin  6 =   [0] [0]
.
.
.
Bin 18 =   [0] [0]
Bin 19 =   [0] [0]
```

Binomial Distribution

The binomial distribution is a discrete probability distribution that is used to model the number of successes in a sequence of n independent and identical Bernoulli trials assuming that each trial has probability p of success. Such an experiment is also called a Bernoulli experiment or Bernoulli trial. Actually, when $n = 1$ the binomial distribution is a Bernoulli distribution. The key characteristics of the binomial distribution are summarized below [65, 66, 49].

$$\begin{aligned} \text{Parameters: } & p = \text{probability of success in a trial, } 0 \leq p \leq 1 \\ & n = \text{number of trials, } n \geq 0 \end{aligned}$$

$$\text{PMF: } f(x) = \begin{cases} \dfrac{n!}{x!(n-x)!} p^x (1-p)^{n-x} & \text{for } x = 0, 1, 2, \ldots, n \\ 0 & \text{otherwise} \end{cases}$$

Binomial variates can be generated using a composition method that is based on the observation that the sum of n Bernoulli variates has a binomial distribution.

1. Generate n random numbers $u_1, u_2, \ldots, u_n$ on the interval $[0, 1]$.
2. Return the number of random numbers that are less than p.

The following code shown below illustrates how the binomial probability distribution may be implemented in C#. Note that the `Gamma` function is provided in Chapter 14, which focuses on the topic of special functions. The factorial $n!$ of an integer n is related to the $\Gamma()$ function by $n! = \Gamma(n+1)$.

```
public static double BinomialPMF(int x, int n, double p)
{
    return Gamma(n+1)*Math.Pow(p,x)*Math.Pow(1-p,n-x)/Gamma(x+1)/
        Gamma(n-x+1);
}

public static double[] BinomialPMF(int[] x, int n, double p)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    { tempArr[i] = BinomialPMF(x[i], n, p); }
    return tempArr;
}

public static double NextBinomial(int n, double p)
{
    double total = 0.0;
    for (int i = 0; i < n; i++)
    {
        if (randObj.NextDouble() < p)
        { total++; }
    }
    return total;
}
```

```
public static double[] NextBinomial(int n, double p, int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextBinomial(n, p);
    }
    return tempArr;
}

static void TestBinomial()
{
    int nBins = 20;
    int nTrials = 10;
    int nPoints = 10000;
    double probSuccess = 0.60;
    double displayScaleFactor = 500.0;

    double[] randObj = NextBinomial(nTrials, probSuccess, nPoints);
    double[] HistogramValues = makeHistogram(randObj, nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];
    double[] yCalcDistribution = new double[nBins];

    for (int i = 0; i < nTrials; i++)
    {
        xdata[i] = i;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i] = BinomialPMF(i, nTrials, probSuccess);
    }

    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);

    Console.WriteLine("\nTesting the Binomial Probability
        Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
         [calculated distrib.]\n");

    for (int i = 0; i < nBins; i++)
    {
        Console.Write("Bin {0,2:n0} = ", i + 1);
        for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
        Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                  ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                  yCalcDistributionMax,0));
    }
    Console.WriteLine("\nPress ENTER key to continue...");
    Console.ReadLine(); Console.Clear();
}
```

```
Results: Binomial Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 =   [0] [0]
Bin  1 =   [1] [1]
Bin  2 = *  [5] [4]
Bin  3 = ****  [17] [17]
Bin  4 = ***********  [45] [44]
Bin  5 = ********************  [80] [80]
Bin  6 = *************************  [100] [100]
Bin  7 = **********************  [88] [86]
Bin  8 = *************  [50] [48]
Bin  9 = ****  [17] [16]
Bin 10 =   [0] [0]
Bin 11 =   [0] [0]
Bin 12 =   [0] [0]
Bin 13 =   [0] [0]
Bin 14 =   [0] [0]
Bin 15 =   [0] [0]
Bin 16 =   [0] [0]
Bin 17 =   [0] [0]
Bin 18 =   [0] [0]
Bin 19 =   [0] [0]
```

Geometric Distribution

The geometric distribution is a discrete probability distribution obtained from the number of Bernoulli trials needed to get one success. In other words, suppose that a random experiment has two possible outcomes, success with probability p and failure with probability $q = 1 - p$. The experiment is repeated until a success happens. The number of trials before the success is a random variable x with density function given by the PMF of the geometric distribution whose key characteristics are summarized below [65, 66, 49].

$$\text{Parameters: } p = \text{probability of success in a trial, } 0 \le p \le 1$$

$$\text{PMF: } f(x) = \begin{cases} p\,(1-p)^{x-1} & \text{for } x = 1,2,3,\dots \\ 0 & \text{otherwise} \end{cases}$$

Equivalently, if the probability of success on each trial is p, then the probability that there are x failures before the first success is

$$\text{PMF: } f(x) = \begin{cases} p\,(1-p)^{x} & \text{for } x = 0,1,2,3,\dots \\ 0 & \text{otherwise} \end{cases}$$

Both of these distributions are referred to as a geometric probability distribution.

As derived earlier in this chapter, geometric variates can be generated using the inverse transform method which leads to the formula

$$X = \lceil \frac{\log(u)}{\log(1-p)} \rceil$$

Geometric variates can also be generated using the following alternate algorithm [65].

```
Initialize $X \leftarrow 0$,   $u \sim u[0,1]$
start loop: while $u < p$
$X \leftarrow X + 1$
$u \sim u[0,1]$
end loop: return $X$
```

The following code shown below illustrates how the binomial probability distribution may be implemented in C#.

```
public static double GeometricPMF(int x, double p)
{
    if (x >= 1)
        return (p * Math.Pow(1 - p, x - 1));
    else
        return 0.0;
}

public static double[] GeometricPMF(int[] x, double p)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    {
        tempArr[i] = GeometricPMF(x[i], p);
    }
    return tempArr;
}

public static double NextGeometric(double p)
{
 return Math.Ceiling(Math.Log(randObj.NextDouble())/Math.Log(1.0-p));
}

public static double NextGeometric2(double p)
{
    //Alternate method for calculating geometric random variates
    double rn = 1.0;
    for (; randObj.NextDouble() >= p; rn++) { }
    return rn;
}

public static double[] NextGeometric(int nLen, double p)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextGeometric(p);
    }
    return tempArr;
}
```

```
static void TestGeometric()
{
  int nBins = 20;
  int nPoints = 10000;
  double probSuccess = 0.60;
  double displayScaleFactor = 500.0;

  double[] randObj = NextGeometric(nPoints, probSuccess);
  double[] HistogramValues = makeHistogram(randObj, nBins);
  double[] xdata = new double[nBins];
  double[] ydata = new double[nBins];
  double[] yCalcDistribution = new double[nBins];
  for (int i = 0; i < nBins; i++)
  {
    xdata[i] = i;
    ydata[i] = (double)HistogramValues[i];
    yCalcDistribution[i] = GeometricPMF(i, probSuccess);
  }

  double ydataMax = dataMax(ydata);
  double yCalcDistributionMax = dataMax(yCalcDistribution);

  Console.WriteLine("\nTesting the Geometric Probability Distribution
      ");
  Console.WriteLine("\nKey: Bin number = *****... [random distrib.] [
      calculated distrib.]\n");
  for (int i = 0; i < nBins; i++)
  {
        Console.Write("Bin {0,2:n0} = ", i);
        for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
        nBins / ydataMax, 0)); j++) Console.Write("*");
        Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
            ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
            yCalcDistributionMax,0));
  }
  Console.WriteLine("\nPress ENTER key to continue...");
  Console.ReadLine(); Console.Clear();
}
```

```
Results: Geometric Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 =   [0] [0]
Bin  1 = *************************  [100] [100]
Bin  2 = **********  [41] [40]
Bin  3 = ****  [16] [16]
Bin  4 = **  [6] [6]
Bin  5 = *  [2] [3]
Bin  6 =   [1] [1]
Bin  7 =   [0] [0]
Bin  8 =   [0] [0]
.
.
.
Bin 18 =   [0] [0]
Bin 19 =   [0] [0]
```

Negative Binomial Distribution

The negative binomial distribution is a discrete probability distribution that arises in calculating the number of trials, x, needed to get a fixed non-random number of successes, r, in a Bernoulli process where each trial has a success probability p. Therefore, the PMF for this distribution calculates the probability that the r-th success occurs on the x-th trial. The key characteristics of the negative binomial distribution are summarized below [65, 66, 49].

Parameters: $p =$ probability of success, $0 \le p \le 1$

$r =$ integer number of successes, $r > 0$

$x =$ integer number of Bernoulli trials to obtain x successes, $x \ge 1$

$$\text{PMF: } f(x) = \begin{cases} \dfrac{\Gamma(r+x)}{\Gamma(r)\Gamma(x+1)} p^x (1-p)^r & \text{for } x = 0,1,2,3,\ldots \\ 0 & \text{otherwise} \end{cases}$$

Negative binomial random variates may be calculated using of the following algorithm [65]:

1. Generate y from a gamma distribution with parameters r and $p/(1-p)$.
2. Generate X from a Poisson distribution with parameter y. Now X is negative binomial with parameters p and r.

The following code shown below illustrates how the negative binomial probability distribution may be implemented in C#.

```
public static double NegativeBinomialPMF(int r, double p, int x)
{
    return (Gamma(r + x) / Gamma(r) / Gamma(x + 1)) * Math.Pow(p, r)
        * Math.Pow(1 - p, x);
}

public static double[] NegativeBinomialPMF(int r, double p, int[] x)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    {
        tempArr[i] = NegativeBinomialPMF(r, p, x[i]);
    }
    return tempArr;
}

public static double NextNegativeBinomial(int r, double p, int x)
{
    double y = NextGamma(r, (p / (1.0 - p)));
    return NextPoisson(y);
}
```

```
public static double[] NextNegativeBinomial(int r, double p, int x,
    int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextNegativeBinomial(r, p, x);
    }
    return tempArr;
}

static void TestNegativeBinomial()
{
    int nBins = 20;
    int nPoints = 10000;
    double pVal = 0.5;
    int rVal = 10;
    int xVal = 10;
    double displayScaleFactor = 500.0;

    double[] randObj = NextNegativeBinomial(rVal,pVal,xVal,nPoints);
    double[] HistogramValues = makeHistogram(randObj, nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];
    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = i;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i] = NegativeBinomialPMF(rVal,pVal,i);
    }

    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);

    Console.WriteLine("\nTesting the Negative Binomial Probability
        Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
         [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
       Console.Write("Bin {0,2:n0} = ", i + 1);
        for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
        Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                 ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                 yCalcDistributionMax,0));
    }
    Console.WriteLine("\nPress ENTER key to continue...");
    Console.ReadLine();
    Console.Clear();
}
```

```
Results: Negative Binomial Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 =   [1] [1]
Bin  1 = *  [4] [5]
Bin  2 = ***  [13] [14]
Bin  3 = *******  [28] [29]
Bin  4 = ************  [49] [47]
Bin  5 = *****************  [70] [66]
Bin  6 = *******************  [78] [82]
Bin  7 = *************************  [99] [94]
Bin  8 = *************************  [100] [100]
Bin  9 = *************************  [100] [100]
Bin 10 = **********************  [89] [95]
Bin 11 = *********************  [84] [86]
Bin 12 = *******************  [76] [76]
Bin 13 = ***************  [61] [64]
Bin 14 = **************  [57] [53]
Bin 15 = **********  [41] [42]
Bin 16 = ********  [34] [33]
Bin 17 = ******  [24] [25]
Bin 18 = ****  [16] [19]
Bin 19 = ***  [12] [14]
```

Poisson Distribution

The Poisson distribution is a discrete probability distribution that expresses the probability of a number of events occurring within a fixed period of time if these events occur with a known average rate and are independent of the elapsed time since the last event. The Poisson distribution can also be used in modeling the number of events in other specified intervals such as distance, area or volume. The key characteristics of the Poisson distribution are summarized below [65, 66, 49].

Parameters: $x =$ number of event occurrences, $x \geq 0$

$\lambda =$ mean number of event occurrences during the given interval, $\lambda > 0$

$$\text{PMF: } f(x) = \begin{cases} \dfrac{\lambda^x e^{-\lambda}}{x!} & \text{for } x = 0,1,2,3,\ldots \\ 0 & \text{otherwise} \end{cases}$$

Poisson random variates may be calculated in a number of different ways [65, 66, 49]. For example, by using the Poisson PMF

$$P(X = x_i) = p_i = \frac{\lambda^i e^{-\lambda}}{i!}$$

we can easily derive a recursive relation

$$p_{i+1} = e^{-\lambda} \frac{\lambda^{i+1}}{(i+1)!} = \frac{\lambda}{i+1} e^{-\lambda} \frac{\lambda^i}{i!} = \frac{\lambda}{i+1} p_i$$

which leads to the following algorithm:

```
Initialize: $i \leftarrow 0$, $w \leftarrow e^{-\lambda}$, $p_0 \leftarrow w$ and $u \sim u[0,1]$
start loop: while $(w \leq u)$
$p_{i+1} \leftarrow \frac{\lambda}{i+1} p_i$
$w \leftarrow w + v$
$i \leftarrow i + 1$
end loop: return $i$.
```

A second algorithm, due to Knuth [67], is outlined below.

```
Initialize: Let $L \leftarrow e^{-\lambda}$, $k \leftarrow 0$, $p \leftarrow 1$
do: $k \leftarrow k + 1$
Generate uniform random number $u \sim [0,1]$ and let $p \leftarrow p \times u$
while $p > L$
return k-1.
```

A third method, proposed by Kemp [68], is given below.

```
Initialize: Let $w \leftarrow e^{-\lambda}$, $i \leftarrow 0$, $u \sim u[0,1]$
start loop: while $(u > w)$
$u \leftarrow u - w$
$i \leftarrow i + 1$
$w \leftarrow w * \lambda / i$
end loop: return $i$
```

The following code illustrates how the Poisson probability distribution along with these three algorithms for generating Poisson random variates may be implemented in C#.

```
public static double PoissonPMF(int x, double lambda)
{
    return Math.Exp(-lambda) * Math.Pow(lambda, x) / Gamma(x + 1);
}


public static double[] PoissonPMF(int[] x, double lambda)
{
    double[] tempArray = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    {
        tempArray[i] = PoissonPMF(x[i], lambda);
    }
    return tempArray;
}
```

```
public static double NextPoisson(double lambda)
{
    //Using the recursive algorithm
    int i = 0;
    double w = Math.Exp(-lambda);
    double v = w;
    double u = randObj.NextDouble();
    while (w <= u)
    {
        v = v * (lambda / (double)(i + 1));
        w = w + v;
        i = i + 1;
    }
    return i;
}

public static double NextPoisson2(double lambda)
{
    //Using algorithm proposed by Knuth
    //see http://en.wikipedia.org/wiki/Poisson_distribution
    int k = 0;
    double p = 1.0;
    double L = Math.Exp(-lambda);
    do
    {
        k++;
        p *= randObj.NextDouble();
    } while (p >= L);
    return k - 1;
}

public static double NextPoisson3(double lambda)
{
    //Using algorithm proposed by Kemp (1981)
    int i = 0;
    double w = Math.Exp(-lambda);
    double u = randObj.NextDouble();
    while (u > w)
    {
        u = u - w;
        i = i + 1;
        w = w * lambda / i;
    }
    return i;
}

public static double[] NextPoisson(double lambda, int nLength)
{
    double[] tempArr = new double[nLength];
    for (int i = 0; i < nLength; i++)
    {
        tempArr[i] = NextPoisson(lambda);
    }
    return tempArr;
}
```

```
static void TestPoisson()
{
    int nBins = 20; int nPoints = 10000;
    double lambdaValue = 4.0; double displayScaleFactor = 500.0;
    double[] randObj = NextPoisson(lambdaValue, nPoints);
    double[] HistogramValues = makeHistogram(randObj, nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];
    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = i;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i] = PoissonPMF(i, lambdaValue);
    }
    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);
    Console.WriteLine("\nTesting the Poisson Probability
                        Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
         [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
        Console.Write("Bin {0,2:n0} = ", i + 1);
        for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
        Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                  ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                  yCalcDistributionMax,0));
    }
    Console.ReadLine(); Console.Clear();
}
```

```
Results: Poisson Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 = **  [10] [9]
Bin  1 = *********  [37] [37]
Bin  2 = ******************  [76] [75]
Bin  3 = ************************  [99] [100]
Bin  4 = ************************  [100] [100]
Bin  5 = *******************  [80] [80]
Bin  6 = *************  [53] [53]
Bin  7 = *******  [30] [30]
Bin  8 = ****  [15] [15]
Bin  9 = **  [6] [7]
Bin 10 = *  [3] [3]
Bin 11 =   [1] [1]
Bin 12 =   [0] [0]
Bin 13 =   [0] [0]
Bin 14 =   [0] [0]
Bin 15 =   [0] [0]
Bin 16 =   [0] [0]
Bin 17 =   [0] [0]
Bin 18 =   [0] [0]
Bin 19 =   [0] [0]
```

Uniform Distribution (discrete)

The discrete uniform distribution is a discrete probability distribution that can be characterized by saying that all values of a finite set of possible values are equally probable. If a random variable has any one of n possible values $x_1, x_2, \ldots, x_n$ that are equally probable, then it is a discrete uniform distribution with the probability of any outcome $P(x_i) = 1/n$. The key characteristics of the discrete uniform distribution are summarized below [65, 66, 49].

$$\begin{aligned}\text{Parameters: } a &= \text{lower limit; } a > 0\\ b &= \text{upper limit; } b > a\\ \text{PMF: } f(x) &= \begin{cases}\dfrac{1}{b-a+1} & \text{if } a \le x \le b\\ 0 & \text{otherwise}\end{cases}\end{aligned}$$

The following code illustrates how the discrete uniform probability distribution may be implemented in C#.

```
public static double UniformDiscretePMF(double x, double a,
        double b)
{
    if ((x >= a) && (x < b))
        return (1.0 / (b - a + 1));
    else
        return 0.0;
}

public static double[] UniformDiscretePMF(double[] x, double a,
        double b)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    { tempArr[i] = UniformDiscretePMF(x[i], a, b); }
    return tempArr;
}

public static double NextUniformDiscrete(double a, double b)
{
    return Math.Floor((a + (b - a + 1) * randObj.NextDouble()));
}

public static double[] NextUniformDiscrete(int nLen, double a,
        double b)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextUniformDiscrete(a, b);
    }
    return tempArr;
}
```

```
static void TestUniformDiscrete()
{
    int nBins = 20; int nPoints = 10000; double xmin = 0.0;
    double xmax = 20.0; double a = 5.0; double b = 15.0;
    double displayScaleFactor = 500.0;
    double[] randObj=NextUniformDiscrete(nPoints,a,b);
    double[] HistogramValues=makeHistogram(randObj,xmin,xmax,nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];
    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = i;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i] = UniformDiscretePMF(i, a, b);
    }
    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);
    Console.WriteLine("\nTesting the Uniform (Discrete) Probability
        Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
         [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
        Console.Write("Bin {0,2:n0} = ", i + 1);
        for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
        Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                  ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                  yCalcDistributionMax,0));
    }
}
```

```
Results: Uniform (Discrete) Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 =   [0] [0]
Bin  1 =   [0] [0]
Bin  2 =   [0] [0]
Bin  3 =   [0] [0]
Bin  4 =   [0] [0]
Bin  5 = ************************  [96] [100]
Bin  6 = *************************  [99] [100]
Bin  7 = ************************  [97] [100]
Bin  8 = ************************  [98] [100]
Bin  9 = *************************  [100] [100]
Bin 10 = ************************  [96] [100]
Bin 11 = ***********************  [91] [100]
Bin 12 = ************************  [95] [100]
Bin 13 = ***********************  [93] [100]
Bin 14 = ***********************  [93] [100]
Bin 15 = ************************  [97] [0]
Bin 16 =   [0] [0]
Bin 17 =   [0] [0]
Bin 18 =   [0] [0]
Bin 19 =   [0] [0]
```

10.7.2 Continuous Distributions

Beta Distribution

Beta distributions are used extensively in Bayesian statistics. Bayesian inference is a statistical inference in which evidence or observations are used to update or to infer the probability that a hypothesis may be true. The name "Bayesian" comes from the frequent use of Bayes' theorem in the inference process. The beta distribution can also be used to model events which are constrained to take place within an interval defined by a minimum and maximum value. For this reason, the beta distribution is used extensively in project management control systems to describe and model the time for the completion of a task.

The beta distribution is a family of continuous probability distributions defined on the interval $[0,1]$ and parameterized by two positive shape parameters, typically denoted by a and b. The formulas derived for the default interval $[0,1]$ can be easily extended to a more general interval $[x_{min}, x_{max}]$ by making the following substitution

$$\frac{x - x_{min}}{x_{max} - x_{min}}$$

for x. The key characteristics of the beta distribution are summarized below [65].

Parameters: $a > 0,\ b > 0$

$$\text{PDF: } f(x) = \begin{cases} \dfrac{x^{a-1}(1-x)^{b-1}}{\beta(a,b)} & \text{if } 0 \le x \le 1 \\ 0 & \text{otherwise} \end{cases}$$

where the $\beta(a,b) = \int_0^1 x^{a-1}(1-x)^{b-1}dx = \dfrac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$.

Beta distribution random variates can be generated in number of ways [66]. For example, it can be shown that one beta random variable can be derived from two gamma random variables. Suppose that G_1 and G_2 are independently distributed with $G_1 = G(a,1)$ and $G_2 = G(b,1)$. Then $X = G_1/(G_1+G_2)$ has a beta distribution with parameters a and b. The following code illustrates how the beta probability density distribution may be implemented in C#.

```
public static double BetaPDF(double x, double a, double b)
{
    return Math.Pow(x, a - 1) * Math.Pow(1 - x, b - 1) / Beta(a, b);
}

public static double[] BetaPDF(double[] x, double a, double b)
{
    double[] tempArray = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    { tempArray[i] = BetaPDF(x[i], a, b); }
    return tempArray;
}
```

```
public static double NextBeta(int a, int b)
{
    double gamma1 = NextGamma(a, 1);
    double gamma2 = NextGamma(b, 1);
    return gamma1 / (gamma1 + gamma2);
}

public static double[] NextBeta(int a, int b, int nLength)
{
    double[] tempArr = new double[nLength];
    for (int i = 0; i < nLength; i++)
    { tempArr[i] = NextBeta(a, b); }
    return tempArr;
}

static void TestBeta()
{
    int nBins = 20;
    int nPoints = 10000;
    double xmin = 0.0;
    double xmax = 1.0;
    double aVal = 2.0;
    double bVal = 5.0;
    double displayScaleFactor = 500.0;

    double[] randObj=NextBeta((int)aVal,(int)bVal,nPoints);
    double[] HistogramValues=makeHistogram(randObj,xmin,xmax,nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];
    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = xmin + (i + 0.5) * (xmax - xmin) / nBins;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i] = BetaPDF(xdata[i], aVal, bVal);
    }

    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);

    Console.WriteLine("\nTesting the Beta Probability Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
         [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
       Console.Write("Bin {0,2:n0} = ", i + 1);
        for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
               nBins / ydataMax, 0)); j++) Console.Write("*");
        Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                 ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                 yCalcDistributionMax,0));
    }
    Console.WriteLine("\nPress ENTER key to continue...");
    Console.ReadLine(); Console.Clear();
}
```

```
Results: Beta Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 = ******  [25] [28]
Bin  1 = *****************  [67] [68]
Bin  2 = ***********************  [92] [90]
Bin  3 = *************************  [99] [100]
Bin  4 = *************************  [100] [100]
Bin  5 = **********************  [89] [94]
Bin  6 = ********************  [82] [83]
Bin  7 = *****************  [67] [70]
Bin  8 = **************  [57] [57]
Bin  9 = ***********  [42] [44]
Bin 10 = *******  [29] [33]
Bin 11 = ******  [25] [23]
Bin 12 = ****  [17] [15]
Bin 13 = ***  [11] [9]
Bin 14 = *  [6] [5]
Bin 15 =   [2] [2]
Bin 16 =   [1] [1]
Bin 17 =   [0] [0]
Bin 18 =   [0] [0]
Bin 19 =   [0] [0]
```

Beta Prime Distribution

A beta prime distribution is a probability distribution defined for $x > 0$ with two parameters, a and b, having the following characteristics [65, 66, 49]:

$$\text{Parameters: } a > 0,\ b > 0$$

$$\text{PDF: } f(x) = \begin{cases} \dfrac{x^{a-1}\,(1+x)^{-a-b}}{\beta(a,b)} & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

This distribution is also known as the beta distribution of the second kind, in contrast to the related beta distribution. Random variates following a beta prime distribution can be generated very easily from their corresponding variates of the beta distribution: $X_{\text{beta prime}} = X_{\text{beta}}/(1 - X_{\text{beta}})$. The following code illustrates how the beta prime probability density distribution may be implemented in C#.

```
public static double BetaPrimePDF(double x,double a,double b)
{
    return Math.Pow(x,a-1) * Math.Pow(1+x,-a-b) / Beta(a,b);
}
public static double[] BetaPrimePDF(double[] x,double a,double b)
{
    double[] tempArray = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    { tempArray[i] = BetaPDF(x[i], a, b); }
    return tempArray;
}
```

```
public static double NextBetaPrime(int a, int b)
{
   double betaVariate = NextBeta(a, b);
   return betaVariate / (1.0 - betaVariate);
}

public static double[] NextBetaPrime(int a,int b,int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextBetaPrime(a, b);
    }
    return tempArr;
}

static void TestBetaPrime()
{
    int nBins = 20;
    int nPoints = 10000;
    double xmin = 0.0; double xmax = 1.0;
    int aVal = 3; int bVal = 6;
    double displayScaleFactor = 500.0;
    double[] randObj = NextBetaPrime(aVal, bVal, nPoints);
    double[] HistogramValues=makeHistogram(randObj,xmin,xmax,nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];
    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = xmin + (i+0.5) * (xmax-xmin)/nBins;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i] = BetaPrimePDF(xdata[i],aVal,bVal);
    }
    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);
    Console.WriteLine("\nTesting the Beta Prime Probability
        Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
         [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
        Console.Write("Bin {0,2:n0} = ", i + 1);
        for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
        Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                  ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                  yCalcDistributionMax,0));
    }
    Console.WriteLine("\nPress ENTER key to continue...");
    Console.ReadLine();
    Console.Clear();
}
```

```
Results: Beta Prime Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 = **  [8] [6]
Bin  1 = *********  [37] [35]
Bin  2 = **************  [55] [64]
Bin  3 = *********************  [83] [84]
Bin  4 = **********************  [87] [96]
Bin  5 = *************************  [100] [100]
Bin  6 = **********************  [88] [99]
Bin  7 = **********************  [87] [94]
Bin  8 = ********************  [81] [88]
Bin  9 = ******************  [73] [80]
Bin 10 = *******************  [75] [73]
Bin 11 = ***************  [58] [65]
Bin 12 = **************  [57] [58]
Bin 13 = ************  [46] [52]
Bin 14 = **********  [42] [46]
Bin 15 = **********  [40] [40]
Bin 16 = *********  [37] [36]
Bin 17 = ********  [30] [31]
Bin 18 = ******  [25] [28]
Bin 19 = ******  [24] [24]
```

Cauchy Distribution

The Cauchy distribution, also referred to as the Lorentz distribution, the Breit-Wigner distribution, or the Lorentzian function, is a continuous probability distribution most closely associated with the study of resonance behavior. In physics, resonance is the tendency of a system to oscillate at a larger amplitude at certain frequencies than at others. These are known as the system's resonance frequencies. At resonance frequencies, even small periodic driving forces can produce large amplitude vibrations. Resonance phenomena has been observed to occur with all types of vibrations or waves. For example, there is mechanical resonance, acoustic resonance, electromagnetic resonance, nuclear magnetic resonance (NMR), electron spin resonance (ESR) and resonance of quantum wave functions. Resonant systems can be used to generate vibrations of a specific frequency, such as in the case of musical instruments, or pick out specific frequencies from a complex vibration containing many frequencies. The key characteristics of the Cauchy distribution are summarized below [65, 66, 49].

$$\text{Parameters: } a > 0,\ b > 0$$

$$\text{Range: } -\infty < x < \infty$$

$$\text{PDF: } f(x) = \frac{1}{\pi b\left[1 + \left(\frac{x-a}{b}\right)^2\right]} = \frac{b}{\pi[b^2 + (x-a)^2]}$$

where a is the scale parameter that specifies the location of the resonance peak of the distribution and b specifies the half width at the half maximum.

Random variates from a Cauchy distribution can be generated using the inversion transformation method and are given by the following formula [65, 66, 49]:

$$X = a + b[\tan(\pi u[0,1] - 0.5)]$$

where $u[0,1]$ is a random variate drawn from the uniform distribution. The following code illustrates how the Cauchy distribution may be implemented in C#.

```
public static double CauchyPDF(double x, double a, double b)
{
    return b / (Math.PI * (b * b + (x - a) * (x - a)));
}

public static double[] CauchyPDF(double[] x, double a, double b)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    { tempArr[i] = CauchyPDF(x[i], a, b); }
    return tempArr;
}

public static double NextCauchy(double a, double b)
{
    return a + b * (Math.Tan(Math.PI * randObj.NextDouble() - 0.5));
}

public static double[] NextCauchy(double a, double b, int nLength)
{
    double[] tempArr = new double[nLength];
    for (int i = 0; i < nLength; i++)
    { tempArr[i] = NextCauchy(a, b); }
    return tempArr;
}

static void TestCauchy()
{
  int nBins = 20;
  int nPoints = 10000;
  double xmin = -5;
  double xmax = 5;
  double aVal = 0.0;
  double bVal = 0.5;
  double displayScaleFactor = 500.0;
  double[] randObj = NextCauchy(aVal, bVal, nPoints);
  double[] HistogramValues=makeHistogram(randObj,xmin,xmax,nBins);
  double[] xdata = new double[nBins];
  double[] ydata = new double[nBins];
  double[] yCalcDistribution = new double[nBins];
  for (int i = 0; i < nBins; i++)
  {
    xdata[i] = xmin + (i + 0.5) * (xmax - xmin) / nBins;
    ydata[i] = HistogramValues[i];
    yCalcDistribution[i] = CauchyPDF(xdata[i], aVal, bVal);
  }
  double ydataMax = dataMax(ydata);
  double yCalcDistributionMax = dataMax(yCalcDistribution);
  Console.WriteLine("\nTesting the Cauchy Probability Distribution");
  Console.WriteLine("\nKey: Bin number = *****... [random distrib.] [
      calculated distrib.]\n");
  for (int i = 0; i < nBins; i++)
  {
    Console.Write("Bin {0,2:n0} = ", i + 1);
    for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
         nBins / ydataMax, 0)); j++) Console.Write("*");
    Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
              ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
              yCalcDistributionMax,0));
  }
  Console.WriteLine("\nPress ENTER key to continue...");
  Console.ReadLine(); Console.Clear();
}
```

```
Results: Cauchy Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 =   [1] [1]
Bin  1 =   [2] [2]
Bin  2 = *  [3] [2]
Bin  3 = *  [3] [3]
Bin  4 = *  [4] [4]
Bin  5 = *  [6] [6]
Bin  6 = **  [9] [9]
Bin  7 = ****  [18] [17]
Bin  8 = **********  [39] [38]
Bin  9 = *************************  [99] [100]
Bin 10 = *************************  [100] [100]
Bin 11 = **********  [39] [38]
Bin 12 = ****  [18] [17]
Bin 13 = ***  [11] [9]
Bin 14 = **  [7] [6]
Bin 15 = *  [4] [4]
Bin 16 = *  [3] [3]
Bin 17 =   [2] [2]
Bin 18 =   [1] [2]
Bin 19 =   [1] [1]
```

Chi Distribution

If X_i are n independent, normally distributed, random variables with means μ_i and standard deviations σ_i, then the statistic

$$Z = \sqrt{\sum_{i=1}^{k} \left(\frac{X_i - \mu_i}{\sigma_i} \right)^2}$$

is distributed according to the χ distribution with n degrees of freedom (i.e. the number of X_i). The most familiar example is perhaps the Maxwell distribution of normalized molecular speeds which is a χ distribution with 3 degrees of freedom.

The key characteristics of the χ distribution are summarized below [65, 66, 49].

Parameters: $n =$ degrees of freedom, n must be a positive integer

$$\text{PDF: } f(x) = \begin{cases} \dfrac{2\left(\dfrac{n}{2}\right)^{n/2} x^{n-1} \exp\left[-\left(\dfrac{n}{2\sigma^2}x^2\right)\right]}{\Gamma\left(\dfrac{n}{2}\right)\sigma^n} & \text{for } 0 \le x < \infty \\ 0 & \text{otherwise} \end{cases}$$

Random variates from a χ distribution can be generated from the inverted CDF of the distribution and are given by the following formula [65, 66, 49]:

$$\chi(n,\sigma) = \sqrt{\frac{\chi^2(n,\sigma)}{n}}$$

The following code illustrates how the χ distribution may be implemented in C#.

```
public static double ChiPDF(double x, int n)
{
    double gamma = Gamma(n / 2.0);
    double exp = Math.Exp(-n * x * x / 2.0);
    return 2.0 * Math.Pow(n / 2.0, n / 2) * Math.Pow(x, n - 1) * exp
        / Math.Pow(2, n / 2) / gamma;
}

public static double[] ChiPDF(double[] x, int n)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    {
        tempArr[i] = ChiPDF(x[i], n);
    }
    return tempArr;
}

public static double ChiPDF(double x, int n, double sigma)
{
    double gamma = Gamma(n / 2.0);
    double exp = Math.Exp(-n * x * x / 2.0 / sigma / sigma);
    return 2.0 * Math.Pow(n / 2.0, n / 2) * Math.Pow(x, n - 1) * exp
        / Math.Pow(2, n / 2) / gamma / Math.Pow(sigma, n);
}

public static double[] ChiPDF(double[] x, int n, double sigma)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    { tempArr[i] = ChiPDF(x[i], n, sigma); }
    return tempArr;
}
```

```
public static double NextChi(int n)
{
    return Math.Sqrt(NextChiSquare(n) / n);
}

public static double[] NextChi(int n, int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextChi(n);
    }
    return tempArr;
}

public static double NextChi(int n, double sigma)
{
    return Math.Sqrt(NextChiSquare(n, sigma) / n);
}

public static double[] NextChi(int n, double sigma, int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    { tempArr[i] = NextChi(n, sigma); }
    return tempArr;
}
```

```
Note: The driver routine for the chi distribution is listed
together with the chi-square distribution.

Results: Chi Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 = ********  [30] [29]
Bin  1 = *******************  [75] [77]
Bin  2 = *************************  [100] [100]
Bin  3 = ************************  [98] [96]
Bin  4 = ******************  [73] [75]
Bin  5 = *************  [50] [49]
Bin  6 = *******  [28] [27]
Bin  7 = ***  [13] [13]
Bin  8 = *  [6] [5]
Bin  9 =   [1] [2]
Bin 10 =   [1] [1]
Bin 11 =   [0] [0]
Bin 12 =   [0] [0]
Bin 13 =   [0] [0]
Bin 14 =   [0] [0]
Bin 15 =   [0] [0]
Bin 16 =   [0] [0]
Bin 17 =   [0] [0]
Bin 18 =   [0] [0]
Bin 19 =   [0] [0]
```

Chi-Square Distribution

The χ^2 distribution is one of the most widely used theoretical probability distributions in inferential statistics. It is useful because under reasonable assumptions, easily calculated quantities can be proven to have distributions that approximate to the χ^2 distribution if the null hypothesis is true. The null hypothesis describes in a formal way some aspect of the statistical behavior of a set of data and this description is treated as valid unless the actual behavior of the data contradicts this assumption. The key characteristics of the χ^2 distribution are summarized below [65, 66, 49].

Parameters: $n =$ degrees of freedom, n must be a positive integer

$$\text{PDF: } f(x) = \begin{cases} \dfrac{x^{n/2-1} \exp\left[-\left(\dfrac{n}{2\sigma^2} x^2\right)\right]}{2^{n/2}\Gamma\left(\dfrac{n}{2}\right)\sigma^n} & \text{for } 0 \le x < \infty \\ 0 & \text{otherwise} \end{cases}$$

Random variates from a χ^2 distribution can be generated from the inverted CDF of the distribution and are given by the following formula [65, 66, 49]:

$$\chi^2(n,\sigma) = \sum_{i=1}^{n} \left[N_i(0,\sigma^2)\right]^2$$

where $N_i(0,\sigma^2)$ is the normal random distribution. The following code illustrates how the χ^2 distribution may be implemented in C#.

```
public static double ChiSquarePDF(double x, int n)
{
    double gamma = Gamma(n / 2.0);
    double exp = Math.Exp(-x / 2.0);
    return Math.Pow(x, n / 2 - 1) * exp / Math.Pow(2, n / 2) / gamma;
}

public static double[] ChiSquarePDF(double[] x, int n)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    {
        tempArr[i] = ChiSquarePDF(x[i], n);
    }
    return tempArr;
}

public static double ChiSquarePDF(double x, int n, double sigma)
{
    double gamma = Gamma(n / 2.0);
    double exp = Math.Exp(-x / 2.0 / sigma / sigma);
    return Math.Pow(x, n / 2 - 1) * exp / Math.Pow(2, n / 2) / gamma
        / Math.Pow(sigma, n);
}
```

```
public static double[] ChiSquarePDF(double[] x, int n, double sigma)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    {
        tempArr[i] = ChiSquarePDF(x[i], n, sigma);
    }
    return tempArr;
}

public static double NextChiSquare(int n)
{
    double sum = 0.0;
    for (int i = 0; i < n; i++)
    {
        sum += Math.Pow(NextNormal(0, 1), 2);
    }
    return sum;
}

public static double[] NextChiSquare(int n, int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextChiSquare(n);
    }
    return tempArr;
}

public static double NextChiSquare(int n, double sigma)
{
    double sum = 0.0;
    for (int i = 0; i < n; i++)
    {
        sum += Math.Pow(NextNormal(0, sigma * sigma), 2);
    }
    return sum;
}

public static double[] NextChiSquare(int n, double sigma, int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextChiSquare(n, sigma);
    }
    return tempArr;
}

static void TestChiChiSquare()
{
  int nBins = 20;
  int nPoints = 10000;
  double xmin = 0;
  double xmax = 5;
  double sigma = 1.0;
  int nVal = 2;
  double displayScaleFactor = 500.0;

  double[] randChi = NextChi(nVal, sigma, nPoints);
  double[] randChiSquare = NextChiSquare(nVal, sigma, nPoints);
  double[] RandomDistributionValuesChi =
                makeHistogram(randChi, xmin, xmax, nBins);
  double[] RandomDistributionValuesChiSquare =
                makeHistogram(randChiSquare, xmin, xmax, nBins);
  double[] xdata = new double[nBins];
  double[] ydataChi = new double[nBins];
  double[] ydataChiSquare = new double[nBins];
  double[] yChiCalcDistribution = new double[nBins];
  double[] yChiSquareCalcDistribution = new double[nBins];

  for (int i = 0; i < nBins; i++)
  {
    xdata[i] = xmin + (i+0.5) * (xmax-xmin) / nBins;
    ydataChi[i]=(double)RandomDistributionValuesChi[i];
    ydataChiSquare[i]=(double)RandomDistributionValuesChiSquare[i];
    yChiCalcDistribution[i]=ChiPDF(xdata[i], nVal, sigma);
    yChiSquareCalcDistribution[i]=ChiSquarePDF(xdata[i],nVal,sigma);
  }

  double ydataChiMax = dataMax(ydataChi);
  double yChiCalcDistributionMax = dataMax(yChiCalcDistribution);

  double ydataChiSquareMax = dataMax(ydataChiSquare);
  double yChiSquareCalcDistributionMax =
        dataMax(yChiSquareCalcDistribution);

  Console.WriteLine("\nTesting the Chi Probability Distribution");
  Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
        [calculated distrib.]\n");
  for (int i = 0; i < nBins; i++)
  {
    Console.Write("Bin {0,2:n0} = ", i + 1);
    for (int j = 0; j < (Math.Round(ydataChi[i] *
        displayScaleFactor/nBins/ydataChiMax,0));j++)
                Console.Write("*");
    Console.WriteLine("  [{0}] [{1}]", Math.Round(ydataChi[i] *
        100.0 / ydataChiMax, 0), Math.Round(yChiCalcDistribution[i] *
        100.0 / yChiCalcDistributionMax, 0));
  }

  Console.WriteLine("\nTesting the Chi-Square Probability
      Distribution");
  Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
        [calculated distrib.]\n");
  for (int i = 0; i < nBins; i++)
  {
    Console.Write("Bin {0,2:n0} = ", i + 1);
    for (int j = 0; j < (Math.Round(ydataChiSquare[i] *
        displayScaleFactor/nBins/ydataChiSquareMax,0));j++)
```

```
                Console.Write("*");
        Console.WriteLine("  [{0}] [{1}]", Math.Round(ydataChiSquare[i] *
            100.0 / ydataChiSquareMax, 0),
            Math.Round(yChiSquareCalcDistribution[i] *
            100.0 / yChiSquareCalcDistributionMax, 0));
    }
    Console.WriteLine("\nPress ENTER key to continue...");
    Console.ReadLine(); Console.Clear();
}
```

```
Results: Chi Square Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 = *************************  [100] [100]
Bin  1 = **********************  [87] [88]
Bin  2 = *******************  [76] [78]
Bin  3 = *****************  [67] [69]
Bin  4 = ***************  [58] [61]
Bin  5 = *************  [52] [54]
Bin  6 = ************  [47] [47]
Bin  7 = **********  [39] [42]
Bin  8 = **********  [39] [37]
Bin  9 = ********  [31] [32]
Bin 10 = *******  [29] [29]
Bin 11 = ******  [23] [25]
Bin 12 = *****  [21] [22]
Bin 13 = ****  [18] [20]
Bin 14 = *****  [18] [17]
Bin 15 = ****  [15] [15]
Bin 16 = ****  [15] [14]
Bin 17 = ***  [12] [12]
Bin 18 = **  [10] [11]
Bin 19 = **  [9] [9]
```

Erlang Distribution

The Erlang distribution is a special case of the gamma distribution where the shape parameter is a positive integer. It represents the sum of a series of exponential distributions. The distribution resulted from work done by the Danish mathematician Agner Erlang who was a pioneer in the application of statistical methods to the analysis of telephone networks. The distribution was derived to model the total waiting time associated with a queue of requests on a telephone exchange. Today, the Erlang distribution is commonly used in queueing models as an extension of the exponential distribution. The key characteristics of the Erlang distribution are summarized below [65, 66, 49].

Parameters: $a =$ scale parameter, $a > 0$

$m =$ shape parameter, m is a positive integer

$$\text{PDF: } f(x) = \begin{cases} \dfrac{x^{m-1}e^{-x/a}}{(m-1)!a^m} & \text{where } 0 \le x < \infty \\ 0 & \text{otherwise} \end{cases}$$

Erlang distribution random variates can be generated using the following equation.

$$X = -a \prod_{i=1}^{m} u_i$$

The code below illustrates how the Erlang distribution may be implemented in C#.

```
public static double ErlangPDF(double x, double a, int b)
{
  if (x >= 0.0)
    return (Math.Pow(x,b-1)*Math.Exp(-x/a))/(Gamma(b)*Math.Pow(a,b));
  else
    return 0.0;
}

public static double[] ErlangPDF(double[] x, double a, int b)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    {
        tempArr[i] = ErlangPDF(x[i], a, b);
    }
    return tempArr;
}

public static double NextErlang(double a, int b)
{
    double product = 1.0;
    for (int i = 0; i < b; i++)
    {
        product *= randObj.NextDouble();
    }
    return (-a) * Math.Log(product);
}

public static double[] NextErlang(double a, int b, int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextErlang(a,b);
    }
    return tempArr;
}

static void TestErlang()
{
    int nBins = 20;
    int nPoints = 10000;
    double xmin = 0;
    double xmax = 4;
    double aVal = 1.0;
    int bVal = 1;
    double displayScaleFactor = 500.0;

    double[] randObj = NextErlang(aVal, bVal, nPoints);
    double[] HistogramValues=makeHistogram(randObj,xmin,xmax,nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];

    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = xmin + (i + 0.5) * (xmax - xmin) / nBins;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i] = ErlangPDF(xdata[i], aVal, bVal);
    }
    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);
    Console.WriteLine("\nTesting the Erlang Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
        [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
        Console.Write("Bin {0,2:n0} = ", i + 1);
        for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
        Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                  ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                  yCalcDistributionMax,0));
    }
    Console.WriteLine("\nPress ENTER key to continue...");
    Console.ReadLine(); Console.Clear();
}
```

```
Results: Erlang Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 = ************************  [100] [100]
Bin  1 = *********************  [88] [82]
Bin  2 = *****************  [72] [67]
Bin  3 = *************  [56] [55]
Bin  4 = **********  [46] [45]
Bin  5 = *********  [39] [37]
Bin  6 = *******  [32] [30]
Bin  7 = ******  [25] [25]
Bin  8 = *****  [21] [20]
Bin  9 = *****  [19] [17]
Bin 10 = ****  [16] [14]
Bin 11 = ***  [12] [11]
Bin 12 = **  [9] [9]
Bin 13 = **  [8] [7]
Bin 14 = **  [7] [6]
Bin 15 = *  [5] [5]
Bin 16 = *  [4] [4]
Bin 17 = *  [3] [3]
Bin 18 = *  [2] [3]
Bin 19 = *  [2] [2]
```

Exponential Distribution

The exponential distribution occurs naturally when describing the lengths of the inter-arrival times in a homogeneous Poisson process. A Poisson process is the stochastic process in which events occur continuously and independently of one another. A well-known example is radioactive decay of atoms. The key characteristics of the exponential distribution are summarized below [65, 66, 49].

$$\text{Parameters: } a > 0$$

$$\text{PDF: } f(x) = \begin{cases} a\, e^{-ax} & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Random variates X from the exponential distribution can be generated using the inversion transform method which results in the formula shown below.

$$X = -\frac{1}{a}\log u$$

where $u \sim u[0,1]$ is a random variate drawn from the uniform distribution. The code below illustrates how the exponential distribution may be implemented in C#.

```
public static double ExponentialPDF(double x, double a)
{
    return a * Math.Exp(-a * x);
}

public static double[] ExponentialPDF(double[] x, double a)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    { tempArr[i] = ExponentialPDF(x[i], a); }
    return tempArr;
}

public static double NextExponential(double a)
{
    return -Math.Log(randObj.NextDouble()) / a;
}

public static double[] NextExponential(double a, int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    { tempArr[i] = NextExponential(a); }
    return tempArr;
}

static void TestExponential()
{
    int nBins = 20;
    int nPoints = 10000;
    double xmin = 0.0; double xmax = 4.0;
    double alphaVal = 1.5;
    double displayScaleFactor = 500.0;
    double[] randObj = NextExponential(alphaVal, nPoints);
    double[] HistogramValues =
        makeHistogram(randObj, xmin, xmax, nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];
    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = xmin + (i + 0.5) * (xmax - xmin) / nBins;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i]=ExponentialPDF(xdata[i],alphaVal);
    }
    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);
    Console.WriteLine("\nTesting the Exponential Probability
        Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
        [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
        Console.Write("Bin {0,2:n0} = ", i + 1);
        for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
        Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                  ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                  yCalcDistributionMax,0));
    }
    Console.WriteLine("\nPress ENTER key to continue...");
    Console.ReadLine();
    Console.Clear();
}
```

```
Results: Exponential Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 = *************************  [100] [100]
Bin  1 = ******************  [73] [74]
Bin  2 = *************  [54] [55]
Bin  3 = **********  [40] [41]
Bin  4 = ********  [31] [30]
Bin  5 = *****  [20] [22]
Bin  6 = ****  [16] [17]
Bin  7 = ***  [12] [12]
Bin  8 = **  [10] [9]
Bin  9 = **  [7] [7]
Bin 10 = *  [4] [5]
Bin 11 = *  [3] [4]
Bin 12 = *  [3] [3]
Bin 13 =   [2] [2]
Bin 14 =   [2] [1]
Bin 15 =   [1] [1]
Bin 16 =   [1] [1]
Bin 17 =   [1] [1]
Bin 18 =   [1] [0]
Bin 19 =   [0] [0]
```

Extreme Value Distribution

Extreme value theory is a branch of statistics responsible for studying extreme deviations from the median of probability distributions. Consequently, extreme value theory is important for assessing risk for highly unusual events, such as 100-year floods. Extreme value distributions are usually considered to comprise of the following three families:

Type 1: Gumbel-type distributions.
Type 2: Frechet-type distributions.
Type 3: Weibull-type distributions.

The key characteristics of the extreme value distribution of type 1 (Gumbel-type) are summarized below [66].

$$\text{Parameters: } -\infty < \mu < \infty,\ \sigma > 0$$
$$\text{Range: } -\infty < x < \infty$$
$$\text{PDF: } f(x) = \frac{1}{\sigma}\exp(-\frac{x-\mu}{\sigma})\exp(-\exp(-\frac{x-\mu}{\sigma}))$$

Given a random variate u drawn from the uniform distribution $u(0,1]$, the variate

$$X = \mu - \sigma\log(-\log(u))$$

has a Gumbel distribution with parameters μ and σ. The following code illustrates how the extreme value distribution (Gumbel-type) may be implemented in C#.

```
public static double ExtremeValuePDF(double x,double mu,double sigma)
{
    double w = (x-mu)/sigma;
    return (1.0/sigma) * Math.Exp(-w) * Math.Exp(-Math.Exp(-w));
}

public static double[] ExtremeValuePDF(double[] x,double mu,double
    sigma)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    {
        tempArr[i] = ExtremeValuePDF(x[i], mu, sigma);
    }
    return tempArr;
}

public static double NextExtremeValue(double mu, double sigma)
{
    return (mu - sigma*Math.Log(-Math.Log(randObj.NextDouble())));
}
```

```
public static double[] NextExtremeValue(double mu, double sigma, int
    nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextExtremeValue(mu,sigma);
    }
    return tempArr;
}

static void TestExtremeValue()
{
    int nBins = 20;
    int nPoints = 10000;
    double xmin =-4;
    double xmax = 8;
    double muVal = 1.0;
    double sigmaVal = 0.750;
    double displayScaleFactor = 500.0;

    double[] randObj=NextExtremeValue(muVal,sigmaVal, nPoints);
    double[] HistogramValues=makeHistogram(randObj,xmin,xmax,nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];

    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
      xdata[i] = xmin + (i + 0.5) * (xmax - xmin) / nBins;
      ydata[i] = (double)HistogramValues[i];
      yCalcDistribution[i]=ExtremeValuePDF(xdata[i],muVal,sigmaVal);
    }

    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);

    Console.WriteLine("\nTesting the Extreme Value Probability
        Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
        [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
        Console.Write("Bin {0,2:n0} = ", i + 1);
        for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
        Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                  ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                  yCalcDistributionMax,0));
    }
    Console.WriteLine("\nPress ENTER key to continue...");
    Console.ReadLine();
    Console.Clear();
}
```

```
Results: Extreme Value Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 =   [0] [0]
Bin  1 =   [0] [0]
Bin  2 =   [0] [0]
Bin  3 =   [0] [0]
Bin  4 =   [0] [0]
Bin  5 =   [0] [0]
Bin  6 = ****  [17] [16]
Bin  7 = ******************  [77] [76]
Bin  8 = ************************  [100] [100]
Bin  9 = *****************  [73] [73]
Bin 10 = *********  [40] [41]
Bin 11 = *****  [20] [20]
Bin 12 = **  [10] [9]
Bin 13 = *  [5] [4]
Bin 14 = *  [2] [2]
Bin 15 =   [1] [1]
Bin 16 =   [0] [0]
Bin 17 =   [0] [0]
Bin 18 =   [0] [0]
Bin 19 =   [0] [0]
```

Gamma Distribution

The gamma distribution is essentially a generalization of the Erlang distribution. Like the exponential and Erlang distributions, the gamma distribution is also used in the modeling of some queueing networks. The key characteristics of the gamma distribution are summarized below [65, 66, 49].

$$\text{Parameters: } a > 0,\ b > 0$$

$$\text{PDF: } f(x) = \begin{cases} \dfrac{a^b\, x^{b-1} e^{-a\,x}}{\Gamma(b)} & \text{if } 0 \le x < \infty \\ 0 & \text{if } x < 0 \end{cases}$$

Given a random variate u drawn from the uniform distribution $u[0,1]$, the variate

$$X = -\frac{1}{a}\sum_{i=1}^{b} \log u_i$$

has a gamma distribution with parameters a and b. The code below illustrates how the gamma distribution may be implemented in C#.

```
public static double GammaPDF(double x, int b, double a)
{
    return Math.Pow(a,b)*Math.Pow(x,b-1)*Math.Exp(-a*x)/Gamma(b);
}
```

```
public static double[] GammaPDF(double[] x, int b, double a)
{
    double[] tempArray = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    {
        tempArray[i] = GammaPDF(x[i], b, a);
    }
    return tempArray;
}

public static double NextGamma(int b, double a)
{
    double temp = 0.0;
    for (int i = 0; i < b; i++)
    {
        temp += -Math.Log(randObj.NextDouble()) / a;
    }
    return temp;
}

public static double[] NextGamma(int b, double a, int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextGamma(b, a);
    }
    return tempArr;
}

static void TestGamma()
{
  int nBins = 20;       int nPoints = 10000;
  double xmin = 0.0;    double xmax = 15.0;
  double aVal = 0.6;    int bVal = 2;
  double displayScaleFactor = 500.0;
  double[] randObj = NextGamma(bVal, aVal, nPoints);
  double[] HistogramValues =
        makeHistogram(randObj, xmin, xmax, nBins);
  double[] xdata = new double[nBins];
  double[] ydata = new double[nBins];
  double[] yCalcDistribution = new double[nBins];
  for (int i = 0; i < nBins; i++)
  {
    xdata[i] = xmin + (i + 0.5) * (xmax - xmin) / nBins;
    ydata[i] = (double)HistogramValues[i];
    yCalcDistribution[i] = GammaPDF(xdata[i], bVal, aVal);
  }
  double ydataMax = dataMax(ydata);
  double yCalcDistributionMax = dataMax(yCalcDistribution);
  Console.WriteLine("\nTesting the Gamma Probability Distribution");
  Console.WriteLine("\nKey: Bin number = *****... [random distrib.] [
      calculated distrib.]\n");
  for (int i = 0; i < nBins; i++)
  {
    Console.Write("Bin {0,2:n0} = ", i + 1);
    for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
              nBins / ydataMax, 0)); j++) Console.Write("*");
    Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
             ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
             yCalcDistributionMax,0));
  }
  Console.WriteLine("\nPress ENTER key to continue...");
  Console.ReadLine(); Console.Clear();
}
```

```
Results: Gamma Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 = ************  [48] [49]
Bin  1 = ***********************  [95] [94]
Bin  2 = ************************  [100] [100]
Bin  3 = ***********************  [95] [89]
Bin  4 = *******************  [78] [73]
Bin  5 = **************  [55] [57]
Bin  6 = ***********  [46] [43]
Bin  7 = ********  [30] [32]
Bin  8 = ******  [24] [23]
Bin  9 = ****  [17] [16]
Bin 10 = ***  [13] [11]
Bin 11 = **  [8] [8]
Bin 12 = **  [6] [6]
Bin 13 = *  [3] [4]
Bin 14 = *  [2] [3]
Bin 15 =   [2] [2]
Bin 16 =   [1] [1]
Bin 17 =   [1] [1]
Bin 18 =   [1] [1]
Bin 19 =   [0] [0]
```

Laplace Distribution

The Laplace distribution is a continuous probability distribution consisting of two exponential distributions, one positive and one negative. A major distinguishing feature of the Laplace distribution is that it has significantly longer tails compared to the normal distribution. In recent years the Laplace distribution has been applied in various disciplines such as communications, economics, engineering and finance. The key characteristics of the Laplace distribution are summarized below [65, 66, 49].

Parameters: $\mu > 0,\ b > 0$

Range: $-\infty < x < \infty$

$$\text{PDF: } f(x) = \frac{1}{2b}\exp\left(-\frac{|x-\mu|}{b}\right) = \frac{1}{2b}\begin{cases}\exp\left(-\frac{\mu-x}{b}\right) & \text{if } x < \mu \\ \exp\left(-\frac{x-\mu}{b}\right) & \text{if } x \geq \mu\end{cases}$$

The inverse transform method can be used to obtain equations for generating random variates X from the Laplace distribution. The results are shown below.

$$X = \begin{cases} \mu + b\log(2u) & \text{for } u \leq 0.5 \\ \mu - b\log(2(1-u)) & \text{for } u > 0.5 \end{cases}$$

where $u \sim u[0,1]$ is a random variate drawn from uniform distribution. The following code illustrates how the Laplace distribution may be implemented in C#.

```
public static double LaplacePDF(double x, double sigma)
{
    double coef = Math.Sqrt(2.0) / sigma;
    return 0.5 * coef * Math.Exp(-coef * Math.Abs(x));
}

public static double[] LaplacePDF(double[] x, double sigma)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    {
        tempArr[i] = LaplacePDF(x[i], sigma);
    }
    return tempArr;
}

public static double NextLaplace(double sigma)
{
    double coef = sigma / Math.Sqrt(2.0);
    double u = randObj.NextDouble();
    if ((u > 0.0) && (u < 0.5))
        return coef * Math.Log(2.0 * u);
    else if ((u >= 0.5) && (u < 1.0))
        return coef * Math.Log(1.0 / (2.0 * (1.0 - u)));
    else return 0.0;
}

public static double[] NextLaplace(double sigma, int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextLaplace(sigma);
    }
    return tempArr;
}

static void TestLaplace()
{
    int nBins = 20;
    int nPoints = 10000;
    double xmin = -5.0;
    double xmax = 5.0;
    double sigmaVal = 2.0;
    double displayScaleFactor = 500.0;

    double[] randObj = NextLaplace(sigmaVal, nPoints);
    double[] HistogramValues =
        makeHistogram(randObj, xmin, xmax, nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];

    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = xmin + (i + 0.5) * (xmax - xmin) / nBins;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i] = LaplacePDF(xdata[i], sigmaVal);
    }
    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);
    Console.WriteLine("\nTesting the Laplace Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
         [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
      Console.Write("Bin {0,2:n0} = ", i + 1);
      for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
      Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                yCalcDistributionMax,0));
    }
    Console.WriteLine("\nPress ENTER key to continue...");
    Console.ReadLine();
    Console.Clear();
}
```

```
Results: Laplace Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 = *  [5] [4]
Bin  1 = *  [5] [6]
Bin  2 = **  [8] [8]
Bin  3 = ***  [12] [12]
Bin  4 = ****  [17] [17]
Bin  5 = ******  [24] [24]
Bin  6 = *********  [38] [35]
Bin  7 = ************  [47] [49]
Bin  8 = *****************  [68] [70]
Bin  9 = ***********************  [93] [98]
Bin 10 = *************************  [99] [100]
Bin 11 = *****************  [71] [70]
Bin 12 = ************  [49] [49]
Bin 13 = ********  [33] [35]
Bin 14 = ******  [26] [24]
Bin 15 = ****  [17] [17]
Bin 16 = ***  [13] [12]
Bin 17 = **  [8] [8]
Bin 18 = *  [5] [6]
Bin 19 = *  [4] [4]
```

Logistic Distribution

The logistic distribution has been extensively applied to many different disciplines. In biology, for example, it has been used to model how certain population species grow in competition. In epidemiology, it has been use to model the spreading of epidemics. The key characteristics of the logistic distribution are summarized below [66].

$$\text{Parameters: } -\infty < a < \infty \quad \text{and} \quad b > 0$$

$$\text{Range: } -\infty < x < \infty$$

$$\text{PDF: } f(x) = \frac{\exp(\frac{x-a}{b})}{b\left(1+\exp(\frac{x-a}{b})\right)^2}$$

Using the inverse transform method we may obtain an equation for generating random variates X from the logistic distribution as shown below.

$$X = a + b \log\left(\frac{u}{1-u}\right)$$

where u is a random variate drawn from the uniform distribution $u[0,1]$. The following code illustrates how the logistic distribution may be implemented in C#.

```
public static double LogisticPDF(double x, double b, double a)
{
return Math.Exp(-(x-a)/b)/(b*Math.Pow((1.0+Math.Exp(-(x-a)/b)),2.0));
}

public static double[] LogisticPDF(double[] x,double b,double a)
{
  double[] tempArr = new double[x.Length];
  for (int i = 0; i < x.Length; i++)
  { tempArr[i] = LogisticPDF(x[i], b, a); }
  return tempArr;
}

public static double NextLogistic(double b, double a)
{
  double u = randObj.NextDouble();
  return a + b * Math.Log(u / (1 - u));
}

public static double[] NextLogistic(double b, double a, int nLen)
{
  double[] tempArr = new double[nLen];
  for (int i = 0; i < nLen; i++)
  { tempArr[i] = NextLogistic(b,a); }
  return tempArr;
}
```

```
static void TestLogistic()
{
  int nBins = 20; int nPoints = 10000; double xmin = -5;
  double xmax = 15; double aVal = 2.0; double bVal = 1.0;
  double displayScaleFactor = 500.0;
  double[] randObj = NextLogistic(bVal,aVal,nPoints);
  double[] HistogramValues = makeHistogram(randObj,xmin,xmax,nBins);
  double[] xdata = new double[nBins];
  double[] ydata = new double[nBins];
  double[] yCalcDistribution = new double[nBins];
  for (int i = 0; i < nBins; i++)
  {
    xdata[i] = xmin + (i + 0.5) * (xmax - xmin) / nBins;
    ydata[i] = (double)HistogramValues[i];
    yCalcDistribution[i] = LogisticPDF(xdata[i], bVal, aVal);
  }
  double ydataMax = dataMax(ydata);
  double yCalcDistributionMax = dataMax(yCalcDistribution);
  Console.WriteLine("\nTesting the Logistic Probability Distribution"
      );
  Console.WriteLine("\nKey: Bin number = *****... [random distrib.] [
      calculated distrib.]\n");
  for (int i = 0; i < nBins; i++)
  {
    Console.Write("Bin {0,2:n0} = ", i + 1);
    for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
    Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
               ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                yCalcDistributionMax,0));
  }
}

Results: Logistic Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]
Bin  0 =   [1] [1]
Bin  1 =   [2] [2]
Bin  2 = *  [5] [5]
Bin  3 = ***  [12] [12]
Bin  4 = *******  [30] [30]
Bin  5 = ***************  [62] [63]
Bin  6 = *************************  [100] [100]
Bin  7 = ************************  [96] [100]
Bin  8 = ****************  [65] [63]
Bin  9 = ********  [30] [30]
Bin 10 = ***  [13] [12]
Bin 11 = *  [5] [5]
Bin 12 =   [1] [2]
Bin 13 =   [1] [1]
Bin 14 =   [0] [0]
Bin 15 =   [0] [0]
Bin 16 =   [0] [0]
Bin 17 =   [0] [0]
Bin 18 =   [0] [0]
Bin 19 =   [0] [0]
```

Lognormal Distribution

The lognormal distribution is commonly used to model the lives of units whose failure modes are of a fatigue-stress nature. Since this includes most, if not all, mechanical systems, the lognormal distribution can have widespread application. The key characteristics of the lognormal distribution are summarized below [66].

Parameters: $\mu > 0$ and $\sigma > 0$

$$\text{PDF: } f(x) = \begin{cases} \dfrac{1}{\sigma x\sqrt{2\pi}} e^{-(\log x-\mu)^2/2\sigma^2} & \text{if } 0 \leq x < \infty \\ 0 & \text{otherwise} \end{cases}$$

Given a random variate $N(0,1)$ drawn from the normal distribution with 0 mean and 1 standard deviation, then the variate

$$X = e^{\mu+\sigma N(0,1)}$$

has a lognormal distribution with parameters μ and σ. The following code illustrates how the lognormal distribution may be implemented in C#.

```
public static double LognormalPDF(double x, double mu, double sigma)
{
    double x1, x2;
    if (x > 0.0)
    {
        x1 = 1.0/(sigma*x*Math.Sqrt(2.0*Math.PI));
        x2 = (Math.Log(x)-mu)*(Math.Log(x)-mu)/(2.0*sigma*sigma);
        return x1*Math.Exp(-x2);
    }
    else
        return 0.0;
}


public static double[] LognormalPDF(double[] x, double mu, double
    sigma)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    {
        tempArr[i] = LognormalPDF(x[i], mu, sigma);
    }
    return tempArr;
}


public static double NextLognormal(double mu, double sigma)
{
    return Math.Exp(mu + sigma * NextNormal(0.0, 1.0));
}
```

```
public static double[] NextLognormal(double mu, double sigma, int
    nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextLognormal(mu, sigma);
    }
    return tempArr;
}

static void TestLognormal()
{
  int nBins = 20;
  int nPoints = 10000;
  double xmin = 0.0;
  double xmax = 5.0;
  double mu = 0.0;
  double sigma =0.5;
  double displayScaleFactor = 500.0;

  double[] randObj = NextLognormal(mu, sigma, nPoints);
  double[] HistogramValues =
        makeHistogram(randObj, xmin, xmax, nBins);
  double[] xdata = new double[nBins];
  double[] ydata = new double[nBins];

  double[] yCalcDistribution = new double[nBins];
  for (int i = 0; i < nBins; i++)
  {
    xdata[i] = xmin + (i + 0.5) * (xmax - xmin) / nBins;
    ydata[i] = (double)HistogramValues[i];
    yCalcDistribution[i]=LognormalPDF(xdata[i],mu,sigma);
  }

  double ydataMax = dataMax(ydata);
  double yCalcDistributionMax = dataMax(yCalcDistribution);

  Console.WriteLine("\nTesting the Lognormal Probability
        Distribution");
  Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
        [calculated distrib.]\n");
  for (int i = 0; i < nBins; i++)
  {
    Console.Write("Bin {0,2:n0} = ", i + 1);
    for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
          nBins / ydataMax, 0)); j++) Console.Write("*");
    Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
              ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
              yCalcDistributionMax,0));
  }
  Console.WriteLine("\nPress ENTER key to continue...");
  Console.ReadLine(); Console.Clear();
}
```

```
Results: Lognormal Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 =   [1] [0]
Bin  1 = *********  [35] [35]
Bin  2 = ***********************  [94] [93]
Bin  3 = *************************  [100] [100]
Bin  4 = ********************  [81] [78]
Bin  5 = *************  [53] [54]
Bin  6 = *********  [34] [35]
Bin  7 = ******  [23] [22]
Bin  8 = ***  [13] [14]
Bin  9 = **  [9] [9]
Bin 10 = **  [6] [5]
Bin 11 = *  [3] [3]
Bin 12 = *  [2] [2]
Bin 13 =   [1] [1]
Bin 14 =   [1] [1]
Bin 15 =   [1] [1]
Bin 16 =   [0] [0]
Bin 17 =   [0] [0]
Bin 18 =   [0] [0]
Bin 19 =   [0] [0]
```

Normal Distribution

The normal or Gaussian distribution is a continuous probability distribution that is often used to describe, at least approximately, any variable or data that tends to cluster around some mean or average value. The normal distribution is characterized by two parameters: its mean value μ and its standard deviation σ. The graph of the associated probability density function is bell-shaped, with a peak at the mean, and is known as the Gaussian function or bell curve. The key characteristics of the normal distribution are summarized below [65, 66, 49].

$$\begin{aligned}\text{Parameters: } \mu &= \text{ mean}\\ \sigma &= \text{ standard deviation, } \sigma > 0\\ \text{Range: } &-\infty < x < \infty\\ \text{PDF: } f(x) &= \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}\end{aligned}$$

There are several methods available for calculating random variates from a normal distribution. However, not all of them are optimized for speed. The Box-Muller transform, for example, is a well known method for generating pairs of independent random numbers that are characterized by a normal distribution. Suppose u_1 and u_2 are independent random variables drawn from a uniform distribution $u[0,1]$. Then by applying the Box-Muller transform

$$\begin{aligned}X_0 &= \sqrt{-2\log u_1}\cos(2\pi u_2)\\ Y_0 &= \sqrt{-2\log u_1}\sin(2\pi u_2)\end{aligned}$$

it can be shown that X_0 and Y_0 are independent random variables with a normal distribution. However, having to calculate a lot of logarithms and trigonometric functions can become computationally expensive for large data samples.

The Marsaglia polar method provides a somewhat faster and more efficient way for generating a pair of independent standard normal random variables. This algorithm works by choosing random points (x,y) in the square $-1 < x < 1$, $-1 < y < 1$ until $s = x^2 + y^2 < 1$ and then returning the required pair of normal random variables as

$$X_0 = x\sqrt{-2\log(s)/s}$$
$$Y_0 = y\sqrt{-2\log(s)/s}$$

The following code illustrates how the normal probability density distribution may be implemented in C# [65, 66, 49]:.

```
public static double NormalPDF(double x, double mu, double sigma)
{
  double x1 = 1 / sigma / Math.Sqrt(2 * Math.PI);
  double x2 = (x - mu) * (x - mu) / (2 * sigma * sigma);
  return x1 * Math.Exp(-x2);
}


public static double[] NormalPDF(double[] x, double mu, double sigma)
{
  double[] tempArr = new double[x.Length];
  for (int i = 0; i < x.Length; i++)
  {
     tempArr[i] = NormalPDF(x[i], mu, sigma);
  }
  return tempArr;
}


public static double NextNormal(double mu, double sigma)
{
  double x = 2.0 * randObj.NextDouble() - 1.0;
  double y = 2.0 * randObj.NextDouble() - 1.0;
  double s = x * x + y * y;

  while (s > 1.0)
  {
    x = 2.0 * randObj.NextDouble() - 1.0;
    y = 2.0 * randObj.NextDouble() - 1.0;
    s = x * x + y * y;
  }

  double xGaussian = Math.Sqrt(-2.0*Math.Log(s)/s)*x*sigma+mu;
  double yGaussian = Math.Sqrt(-2.0*Math.Log(s)/s)*y*sigma+mu;

  return xGaussian;
}
```

```
public static double[] NextNormal(double mu, double sigma, int nLen)
{
  double[] tempArr = new double[nLen];
  for (int i = 0; i < nLen; i++)
  {
    tempArr[i] = NextNormal(mu, sigma);
  }
  return tempArr;
}

static void TestNormal()
{
  int nBins = 20;
  int nPoints = 10000;
  double xmin = 0.0;
  double xmax = 4.0;
  double mu = 2.0;
  double sigma = 0.5;
  double displayScaleFactor = 500.0;

  double[] randObj = NextNormal(mu, sigma, nPoints);
  double[] HistogramValues =
              makeHistogram(randObj, xmin, xmax, nBins);
  double[] xdata = new double[nBins];
  double[] ydata = new double[nBins];

  double[] yCalcDistribution = new double[nBins];
  for (int i = 0; i < nBins; i++)
  {
    xdata[i] = xmin + (i + 0.5) * (xmax - xmin) / nBins;
    ydata[i] = (double)HistogramValues[i];
    yCalcDistribution[i] = NormalPDF(xdata[i], mu, sigma);
  }

  double ydataMax = dataMax(ydata);
  double yCalcDistributionMax = dataMax(yCalcDistribution);

  Console.WriteLine("\nTesting the Normal Probability Distribution");
  Console.WriteLine("\nKey: Bin number = *****... [random distrib.] [
      calculated distrib.]\n");

  for (int i = 0; i < nBins; i++)
  {
    Console.Write("Bin {0,2:n0} = ", i + 1);
    for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
        nBins / ydataMax, 0)); j++) Console.Write("*");
    Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
        ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
        yCalcDistributionMax,0));
  }
  Console.WriteLine("\nPress ENTER key to continue...");
  Console.ReadLine();
  Console.Clear();
}
```

```
Results: Normal (Gaussian) Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 =   [0] [0]
Bin  1 =   [0] [0]
Bin  2 =   [1] [1]
Bin  3 = *  [3] [3]
Bin  4 = **  [7] [9]
Bin  5 = *****  [20] [20]
Bin  6 = *********  [38] [38]
Bin  7 = ***************  [60] [62]
Bin  8 = **********************  [87] [85]
Bin  9 = *************************  [100] [100]
Bin 10 = *************************  [99] [100]
Bin 11 = *********************  [83] [85]
Bin 12 = ***************  [59] [62]
Bin 13 = **********  [39] [38]
Bin 14 = *****  [21] [20]
Bin 15 = **  [8] [9]
Bin 16 = *  [4] [3]
Bin 17 =   [1] [1]
Bin 18 =   [1] [0]
Bin 19 =   [0] [0]
```

Pareto Distribution

The Pareto distribution is a power law probability distribution that coincides with social, scientific, geophysical, actuarial, and many other types of observable phenomena. Outside the field of economics it is at times also referred to as the Bradford distribution. The Pareto distribution was originally used to describe the allocation of wealth among individuals since it seemed to show rather well the way that a larger portion of the wealth of any society is owned by a smaller percentage of the people in that society. The key characteristics of the Pareto distribution are summarized below [66].

$$\text{Parameters: } a > 0,\ b > 0$$

$$\text{PDF: } f(x) = \begin{cases} b\,a^b x^{-b-1} & \text{if } a \le x < \infty \\ 0 & \text{otherwise} \end{cases}$$

An equation for generating random variates X from the Pareto probability distribution may be obtained by applying the inverse transform method to the corresponding PDF which leads to the following result below.

$$X = a\,u^{-1/b}$$

where $u \sim u[0,1]$ is a random variate drawn from the uniform distribution. The following code illustrates how the Pareto distribution may be implemented in C#.

```
public static double ParetoPDF(double x, double a, double b)
{
    if (x > a)
        return b * Math.Pow(a, b) * Math.Pow(x, -b - 1.0);
    else
        return 0.0;
}

public static double[] ParetoPDF(double[] x, double a, double b)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    { tempArr[i] = ParetoPDF(x[i], a, b); }
    return tempArr;
}

public static double NextPareto(double a, double b)
{
    return a * Math.Pow(randObj.NextDouble(), -1.0 / b);
}

public static double[] NextPareto(double a, double b, int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    { tempArr[i] = NextPareto(a,b); }
    return tempArr;
}

static void TestPareto()
{
    int nBins = 20; int nPoints = 10000; double xmin = 0.0;
    double xmax = 5.0; double bVal = 1.0; double aVal = 1.0;
    double displayScaleFactor = 500.0;
    double[] randObj = NextPareto(aVal, bVal, nPoints);
    double[] HistogramValues = makeHistogram(randObj, xmin, xmax,
        nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];
    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = xmin + (i + 0.5) * (xmax - xmin) / nBins;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i] = ParetoPDF(xdata[i], aVal, bVal);
    }
    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);
    Console.WriteLine("\nTesting Pareto Probability Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
         [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
      Console.Write("Bin {0,2:n0} = ", i + 1);
      for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
               nBins / ydataMax, 0)); j++) Console.Write("*");
      Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                yCalcDistributionMax,0));
    }
}

Results: Pareto Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 =   [0] [0]
Bin  1 =   [0] [0]
Bin  2 =   [0] [0]
Bin  3 =   [0] [0]
Bin  4 = *************************  [100] [100]
Bin  5 = ****************  [65] [67]
Bin  6 = ************  [48] [48]
Bin  7 = *********  [37] [36]
Bin  8 = *******  [28] [28]
Bin  9 = ******  [23] [22]
Bin 10 = ****  [18] [18]
Bin 11 = ****  [17] [15]
Bin 12 = ***  [14] [13]
Bin 13 = ***  [11] [11]
Bin 14 = **  [10] [10]
Bin 15 = **  [8] [8]
Bin 16 = **  [8] [7]
Bin 17 = **  [7] [7]
Bin 18 = *  [5] [6]
Bin 19 = *  [5] [5]
```

Rayleigh Distribution

The Rayleigh distribution is a continuous probability distribution which can arise when a two-dimensional vector has elements that are normally distributed, are uncorrelated, and have equal variance. The vector's magnitude is then said to have a Rayleigh distribution. The key characteristics of the Rayleigh distribution are summarized below [65, 66, 49].

Parameters: $\sigma > 0$

$$\text{PDF: } f(x) = \begin{cases} \dfrac{x}{\sigma^2} \exp\left(\dfrac{-x^2}{2\sigma^2}\right) & \text{if } 0 \le x < \infty \\ 0 & \text{otherwise} \end{cases}$$

An equation for generating random variates X from the Rayleigh probability distribution may be obtained by applying the inverse transform method to the corresponding PDF which leads to the following result below.

$$X = \sigma\sqrt{-2\log(u)}$$

where $u \sim u(0,1]$ is a random variate drawn from the uniform distribution. The following code illustrates how the Rayleigh distribution may be implemented in C#.

```
public static double RayleighPDF(double x, double sigma)
{
    if (x >= 0.0)
        return (x/sigma/sigma) * Math.Exp(-x*x/2.0/sigma/sigma);
    else
        return 0.0;
}

public static double[] RayleighPDF(double[] x, double sigma)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    { tempArr[i] = RayleighPDF(x[i], sigma); }
    return tempArr;
}

public static double NextRayleigh(double sigma)
{
    double u = randObj.NextDouble();
    if (u != 0.0)
        return sigma * Math.Sqrt(-2.0 * Math.Log(u));
    else
        return double.NaN;
}

public static double[] NextRayleigh(double sigma, int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    { tempArr[i] = NextRayleigh(sigma); }
    return tempArr;
}

static void TestRayleigh()
{
    int nBins=20; int nPoints=10000; double xmin=0; double xmax=4;
    double sigma = 0.5; double displayScaleFactor = 500.0;
    double[] randObj = NextRayleigh(sigma, nPoints);
    double[] HistogramValues =
        makeHistogram(randObj, xmin, xmax, nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];
    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = xmin + (i + 0.5) * (xmax - xmin) / nBins;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i] = RayleighPDF(xdata[i], sigma);
    }
    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);
    Console.WriteLine("\nTesting the Rayleigh Probability
        Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
         [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
      Console.Write("Bin {0,2:n0} = ", i + 1);
      for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
      Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                yCalcDistributionMax,0));
    }
}
```

```
Results: Rayleigh Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]
Bin  0 = ********  [32] [32]
Bin  1 = *********************  [83] [83]
Bin  2 = *************************  [100] [100]
Bin  3 = *********************  [84] [87]
Bin  4 = **************  [57] [59]
Bin  5 = ********  [32] [32]
Bin  6 = ****  [14] [15]
Bin  7 = *  [6] [5]
Bin  8 =   [2] [2]
Bin  9 =   [0] [0]
Bin 10 =   [0] [0]
Bin 11 =   [0] [0]
Bin 12 =   [0] [0]
Bin 13 =   [0] [0]
Bin 14 =   [0] [0]
Bin 15 =   [0] [0]
Bin 16 =   [0] [0]
Bin 17 =   [0] [0]
Bin 18 =   [0] [0]
Bin 19 =   [0] [0]
```

Student-t Distribution

The student's *t*-distribution, also known as just the *t*-distribution, is a probability distribution that arises when estimating the mean of a normally distributed population when the sample size is small and the population of the standard deviation is unknown and has to be estimated from the data. When we speak of a specific *t*-distribution, we have to specify the degrees of freedom as there is a different *t*-distribution for each sample size. The *t*-distribution curves are symmetric and bell-shaped like the normal distribution and have their peak at 0. However, the spread is more than that of the standard normal distribution. The larger the degrees of freedom, the closer the *t*-distribution is to the normal distribution. The key characteristics of the student-t distribution are summarized below [65, 66, 49].

Parameters: $n > 0$

Range: $-\infty < x < \infty$

$$\text{PDF: } f(x) = \frac{\Gamma((n+1)/2)}{\Gamma(1/2)\sqrt{n}\,\Gamma(n/2)}\left(1+\frac{x^2}{n}\right)^{-(n+1)/2}$$

where the parameter n is the degrees of freedom and Γ is the gamma function.

Random variates X from the student's t-distribution can be obtained by the following equation below.

$$X = \frac{N(0,1)}{\chi(n,1)}$$

where $N(0,1)$ is a random variate drawn from the normal distribution with $\sigma = 1$ and $\mu = 0$ and $\chi(n,1)$ is a random variate drawn from the χ distribution with n degrees of freedom and $\sigma = 1$. The following code illustrates how the student's t-distribution may be implemented in C#.

```
public static double StudentTPDF(double x, int n)
{
    double gamma1 = Gamma((n + 1.0) / 2.0);
    double gamma2 = Gamma(1.0 / 2.0);
    double gamma3 = Gamma(n / 2.0);
    return Math.Pow(n,-0.5)*Math.Pow(1+x*x/n,-(n+1)/2) *
                gamma1/gamma2/gamma3;
}

public static double[] StudentTPDF(double[] x, int n)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    { tempArr[i] = StudentTPDF(x[i], n); }
    return tempArr;
}

public static double NextStudentT(int n)
{
    return NextNormal(0, 1) / NextChi(n);
}

public static double[] NextStudentT(int n, int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    { tempArr[i] = NextStudentT(n); }
    return tempArr;
}

static void TestStudentT()
{
    int nBins = 20; int nPoints = 10000;
    double xmin = -7.0; double xmax = 7.0; int nVal = 5;
    double displayScaleFactor = 500.0;
    double[] randObj = NextStudentT(nVal, nPoints);
    double[] HistogramValues =
        makeHistogram(randObj, xmin, xmax, nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];
    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = xmin + (i + 0.5) * (xmax - xmin) / nBins;
        ydata[i] = HistogramValues[i];
        yCalcDistribution[i] = StudentTPDF(xdata[i], nVal);
    }
    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);
    Console.WriteLine("\nTesting the Student-T Probability
        Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
        [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
      Console.Write("Bin {0,2:n0} = ", i + 1);
      for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
      Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                yCalcDistributionMax,0));
    }
}
```

```
Results: Student-T Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 =   [0] [0]
Bin  1 =   [0] [0]
Bin  2 =   [0] [0]
Bin  3 =   [1] [1]
Bin  4 =   [2] [2]
Bin  5 = *  [4] [4]
Bin  6 = ***  [11] [10]
Bin  7 = *******  [27] [26]
Bin  8 = ***************  [61] [59]
Bin  9 = ************************  [99] [100]
Bin 10 = ************************  [100] [100]
Bin 11 = ***************  [60] [59]
Bin 12 = *******  [28] [26]
Bin 13 = **  [10] [10]
Bin 14 = *  [4] [4]
Bin 15 =   [2] [2]
Bin 16 =   [1] [1]
Bin 17 =   [0] [0]
Bin 18 =   [0] [0]
Bin 19 =   [0] [0]
```

Triangular Distribution

The triangular distribution is typically used as a subjective description of a population for which there is only a limited amount of sample data. It is based on a knowledge of the minimum and maximum values and an inspired guess as to what the modal value might be. Despite being a simplistic description of a population, the triangular distribution is a very useful tool for modeling processes where the relationship between variables is known, but data is scarce possibly because of the high cost of collection. Because of these unique features, the triangular distribution is often used in business decision making, particularly in simulations. In addition, the triangular distribution is also widely used in project planning and management to model events which take place within an interval defined only by a minimum and maximum value. The key characteristics of the triangular distribution are summarized below [65, 66, 49].

$$\begin{aligned}
\text{Parameters: } & a \text{ where } -\infty < a < \infty \\
& b \text{ where } b > a \\
& c \text{ where } a \le c \le b \\
\text{Range: } & a \le x \le b \\
\text{PDF: } f(x) = & \begin{cases} \dfrac{2(x-a)}{(b-a)(c-a)} & \text{for } a \le x \le c \\[2ex] \dfrac{2(b-x)}{(b-a)(b-c)} & \text{for } c \le x \le b \\[2ex] 0 & \text{otherwise} \end{cases}
\end{aligned}$$

An equation for generating random variates X from the triangular probability distribution may be obtained by applying the inverse transform method to the corresponding PDF which leads to the following result below.

```
u=u[0,1]
if u <= (mode - min) / (max - min) then
   X = min + sqrt(u * (max - min) * (mode - min))
else
   X = max - sqrt((1 - u) * (max - min) * (max - mode))
end if
```

The following code shows how the triangular distribution may be implemented in C#.

```
public static double TriangularPMF(double a, double b, double c,
    double x)
{
    if ((a <= x) && (x <= c))
    { return 2.0 * (x - a) / (b - a) / (c - a); }
    else if ((c <= x) && (x <= b))
    { return 2.0 * (b - x) / (b - a) / (b - c); }
    else return 0.0;
}

public static double[] TriangularPMF(double a, double b, double c,
    int[] x)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    { tempArr[i] = TriangularPMF(a, b, c, x[i]); }
    return tempArr;
}
```

```
public static double NextTriangular(double a, double b, double c)
{
    double u = randObj.NextDouble();
    if (u <= ((c - a) / (b - a)))
    {
        return a + Math.Sqrt(u) * Math.Sqrt((c - a) * (b - a));
    }
    else
    {
        return b - Math.Sqrt(u * (b - a) - (c - a)) * (b - c);
    }
}

public static double[] NextTriangular(double a, double b, double c,
    int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    {
        tempArr[i] = NextTriangular(a, b, c);
    }
    return tempArr;
}

static void TestTriangular()
{
    int nBins = 20;
    int nPoints = 10000;
    double aVal = 0.0;
    double bVal = 50;
    double cVal = 100;
    double displayScaleFactor = 500.0;

    double[] randObj = NextTriangular(aVal, bVal, cVal, nPoints);
    double[] HistogramValues = makeHistogram(randObj,
                dataMin(randObj), dataMax(randObj), nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];
    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = i;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i] = TriangularPMF(aVal, bVal, cVal, i);
    }

    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);

    Console.WriteLine("\nTesting the Triangular Probability
        Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
         [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
      Console.Write("Bin {0,2:n0} = ", i + 1);
      for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
               nBins / ydataMax, 0)); j++) Console.Write("*");
      Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
               ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
               yCalcDistributionMax,0));
    }
    Console.WriteLine("\nPress ENTER key to continue...");
    Console.ReadLine();
    Console.Clear();
}
```

```
Results: Triangular Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 = *  [2] [0]
Bin  1 = **  [9] [5]
Bin  2 = ****  [15] [11]
Bin  3 = ****  [15] [16]
Bin  4 = *****  [21] [21]
Bin  5 = ********  [31] [26]
Bin  6 = *********  [35] [32]
Bin  7 = **********  [40] [37]
Bin  8 = ***********  [43] [42]
Bin  9 = *************  [50] [47]
Bin 10 = **************  [56] [53]
Bin 11 = **************  [58] [58]
Bin 12 = ****************  [66] [63]
Bin 13 = ****************  [69] [68]
Bin 14 = ******************  [76] [74]
Bin 15 = ******************  [75] [79]
Bin 16 = *********************  [88] [84]
Bin 17 = **********************  [91] [89]
Bin 18 = ************************  [99] [95]
Bin 19 = ************************  [100] [100]
```

Uniform Distribution (continuous)

A uniform distribution is one for which the probability of occurrence is the same for all values of x. It is sometimes also called a rectangular distribution. The $u[a,b]$ distribution has constant probability density between a and b, and 0 probability density elsewhere. The key characteristics of the continuous uniform distribution are summarized below [65, 66, 49].

$$\text{Parameters: } a = \text{lower limit}$$
$$b = \text{upper limit, } b > a$$
$$\text{PDF: } f(x) = \begin{cases} \dfrac{1}{b-a} & \text{for } a \le x \le b \\ 0 & \text{otherwise} \end{cases}$$

An equation for generating random variates X from the continuous uniform probability distribution may be obtained by applying the inverse transform method to the corresponding PDF which leads to the following simple equation:

$$X = (a + (b - a) * u[0, 1])$$

The code below illustrates how the continuous uniform probability distribution may be implemented in C#.

```
public static double UniformContinuousPDF(double x,double a,double b)
{
    if ((x >= a) && (x < b))
        return (1.0 / (b - a));
    else
        return 0.0;
}

public static double[] UniformContinuousPDF(double[] x, double a,
    double b)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    { tempArr[i] = UniformContinuousPDF(x[i], a, b); }
    return tempArr;
}

public static double NextUniformContinuous(double a, double b)
{
    return (a + (b - a) * randObj.NextDouble());
}

public static double[] NextUniformContinuous(int nLen, double a,
    double b)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    { tempArr[i] = NextUniformContinuous(a, b); }
    return tempArr;
}

static void TestUniformContinuous()
{
    int nBins = 20; int nPoints = 10000; double xmin = 0.0;
    double xmax = 20.0; double a = 5.0; double b = 15.0;
    double displayScaleFactor = 500.0;
    double[] randObj = NextUniformContinuous(nPoints, a, b);
    double[] HistogramValues=makeHistogram(randObj,xmin,xmax,nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];
    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = i;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i] = UniformContinuousPDF(i, a, b);
    }
    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);
    Console.WriteLine("\nTesting the Uniform (Continuous) Probability
         Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
         [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
      Console.Write("Bin {0,2:n0} = ", i + 1);
      for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
      Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                yCalcDistributionMax,0));
    }
}
```

```
Results: Uniform (Continuous) Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]
Bin  0 =   [0] [0]
Bin  1 =   [0] [0]
Bin  2 =   [0] [0]
Bin  3 =   [0] [0]
Bin  4 =   [0] [0]
Bin  5 = ***********************  [91] [100]
Bin  6 = **********************  [88] [100]
Bin  7 = ***********************  [94] [100]
Bin  8 = ************************  [94] [100]
Bin  9 = ***********************  [90] [100]
Bin 10 = *********************  [85] [100]
Bin 11 = ***********************  [92] [100]
Bin 12 = *************************  [100] [100]
Bin 13 = ************************  [95] [100]
Bin 14 = **********************  [89] [100]
Bin 15 =   [0] [0]
Bin 16 =   [0] [0]
Bin 17 =   [0] [0]
Bin 18 =   [0] [0]
Bin 19 =   [0] [0]
```

Weibull Distribution

The Weibull distribution is a very useful probability distribution that is widely used in many different disciplines. For example, the Weibull distribution is one of the most widely used probability distributions in reliability engineering, survival analysis, and weather forecasting to name a few. The key characteristics of the Weibull distribution are summarized below [65, 66, 49].

$$\text{Parameters: } a = \text{scale parameter, } a > 0$$
$$b = \text{shape parameter, } b > 0$$

$$\text{PDF: } f(x) = \begin{cases} \dfrac{a}{b}\left(\dfrac{x}{b}\right)^{a-1} \exp\left[-\left(\dfrac{x}{b}\right)^{a}\right] & \text{for } 0 \le x < \infty \\ 0 & \text{otherwise} \end{cases}$$

As shown earlier in this chapter, random variates X from the Weibull distribution may be obtained by applying the inverse transform method to the corresponding PDF which results in the equation below.

$$x = b(-\log(1-u))^{1/a}$$

The following code illustrates how the Weibull distribution may be implemented in C#.

```
public static double WeibullPDF(double x, double a, double b)
{
    if (x > 0.0)
        return (a/b)*Math.Pow(x/b,a-1)*Math.Exp(-Math.Pow(x/b,a));
    else
        return 0.0;
}

public static double[] WeibullPDF(double[] x, double a, double b)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    { tempArr[i] = WeibullPDF(x[i], a, b); }
    return tempArr;
}

public static double NextWeibull(double a, double b)
{
    return b*Math.Pow(-Math.Log(randObj.NextDouble()),1.0/a);
}

public static double[] NextWeibull(double a, double b, int nLen)
{
    double[] tempArr = new double[nLen];
    for (int i = 0; i < nLen; i++)
    { tempArr[i] = NextWeibull(a,b); }
    return tempArr;
}

static void TestWeibull()
{
    int nBins = 20; int nPoints = 10000; double xmin = 0.0;
    double xmax = 4.0; double aVal = 2.0; double bVal = 1.0;
    double displayScaleFactor = 500.0;
    double[] randObj = NextWeibull(aVal,bVal,nPoints);
    double[] HistogramValues =
        makeHistogram(randObj, xmin, xmax, nBins);
    double[] xdata = new double[nBins];
    double[] ydata = new double[nBins];
    double[] yCalcDistribution = new double[nBins];
    for (int i = 0; i < nBins; i++)
    {
        xdata[i] = xmin + (i + 0.5) * (xmax - xmin) / nBins;
        ydata[i] = (double)HistogramValues[i];
        yCalcDistribution[i] = WeibullPDF(xdata[i],aVal,bVal);
    }
    double ydataMax = dataMax(ydata);
    double yCalcDistributionMax = dataMax(yCalcDistribution);
    Console.WriteLine("\nTesting the Weibull Distribution");
    Console.WriteLine("\nKey: Bin number = *****... [random distrib.]
         [calculated distrib.]\n");
    for (int i = 0; i < nBins; i++)
    {
      Console.Write("Bin {0,2:n0} = ", i + 1);
      for (int j = 0; j < (Math.Round(ydata[i]*displayScaleFactor /
                nBins / ydataMax, 0)); j++) Console.Write("*");
      Console.WriteLine("  [{0}] [{1}]",Math.Round(ydata[i]*100.0 /
                ydataMax,0),Math.Round(yCalcDistribution[i]*100.0 /
                yCalcDistributionMax,0));
    }
}

Results: Weibull Probability Distribution
Key: Bin number = *****... [random distrib.] [calculated distrib.]

Bin  0 = ******  [22] [23]
Bin  1 = **************  [61] [64]
Bin  2 = *******************  [84] [91]
Bin  3 = ***********************  [100] [100]
Bin  4 = ********************  [90] [93]
Bin  5 = *****************  [73] [76]
Bin  6 = ************  [51] [56]
Bin  7 = ********  [35] [37]
Bin  8 = *****  [21] [22]
Bin  9 = ***  [11] [12]
Bin 10 = *  [6] [6]
Bin 11 = *  [2] [3]
Bin 12 =   [1] [1]
Bin 13 =   [0] [0]
Bin 14 =   [0] [0]
Bin 15 =   [0] [0]
Bin 16 =   [0] [0]
Bin 17 =   [0] [0]
Bin 18 =   [0] [0]
Bin 19 =   [0] [0]
```

10.8 Shuffling Algorithms

Shuffling is usually associated with a procedure used to randomize a deck of playing cards to provide an element of chance in card games. However, in computer science, shuffling is equivalent to generating a random permutation of a given array. There are two basic algorithms for doing this, both popularized by Donald Knuth [50]. The first method starts by assigning a unique random number to each array element. The array is then sorted in order of its random numbers. Since each array element has a unique random number assigned to it, this will generate a random permutation of the given array. The second algorithm, generally known as the *Knuth shuffle* or *Fisher-Yates shuffle*, runs in linear time and is used to shuffle an array in random order. The algorithm loops through each item in the array, generates a random number between 0 and the array length, then assigns the array item to a randomly generated array position. An implementation of the Fisher-Yates algorithm is given below. Note that this method essentially generates a random integer array containing *n* unique elements.

```
//Does a random permutation on a one-dimensional input array
//with input array and number of permutations chosen by the user.
public static int[] RandomPermutation(int[] numbers, int n)
{
    //Exchange each entry of the one-dimensional array with
    //another entry located at a random position in the array.
    for (int i = numbers.Length - 1; i > 0; i--)
    {
        int randomPosition = randObj.Next(i + 1);
        int temp = numbers[i];
        numbers[i] = numbers[randomPosition];
        numbers[randomPosition] = temp;
    }
    return numbers;
}

public static void TestRandomPermutation()
{
    //Setup a simple one-dimensional array for testing permutations
    int arraySize = 6;

    Console.WriteLine("Testing random permutation of array with {0}
        elements\n", arraySize);
    int[] arrayTest = new int[arraySize];
    for (int i = 0; i < arraySize; i++)
    {
        arrayTest[i] = i;
    }
    //and display this one-dimensional array on the screen
    Console.WriteLine("Original input array configuration\n");
    foreach (int j in arrayTest) Console.Write("{0}    ", j);
    Console.WriteLine("\n\n");

    //Do some arbitrary number of random permutations on the input
        array
    int nPermutations = 4;
    Console.WriteLine("Results after performing {0} random
        permutations of original input array\n", nPermutations);
    int k = 0;
    do
    {
        int[] x = RandomPermutation(arrayTest, arraySize);
        foreach (int d in x) Console.Write("{0}    ", d);
        Console.WriteLine("\n");
        k++;
    } while (k < nPermutations);
}
```

```
Results: Testing random permutation of array with 6 elements

Original input array configuration
0    1    2    3    4    5

Results after performing 4 random permutations of the
original input array

4    5    2    3    1    0
2    4    0    1    5    3
3    5    0    2    1    4
1    4    3    2    0    5
```

In more advanced applications, such as those found in some optimization algorithms discussed in Chapter 18, it may be desired to perform small perturbations on individual random elements of a randomly permuted array. In other words, first you generate a random array containing *n* unique elements using the Fisher-Yates algorithm `RandomPermutation(int n)` just described. Then randomly select an element of this array and add a small perturbation on it. The value of this small perturbation is, of course, also randomly selected from some desired probability distribution. As with most randomized algorithms of this type, there are many different ways to do this. One approach illustrated below is to first input a given data array and then use the `RandomPermutation(int n)` algorithm to produce a randomized integer array containing unique integers ranging from 0 to the length of the input data array thereby in essence really obtaining a randomized auxiliary array of the indices of the input data array. Then by selecting any value from this integer array and assigning it as the index of the input data array you will be essentially randomly selecting a single element of the input array and all that is left to do is then add a small perturbation value to it as shown in the code below.

```
//Does a random permutation on a one-dimensional input array
//with a default initialized array and an abitrary number of
//permutations as chosen by the user.
public static int[] RandomPermutation(int n)
{
    //Create an array of size specified by nSize
    int[] numbers = new int[n];
    //to hold the numbers 0,1,2,...,n
    for (int i = 0; i < numbers.Length; i++)
    { numbers[i] = i; }
    //Exchange each entry of the array with another
    //entry located at a random position in the array.
    for (int i = numbers.Length - 1; i > 0; i--)
    {
        int randomPosition = randObj.Next(i + 1);
        int temp = numbers[i];
        numbers[i] = numbers[randomPosition];
        numbers[randomPosition] = temp;
    }
    return numbers;
}
```

```
public static double[] RandomPerturbation(double[] data)
{
    //Randomize the positions of all input data array indices
    int[] randomizedIndicesArray = RandomPermutation(data.Length);
    //then just pick one arbitrary position and add a little
    //amount of perturbation to it. The normal distribution was
    //chosen arbitrarily. You can substitute another probablity
    //distribution in its place if so desired.
    data[randomizedIndicesArray[0]] += NextNormal(0, 1) / 100.0;
    return data;
}

public static void TestRandomPertubation()
{
    //Testing perturbations of entries in a simple one-dimensional
        array
    int arraySize = 6;
    int nPerturbations = 5;
    Console.WriteLine("Testing {0} perturbation(s) on array with {0}
        elements\n", nPerturbations, arraySize);

    double[] arrayTest = new double[arraySize];
    for (int i = 0; i < arraySize; i++)
    {
        arrayTest[i] = i;
    }
    //and display this one-dimensional array on the screen
    Console.WriteLine("Original input array configuration\n");
    foreach (double j in arrayTest)
        Console.Write("{0}    ", j);
    Console.WriteLine("\n\n");

    Console.WriteLine("Results after performing {0} random
        perturbations of original input array\n", nPerturbations);
    //Then perturb the input data array a number of times
    int k = 0;
    do
    {
        //Initialize input data array
        for (int i = 0; i < arraySize; i++)
        {
            arrayTest[i] = i;
        }
        //and then perturb it
        double[] y = RandomPerturbation(arrayTest);
        //and print the output results
        foreach (double c in y)
            Console.Write("{0}    ", c);
        Console.WriteLine("\n");
        k++;
    } while (k < nPerturbations);
    Console.WriteLine("\n\nPress ENTER key to continue...");
    Console.ReadLine();
    Console.Clear();
}
```

```
Results: Testing 5 perturbation(s) on array with 5 elements

Original input array configuration
0    1    2    3    4    5

Results after performing 5 random perturbations of the
original input array

0    1    1.99827856404656    3    4    5
0    1    2    3    4.00305070207981    5
-0.00393216947134168    1    2    3    4    5
0    1    2    2.98306053420896    4    5
0    1    2    2.98561781323784    4    5
```

10.9 Adding Random Noise to Data

The motivation behind adding random noise to data is very simple. Computer simulations of naturally occurring physical phenomena often yield idealized results that may approximate but do not necessarily reflect the actual raw data very accurately. There are two basic approaches to solve or at least minimize this problem. The first method is to work with the raw input data to try and match it to the output data generated by the computer. However, this technique may not always be feasible and sometimes may also prove itself to be impractical. The second approach is to do the complete opposite and try to get the output data generated by the computer to match the raw input data. Therefore, it would be useful to develop a reliable method for adding random noise to data values.

The proposed routine for undertaking this task, `AddNoise(data, magnitude)`, is quite simple. It accepts an array containing the data values to be processed and a variable specifying the magnitude of the noise to be randomly added to those input array values. Inside this routine a random number generator produces random numbers between -1 and 1 which is then multiplied by the desired magnitude value before being added back to the input array. The output consists of an array contining the new modified values.

In support of this process a few auxiliary functions have also been introduced with the intention of facilitating the overall coding process of computing a function between two specific minimum and maximum values. The first routine merely calculates the function value for one point. The second routine is just an overloaded version of the first routine and was designed to process a full array of data points at once. The third routine is also an overloaded version of the first and allows the user to generate an array of data points of arbitrary size and arbitrary range from an arbitrary function. All these routines along with a driver program to illustrate how these tools may be applied to add random noise to data is given below.

```
//Purpose: Add random noise to data

//General function routine to process a single data point by
//some user supplied function. In this example, f(x) = 2x.
public static double f(double x)
{
    return 2 * x;
}

//General routine to process an array of data points provided
//by some arbitrary function.
public static double[] f(double[] x)
{
    double[] tempArr = new double[x.Length];
    for (int i = 0; i < x.Length; i++)
    {
        tempArr[i] = f(x[i]);
    }
    return tempArr;
}

//General routine for calculating a grid of n points (n > 1)
//between a set of minimum and maximum values. It the evaluates
//the user supplied function f(x) at each of those points.
public static void f(int n, double xLeft, double xRight,
                        out double[] x, out double[] y)
{
    double dx = (xRight - xLeft) / (n - 1);

    double[] xtemp = new double[n];
    double[] ytemp = new double[n];

    for (int i = 0; i < n; i++)
    {
        xtemp[i] = xLeft + i * dx;
        ytemp[i] = f(xtemp[i]);
    }

    x = xtemp;
    y = ytemp;
}

//Adds random noise between -magtitude and +magnitude to the data
//in array data[]. Random data source can be easily changed to
//whatever distribution desired.
public static double[] AddNoise(double[] data, double magnitude)
{
    for (int i = 0; i < data.Length; i++)
    {
        //Get a random number between -1 and 1
        double r = 1.0 - 2.0 * randObj.NextDouble();
        data[i] = data[i] + magnitude * r;
    }
    return data;
}
```

```
//Driver routine to test adding random noise to data
public static void TestAddRandomNoiseToData()
{
    Console.WriteLine("Results: Add Random Noise Algorithm\n");
    int n = 20; //Number of grid points
    double xLeft = -5.0; //Leftmost starting point
    double xRight = 5.0; //Rightmost ending point
    double[] x = new double[n];
    double[] y = new double[n];
    //Create the data arrays
    f(n, xLeft, xRight, out x, out y);
    Console.WriteLine("Data BEFORE adding noise\n");
    Console.WriteLine("x values\t y values\n");
    for (int i = 0; i < n; i++)
        Console.WriteLine("{0} \t\t {1}", Math.Round(x[i],3),
                       Math.Round(y[i],3));
    double[] y_noisy = new double[n];
    //Magnitude was purposely made very large for this example
    double noise_magnitude = 10.0;
    //Add some random noise to the data array just created
    y_noisy = AddNoise(y, noise_magnitude);
    Console.WriteLine("\n\nData AFTER adding noise\n");
    Console.WriteLine("x values\t y values\n");
    for (int i = 0; i < n; i++)
        Console.WriteLine("{0} \t\t {1}", Math.Round(x[i], 3),
                       Math.Round(y_noisy[i], 3));
}
```

```
Results: Testing the Add Random Noise Algorithm

Data BEFORE adding noise        Data AFTER adding noise
x values          y values       x values          y values
-5                -10            -5                -11.051
-4.474            -8.947         -4.474            -14.38
-3.947            -7.895         -3.947            -3.398
-3.421            -6.842         -3.421            -6.477
-2.895            -5.789         -2.895            -15.179
-2.368            -4.737         -2.368            -6.888
-1.842            -3.684         -1.842            -4.344
-1.316            -2.632         -1.316            -3.759
-0.789            -1.579         -0.789            -0.048
-0.263            -0.526         -0.263            -9.02
0.263             0.526          0.263             8.445
0.789             1.579          0.789             3.461
1.316             2.632          1.316             9.477
1.842             3.684          1.842             -2.221
2.368             4.737          2.368             12.112
2.895             5.789          2.895             14.464
3.421             6.842          3.421             10.112
3.947             7.895          3.947             1.132
4.474             8.947          4.474             3.356
5                 10             5                 18.495
```

10.10 Removing Random Noise from Data

The concept of removing random noise from data, better known as data smoothing, is not one that meets with universal approval and is often the subject of vigorous debate. For valid least-squares fitting, data smoothing is neither desirable nor permissible. However, there are cases where smoothing may be beneficial to some extent. Generally speaking, if rigorously valid results are not required, but rather an averaged estimate of the distribution, smoothing may help obtain reasonably good rough estimates. The secrets of smoothing data are the subject of the very sophisticated field of filtering and choosing the most optimal filter to use can be a rather tricky and lengthy process. Consequently, we shall examine only the most basic filtering method that is neither the most efficient nor the most effective. However, it is adequate enough and easy to understand for many situations of interest.

The justification usually made for data smoothing is that one is measuring a variable that is both slowly changing and is also corrupted by random noise. Under such conditions, it can sometimes be useful to replace each data point by some kind of local average of the surrounding data points. Since nearby data points measure very nearly the same underlying value, averaging can reduce the level of noise without significantly biasing or adversely affecting the original data. The simplest approach is therefore to create a new array of data points where each point is the average of its neighbors from the original array: $X_{\text{new}}[i] = (x_{\text{old}}[i-1] + x_{\text{old}}[i] + x_{\text{old}}[i+1])/3$. This three-point averaging technique is an example of what is called a *moving average*.

Unfortunately there are some major issues associated with this kind of approach. First, there is no reason why such an average should be restricted to just three neighboring points. The most practical solution is to make the width of the averaging neighborhood an input variable to the procedure. The only restriction to make is that the number of points used must be odd, since otherwise we would have to use a different number of neighbors to the right than we do to the left of the points. The second important issue to consider is what to do about the endpoints of the array since taking an average of the neighboring points at that location might entail handling points that lie beyond the scope of the original input array. Just how many such points exist obviously depend on the chosen width of the averaging neighborhood. Possible solutions to this problem include making up values for such non-existent points, assigning the value of 0 to those points, setting them all equal to the border points of the original array, or perhaps not using them at all. However, such approaches are effectively making up data which is rather distasteful and unethical. Another more plausible solution, and the one which we will use, is to simply take a smaller average of the endpoints of the array: $X_{\text{new}}[0] = (x_{\text{old}}[0] + x_{\text{old}}[1])/2$. This way the actual endpoint is still a simple average of its neighbors.

The actual approach used here is to create variable width average by using a nested loop, so that instead of averaging only three neighboring points, we can average a number of points determined at the time of program execution. The auxiliary variables `start` and `stop` control the limits of the inner loop. The auxiliary functions `FirstIndex` and `LastIndex` are used to calculate the necessary values of `start` and `stop` for any particular value of `i`. The code below illustrates how all these ideas may be implemented in C#.

```
//Purpose: Filter random noise from data

//Returns the index <= i at which signal values can start to
//contribute to the filtered value of data element i.
public static int FirstIndex(int i, int width)
{
    if (i >= width / 2)
        return (i - width / 2);
    else
        return 0;
}

//Returns the index >= i beyond which signal values cease to
//contribute to the filtered value of data element i.
public static int LastIndex(int i, int width, int size)
{
    if (i + width / 2 < size)
        return (i + width / 2);
    else
        return (size - 1);
}

//Filters the input signal using a moving average algorithm
//of specified width. The width must be odd.
public static double[] MovingAverage(double[] RoughData, int width)
{
    double[] y = new double[RoughData.Length];
    if (width % 2 == 0)
    {
        Console.WriteLine("Error: The width must be odd.");
        return y;
    }

    for (int i = 0; i < RoughData.Length; i++)
    {
        y[i] = 0.0;
        int start = FirstIndex(i, width);
        int stop = LastIndex(i, width, RoughData.Length);

        for (int j = start; j <= stop; j++)
        {
            y[i] = y[i] + RoughData[j];
        }
        y[i] = y[i] / (stop - start + 1);
    }
    return y;
}
```

```
//Driver routine to test filtering random noise from data
//using a moving average algorithm.
private static void TestRemoveRandomNoiseFromData()
{
  double[] RoughData =
    new double[] {  75.475, 75.600, 75.100, 73.525, 73.275,
    73.325, 73.225, 74.350, 73.600, 74.475, 75.975, 76.850,
    77.525, 78.200, 79.225, 78.850, 76.225, 76.850, 76.725, 76.100};

  Console.WriteLine("Results: Data Filtering Algorithm\n");

  Console.WriteLine("Rough Data\n");
  for (int i = 0; i < RoughData.Length; i++)
       Console.WriteLine("{0}", Math.Round(RoughData[i], 3));

  //Width of filter
  int width = 5;

  //Array to hold the filtered data
  double[] SmoothData = new double[RoughData.Length];

  //Calculate the filtered data values using a moving average
  //algorithm of the specified rough data and filter width.
  SmoothData = MovingAverage(RoughData, width);

  //Display the results on the screen or save the results
  //to a file right here.
  Console.WriteLine("\n\nSmoothed Data\n");
  for (int i = 0; i < RoughData.Length; i++)
      Console.WriteLine("{0}", Math.Round(SmoothData[i], 3));
  Console.WriteLine("\nPress ENTER key to continue...");
  Console.ReadLine(); Console.Clear();
}
```

```
Results: Data Filter Algorithm
Rough Data      Filtered Data
75.475          75.392
75.6            74.925
75.1            74.595
73.525          74.165
73.275          73.69
73.325          73.54
73.225          73.555
74.35           73.795
73.6            74.325
74.475          75.05
75.975          75.685
76.85           76.605
77.525          77.555
78.2            78.13
79.225          78.005
78.85           77.87
76.225          77.575
76.85           76.95
76.725          76.475
76.1            76.558
```

11

Numerical Differentiation

11.1 Introduction

Numerical differentiation is a technique that seeks to find an estimate of the numerical value of a derivative at a point of some given function using only values from the function and perhaps other information but without using the analytical form of the function itself which may not even be known. More formally, for a given function $y = f(x)$ we seek to calculate the derivatives of $f(x)$ defined on a discrete finite set of grid points $(x_0, x_1, x_2, \ldots, x_N)$. It is assumed that the only data available for this calculation are the exact values of the function at the data points: $(x_i, y_i = f(x_i))$, and that the derivatives are sought only at the discrete finite set of data points (x_i, y_i).

Calculating derivatives numerically is generally not greeted with much enthusiasm among mathematicians. This is primarily due to a lack of very precise numerical methods within which to carry out precise derivative calculations. In addition, higher precision calculations often require increasingly more data points which may not always be directly available. Nevertheless, there are two basic approaches to calculating numerical derivatives. The finite difference formalism allows us to calculate both first and higher order derivatives of a function using only discrete sets of adjacent data points called a grid. This method also uses a Taylor series expansion which has the additional advantage of also providing some estimate of the amount of error inevitably incurred in such calculations. With the finite difference method, the higher number of data points you use and the closer they are to each other, the better the precision of the results. The second approach allows discrete data points to be generated at uneven intervals of x and involves approximating the function locally using polynomial interpolation followed by polynomial differentiation.

11.2 Finite Difference Formulas

Assuming that we have an equidistant grid, meaning that the distance between adjacent points $\Delta x = h$ is constant, there are three ways to calculate differences:

$$\text{Forward}: \Delta x = x_{i+1} - x_i \quad \text{Backward}: \Delta x = x_i - x_{i-1} \quad \text{and} \quad \text{Central}: \Delta x = x_{i+1} - x_{i-1}$$

The finite difference approximations for the derivatives of $f(x)$ is based on the forward and backward Taylor series expansions of $f(x)$ about x as shown below [69, 70]:

$$
\begin{aligned}
f(x+h) &= f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \cdots + \frac{h^n}{n!}f^{(n)}(x) + R_n(x) \\
f(x-h) &= f(x) - hf'(x) + \frac{h^2}{2!}f''(x) - \frac{h^3}{3!}f'''(x) + \cdots + \frac{h^n}{n!}f^{(n)}(x) + R_n(x) \\
f(x+2h) &= f(x) + (2h)f'(x) + \frac{(2h)^2}{2!}f''(x) + \frac{(2h)^3}{3!}f'''(x) + \cdots + \frac{(2h)^n}{n!}f^{(n)}(x) + R_n(x) \\
f(x-2h) &= f(x) - (2h)f'(x) + \frac{(2h)^2}{2!}f''(x) - \frac{(2h)^3}{3!}f'''(x) + \cdots + \frac{(2h)^n}{n!}f^{(n)}(x) + R_n(x)
\end{aligned}
$$

where $R_n(x)$ is a remainder term and $h = \Delta x$. In addition, by adding and subtracting the equations above we can obtain the sums and differences of the Taylor series expansion as shown below.

$$
\begin{aligned}
f(x+h) + f(x-h) &= 2f(x) + h^2 f''(x) + \frac{h^4}{12}f^{(4)}(x) + \cdots \\
f(x+h) - f(x-h) &= 2hf'(x) + \frac{h^3}{3}f^{(3)}(x) + \cdots \\
f(x+2h) + f(x+2h) &= 2f(x) + 4h^2 f''(x) + \frac{4h^4}{3}f^{(4)}(x) + \cdots \\
f(x+2h) - f(x+2h) &= 4hf'(x) + \frac{8h^3}{3}f^{(3)}(x) + \cdots
\end{aligned}
$$

The intuitive but rather naive and ultimately wrong approach is then to start truncating terms of order h^2 and higher in order to obtain the following first order approximation for the derivative of $f(x)$:

$$
f'(x) \approx \frac{f(x+h) - f(x)}{h}
$$

This problem can be exacerbated even further by carrying on derivations in a similar manner for higher order derivatives. Unfortunately, this rather simplistic approach to approximating numerical derivatives yields very inaccurate results because of a large inherent truncation error in the order of $O(h)$. Instead, by retaining increasingly higher order terms in the Taylor series expansion of $f(x)$, we can obtain increasingly accurate results for calculating numerical derivatives of $f(x)$ to any chosen order and desired degree of precision. The common practice, however, is to use truncation errors at least in the order of $O(h^2)$.

As these derivations are long and tedious, I will simply list the final results here and refer those who are interested in the gritty details to seek them in just about any textbook on numerical analysis [69, 70].

11.2.1 Forward Difference Method

The finite forward difference differentiation formulas, where $\Delta x = h = x_{i+1} - x_i$, of order $O(h^2)$ are given by:

$$f'(x) \approx \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h}$$
$$f''(x) \approx \frac{2f(x) - 5f(x+h) + 4f(x+2h) - f(x+3h)}{h^2}$$
$$f^{(3)}(x) \approx \frac{-5f(x) + 18f(x+h) - 24f(x+2h) + 14f(x+3h) - 3f(x+4h)}{2h^3}$$
$$f^{(4)}(x) \approx \frac{3f(x) - 14f(x+h) + 26f(x+2h) - 24f(x+3h) + 11f(x+4h) - 2f(x+5h)}{h^4}$$

The implementation for these forward difference formulas in C# is given below and includes methods for approximating numerical derivatives for both an analytical function $f(x)$ and for an array of (x_i, y_i) of data values.

```
public static double DerivativeForward1(Function f,double x,double h)
{
   h=(h==0)?0.01:h;
   return (-3*f(x) + 4*f(x+h) - f(x+2*h))/2/h;
}

public static double DerivativeForward1(double[] y,int i,double h)
{
   if (y==null || y.Length<3 || i<0 || i > y.Length-3 || h==0)
       return double.NaN;
   return (-3*y[i] + 4*y[i+1] - y[i+2])/2/h;
}

public static double DerivativeForward2(Function f,double x,double h)
{
   h=(h==0)?0.01:h;
   return (2*f(x) - 5*f(x+h) + 4*f(x+2*h) - f(x+3*h))/h/h;
}

public static double DerivativeForward2(double[] y,int i,double h)
{
   if (y==null || y.Length<4 || i<0 || i>y.Length-4 || h==0)
       return double.NaN;
   return (2*y[i] - 5*y[i+1] + 4*y[i+2] - y[i+3])/h/h;
}

public static double DerivativeForward3(Function f,double x,double h)
{
   h=(h==0)?0.01:h;
   return (-5*f(x) + 18*f(x+h) - 24*f(x+2*h) + 14*f(x+3*h) -
            3*f(x+4*h))/2/h/h/h;
}
```

```
public static double DerivativeForward3(double[] y,int i,double h)
{
   if (y==null || y.Length<5 || i<0 || i>y.Length-5 || h==0)
        return double.NaN;
   return (-5*y[i] + 18*y[i+1] - 24*y[i+2] + 14*y[i+3] -
            3*y[i+4])/2/h/h/h;
}

public static double DerivativeForward4(Function f,double x,double h)
{
   h=(h==0)?0.01:h;
   return (3*f(x) - 14*f(x+h) + 26*f(x+2*h) - 24*f(x+3*h) +
          11*f(x+4*h) - 2*f(x+5*h))/h/h/h/h;
}

public static double DerivativeForward4(double[] y,int i,double h)
{
 if (y==null || y.Length<6 || i<0 || i>y.Length-6 || h==0)
        return double.NaN;
 return (3*y[i] - 14*y[i+1] + 26*y[i+2] - 24*y[i+3] + 11*y[i+4]
         - 2*y[i+5])/h/h/h/h;
}
```

To test the forward difference method for calculating numerical derivative values I used the test function $f(x) = \cos x$ and calculated derivatives at the point $x = 0.3$ as shown below.

```
public delegate double Function(double x);

static double f1(double x)
{
   return Math.Cos(x);
}

static void Main(string[] args)
{
    Console.WriteLine("Using the FORWARD difference method
        and function: f(x) = cos(x)\n");
    double dy = DerivativeForward1(f1, 0.3, h);
    Console.WriteLine(" f'(x) = " + dy.ToString());
    dy = DerivativeForward2(f1, 0.3, h);
    Console.WriteLine(" f''(x) = " + dy.ToString());
    dy = DerivativeForward3(f1, 0.3, h);
    Console.WriteLine(" f'''(x) = " + dy.ToString());
    dy = DerivativeForward4(f1, 0.3, h);
    Console.WriteLine(" f''''(x) = " + dy.ToString());

    // Calculate derivatives using just array values at x=3:
    Console.WriteLine("\nDerivatives using just array values:\n");
    double h = 0.1;
    double[] y=new double[10];
    for (int i = 0; i < 10; i++)
        y[i] = f1(i * h);
    dy = DerivativeForward1(y, 3, h);
    Console.WriteLine(" y' = " + dy.ToString());
    dy = DerivativeForward2(y, 3, h);
    Console.WriteLine(" y'' = " + dy.ToString());
    dy = DerivativeForward3(y, 3, h);
    Console.WriteLine(" y''' = " + dy.ToString());
    dy = DerivativeForward4(y, 3, h);
    Console.WriteLine(" y'''' = " + dy.ToString());
}

OUTPUT:

Using the FORWARD difference method and the function
f(x) = cos(x) to calculate derivatives at x = 0.3

 f'(0.3)    = -0.296740266278253
 f''(0.3)   = -0.963735911140073
 f'''(0.3)  =  0.303003968494142
 f''''(0.3) =  0.980440888838086

Derivatives using just array values. Since h=0.1 then
the point x=0.3 is equivalent to i=3.

 y'(0.3)    = -0.296740266278253
 y''(0.3)   = -0.963735911140073
 y'''(0.3)  =  0.303003968494142
 y''''(0.3) =  0.980440888820322
```

11.2.2 Backward Difference Method

The finite backward difference differentiation formulas, where $\Delta x = h = x_i - x_{i-1}$, of order $O(h^2)$ are given by

$$
\begin{aligned}
f'(x) &\approx \frac{3f(x) - 4f(x-h) + f(x-2h)}{2h} \\
f''(x) &\approx \frac{2f(x) - 5f(x-h) + 4f(x-2h) - f(x-3h)}{h^2} \\
f^{(3)}(x) &\approx \frac{5f(x) - 18f(x-h) + 24f(x-2h) - 14f(x-3h) + 3f(x-4h)}{2h^3} \\
f^{(4)}(x) &\approx \frac{3f(x) - 14f(x-h) + 26f(x-2h) - 24f(x-3h) + 11f(x-4h) - 2f(x-5h)}{h^4}
\end{aligned}
$$

The implementation for these backward difference formulas in C# is given below and includes methods for approximating numerical derivatives for both an analytical function $f(x)$ and for an array of (x_i, y_i) of data values.

```
public static double DerivativeBackward1(Function f, double x,
        double h)
{
   h=(h==0)?0.01:h;
   return (3*f(x) - 4*f(x-h) + f(x-2*h))/2/h;
}
```

```
public static double DerivativeBackward1(double[] y, int i,
        double h)
{
   if (y==null || y.Length<3 || i<0 || i<3 || h==0)
       return double.NaN;
   return (3*y[i] - 4*y[i-1] + y[i-2])/2/h;
}

public static double DerivativeBackward2(Function f, double x,
        double h)
{
   h=(h==0)?0.01:h;
   return (2*f(x) - 5*f(x-h) + 4*f(x-2*h) - f(x-3*h))/h/h;
}

public static double DerivativeBackward2(double[] y,int i,
        double h)
{
   if (y==null || y.Length<4 || i<0 || i<4 || h==0)
       return double.NaN;
   return (2*y[i] - 5*y[i-1] + 4*y[i-2] - y[i-3])/h/h;
}

public static double DerivativeBackward3(Function f, double x,
        double h)
{
   h=(h==0)?0.01:h;
   return (5*f(x) - 18*f(x-h) + 24*f(x-2*h) - 14*f(x-3*h)
         + 3*f(x-4*h))/2/h/h/h;
}

public static double DerivativeBackward3(double[] y, int i,
        double h)
{
   if (y==null || y.Length<5 || i<0 || i<5 || h==0)
       return double.NaN;
   return (5*y[i] - 18*y[i-1] + 24*y[i-2] - 14*y[i-3]
         + 3*y[i-4])/2/h/h/h;
}

public static double DerivativeBackward4(Function f, double x,
        double h)
{
   h=(h==0)?0.01:h;
   return (3*f(x) - 14*f(x-h) + 26*f(x-2*h) - 24*f(x-3*h) +
          11*f(x-4*h) - 2*f(x-5*h))/h/h/h/h;
}

public static double DerivativeBackward4(double[] y, int i,
        double h)
{
   if (y==null || y.Length<6 || i<0 || i<6 || h==0)
       return double.NaN;
   return (3*y[i] - 14*y[i-1] + 26*y[i-2] - 24*y[i-3] +
          11*y[i-4] - 2*y[i-5])/h/h/h/h;
}
```

To test the backward difference method for calculating numerical derivative values I used the test function $f(x) = \cos x$ and calculated derivatives at the point $x = 0.7$ as shown below.

```
public delegate double Function(double x);

static double f1(double x)
{
   return Math.Cos(x);
}

static void Main(string[] args)
{
    // Calculate derivatives using f(x) = cos(x) at x=0.7:
    Console.WriteLine("\nUsing the BACKWARD difference method\n");
    double dy = DerivativeBackward1(f1, 0.7, h);
    Console.WriteLine(" f'(x) = " + dy.ToString());
    dy = DerivativeBackward2(f1, 0.7, h);
    Console.WriteLine(" f''(x) = " + dy.ToString());
    dy = DerivativeBackward3(f1, 0.7, h);
    Console.WriteLine(" f'''(x) = " + dy.ToString());
    dy = DerivativeBackward4(f1, 0.7, h);
    Console.WriteLine(" f''''(x) = " + dy.ToString());
    // Calculate derivatives for array values at i = 7:
    Console.WriteLine("\nDerivatives for array values:\n");
    double h = 0.1;
    double[] y = new double[10];
    for (int i = 0; i < 10; i++)
        y[i] = f1(i * h);
    dy = DerivativeBackward1(y, 7, h);
    Console.WriteLine(" y' = " + dy.ToString());
    dy = DerivativeBackward2(y, 7, h);
    Console.WriteLine(" y'' = " + dy.ToString());
    dy = DerivativeBackward3(y, 7, h);
    Console.WriteLine(" y''' = " + dy.ToString());
    dy = DerivativeBackward4(y, 7, h);
    Console.WriteLine(" y'''' = " + dy.ToString());
}

OUTPUT:
Using the BACKWARD difference method and the function
f(x) = cos(x) to calculate derivatives at x = 0.7

 f'(0.7)    = -0.646166679474376
 f''(0.7)   = -0.772444642080838
 f'''(0.7)  =  0.653452376802477
 f''''(0.7) =  0.789308976041347

Derivatives for array values. Since h=0.1 then
the point x=0.7 is equivalent to i=7.

 y'(0.7)    = -0.646166679474374
 y''(0.7)   = -0.772444642080794
 y'''(0.7)  =  0.653452376804253
 y''''(0.7) =  0.789308976023584
```

11.2.3 Central Difference Method

The finite central difference differentiation formulas, where $\Delta x = h = x_{i+1} - x_{i-1}$, of order $O(h^2)$ are given by

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$
$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}$$
$$f^{(3)}(x) \approx \frac{-f(x-2h) + 2f(x-h) - 2f(x+h) + f(x+2h)}{2h^3}$$
$$f^{(4)}(x) \approx \frac{f(x-2h) - 4f(x-h) + 6f(x) - 4f(x+h) + f(x+2h)}{h^4}$$

The implementation for these central difference formulas in C# is given below and includes methods for approximating numerical derivatives for both an analytical function $f(x)$ and for an array of (x_i, y_i) of data values.

```
public static double DerivativeCentral1(Function f,double x,double h)
{
   h=(h==0)?0.01:h;
   return (f(x+h) - f(x-h))/2/h;
}

public static double DerivativeCentral1(double[] y,int i,double h)
{
   if (y==null || y.Length<3 || i<1 || i>y.Length-2 || h==0)
        return double.NaN;
   return (y[i+1] - y[i-1])/2/h;
}

public static double DerivativeCentral2(Function f,double x,double h)
{
   h=(h==0)?0.01:h;
   return (f(x-h) - 2*f(x) + f(x+h))/h/h;
}

public static double DerivativeCentral2(double[] y,int i,double h)
{
   if (y==null || y.Length<3 || i<1 || i>y.Length-2 || h==0)
        return double.NaN;
   return (y[i-1] - 2*y[i] + y[i+1])/h/h;
}

public static double DerivativeCentral3(Function f,double x,double h)
{
   h=(h==0)?0.01:h;
   return (-f(x-2*h) + 2*f(x-h) - 2*f(x+h) + f(x+2*h))/2/h/h/h;
}
```

```
public static double DerivativeCentral3(double[] y,int i,double h)
{
   if (y==null || y.Length<5 || i<2 || i>y.Length-3 || h==0)
       return double.NaN;
   return (-y[i-2] + 2*y[i-1] - 2*y[i+1] + y[i+2])/2/h/h/h;
}

public static double DerivativeCentral4(Function f,double x,double h)
{
   h=(h==0)?0.01:h;
   return (f(x-2*h)-4*f(x-h)+6*f(x)-4*f(x+h)+f(x+2*h))/h/h/h/h;
}

public static double DerivativeCentral4(double[] y,int i,double h)
{
   if (y==null || y.Length<5 || i<2 || i >y.Length-3 || h==0)
       return double.NaN;
   return (y[i-2] - 4*y[i-1] + 6*y[i] - 4*y[i+1]
         + y[i+2])/h/h/h/h;
}
```

To test the central difference method for calculating numerical derivative values I used the test function $f(x) = \cos x$ and calculated derivatives at the point $x = 0.5$ as shown below.

```
public delegate double Function(double x);
static double f1(double x)
{ return Math.Cos(x); }

static void Main(string[] args)
{
    // Calculate derivatives using f(x) = cos(x) at x=0.5:
    Console.WriteLine("\nUsing the CENTRAL difference method\n");
    double dy = DerivativeCentral1(f1, 0.5, h);
    Console.WriteLine(" f′(x) = " + dy.ToString());
    dy = DerivativeCentral2(f1, 0.5, h);
    Console.WriteLine(" f′′(x) = " + dy.ToString());
    dy = DerivativeCentral3(f1, 0.5, h);
    Console.WriteLine(" f′′′(x) = " + dy.ToString());
    dy = DerivativeCentral4(f1, 0.5, h);
    Console.WriteLine(" f′′′′(x) = " + dy.ToString());
    // Calculate derivatives for array values at i = 5:
    Console.WriteLine("\nDerivatives for array values:\n");
    double h = 0.1;
    double[] y = new double[10];
    for (int i = 0; i < 10; i++)
        y[i] = f1(i * h);
    dy = DerivativeCentral1(y, 5, h);
    Console.WriteLine(" y′ = " + dy.ToString());
    dy = DerivativeCentral2(y, 5, h);
    Console.WriteLine(" y′′ = " + dy.ToString());
    dy = DerivativeCentral3(y, 5, h);
    Console.WriteLine(" y′′′ = " + dy.ToString());
    dy = DerivativeCentral4(y, 5, h);
    Console.WriteLine(" y′′′′ = " + dy.ToString());
}
```

```
OUTPUT:

Using the CENTRAL difference method and the function
f(x) = cos(x) to calculate derivatives at x = 0.5

 f'(x) = -0.478626895466034
 f''(x) = -0.876851486818209
 f'''(x) = 0.478228172648032
 f''''(x) = 0.876121020770837

Derivatives for array values. Since h=0.1 then
the point x=0.5 is equivalent to i=5.

 y' = -0.478626895466034
 y'' = -0.87685148681822
 y''' = 0.478228172648087
 y'''' = 0.876121020774168
```

11.2.4 Improved Central Difference Method

The finite difference methods presented so far for calculating the first four derivatives of a function $f(x)$ have an estimated precision in the order of $O(h^2)$. By using more terms in the Taylor series expansion of the original $f(x)$ as described at the beginning of this chapter, it is possible to calculate derivatives with increasingly improved precision albeit at a cost of larger and messier equations. For example, using the central difference method with an estimated precision in the order of $O(h^4)$, the first four derivatives of a function $f(x)$ are given by [70]:

$$
\begin{aligned}
f'(x) &\approx \frac{f(x-2h)-8f(x-h)+8f(x+h)-f(x+2h)}{12h} \\
f''(x) &\approx \frac{-f(x-2h)+16f(x-h)-30f(x)+16f(x+h)-f(x+2h)}{12h^2} \\
f^{(3)}(x) &\approx \frac{f(x-3h)-8f(x-2h)+13f(x-h)-13f(x+h)+8f(x+2h)-f(x+3h)}{8h^3} \\
f^{(4)}(x) &\approx \frac{-f(x-3h)+12f(x-2h)-39f(x-h)+56f(x)-39f(x+h)}{6h^4} \\
&\quad + \frac{12f(x+2h)-f(x+3h)}{6h^4}
\end{aligned}
$$

The implementation for the higher order, $O(h^4)$, difference formulas in C# is given below and includes methods for approximating numerical derivatives for both an analytical function $f(x)$ and for an array of (x_i, y_i) of data values.

```
public static double DerivativeOh4Central1(Function f, double x,
        double h)
{
   h=(h==0)?0.01:h;
   return (f(x-2*h) - 8*f(x-h) + 8*f(x+h) - f(x+2*h))/12/h;
}

public static double DerivativeOh4Central1(double[] y, int i,
        double h)
{
   if (y==null || y.Length<5 || i<2 || i>y.Length-3 || h==0)
       return double.NaN;
   return (y[i-2] - 8*y[i-1] + 8*y[i+1] - y[i+2])/12/h;
}

public static double DerivativeOh4Central2(Function f, double x,
        double h)
{
   h=(h==0)?0.01:h;
   return (-f(x-2*h)+16*f(x-h)-30*f(x)+16*f(x+h)-f(x+2*h))/12/h/h;
}

public static double DerivativeOh4Central2(double[] y, int i,
        double h)
{
   if (y==null || y.Length<5 || i<2 || i>y.Length-3 || h==0)
       return double.NaN;
   return (-y[i-2]+16*y[i-1]-30*y[i]+16*y[i+1]-y[i+2])/12/h/h;
}

public static double DerivativeOh4Central3(Function f, double x,
        double h)
{
   h=(h==0)?0.01:h;
   return (f(x-3*h) - 8*f(x-2*h) + 13*f(x-h) - 13*f(x+h)
       + 8*f(x+2*h) - f(x+3*h))/8/h/h/h;
}

public static double DerivativeOh4Central3(double[] y, int i,
        double h)
{
   if (y==null || y.Length<7 || i<3 || i>y.Length-4 || h==0)
       return double.NaN;
   return (y[i-3] - 8*y[i-2] + 13*y[i-1] - 13*y[i+1]
       + 8*y[i+2] - y[i+3])/8/h/h/h;
}


public static double DerivativeOh4Central4(Function f, double x,
        double h)
{
   h=(h==0)?0.01:h;
   return (-f(x-3*h) + 12*f(x-2*h) - 39*f(x-h) + 56*f(x)
       - 39*f(x+h) + 12*f(x+2*h) - f(x+3*h))/6/h/h/h/h;
}
```

```
public static double DerivativeOh4Central4(double[] y, int i,
        double h)
{
   if (y==null || y.Length<7 || i<3 || i>y.Length-4 || h==0)
       return double.NaN;
   return (-y[i-3] + 12*y[i-2] - 39*y[i-1] + 56*y[i]
       - 39*y[i+1] + 12*y[i+2] - y[i+3])/6/h/h/h/h;
}
```

To test the improved central method for calculating numerical derivative values I used the test function $f(x) = \cos x$ and calculated derivatives at the point $x = 0.5$ as shown below.

```
public delegate double Function(double x);
static double f1(double x)
{ return Math.Cos(x); }

static void Main(string[] args)
{
  // Calculate derivatives using f(x) = cos(x) at x=0.5:
  Console.WriteLine("\nUsing the O(h^4)CENTRAL difference method\n");
  double dy = DerivativeOh4Central1(f1, 0.5, h);
  Console.WriteLine(" f'(x) = " + dy.ToString());
  dy = DerivativeOh4Central2(f1, 0.5, h);
  Console.WriteLine(" f''(x) = " + dy.ToString());
  dy = DerivativeOh4Central3(f1, 0.5, h);
  Console.WriteLine(" f'''(x) = " + dy.ToString());
  dy = DerivativeOh4Central4(f1, 0.5, h);
  Console.WriteLine(" f''''(x) = " + dy.ToString());
  // Calculate derivatives for array values at i = 5:
  Console.WriteLine("\nDerivatives for array values:\n");
  double h = 0.1;
  double[] y = new double[10];
  for (int i = 0; i < 10; i++)
        y[i] = f1(i * h);
  dy = DerivativeOh4Central1(y, 5, h);
  Console.WriteLine(" y' = " + dy.ToString());
  dy = DerivativeOh4Central2(y, 5, h);
  Console.WriteLine(" y'' = " + dy.ToString());
  dy = DerivativeOh4Central3(y, 5, h);
  Console.WriteLine(" y''' = " + dy.ToString());
  dy = DerivativeOh4Central4(y, 5, h);
  Console.WriteLine(" y'''' = " + dy.ToString());
  // Analytic results:
  Console.WriteLine("\nAnalytic results:\n");
  dy = -Math.Sin(0.5);
  Console.WriteLine(" y' = " + dy.ToString());
  dy = -Math.Cos(0.5);
  Console.WriteLine(" y'' = " + dy.ToString());
  dy = Math.Sin(0.5);
  Console.WriteLine(" y''' = " + dy.ToString());
  dy = Math.Cos(0.5);
  Console.WriteLine(" y'''' = " + dy.ToString());
}
```

```
OUTPUT:
Using the O(h^4) CENTRAL difference method and the function
f(x) = cos(x) to calculate derivatives at x = 0.5

 f'(0.5)    = -0.479423942420447
 f''(0.5)   = -0.877581587668858
 f'''(0.5)  =  0.47942274710315
 f''''(0.5) =  0.877580006044357

Derivatives for array values. Since h=0.1 then
the point x=0.5 is equivalent to i=5.

 y'(0.5)    = -0.479423942420448
 y''(0.5)   = -0.877581587668872
 y'''(0.5)  =  0.479422747103261
 y''''(0.5) =  0.8775800060562

Analytic results for comparision:

 y'(0.5)    = -0.479425538604203
 y''(0.5)   = -0.877582561890373
 y'''(0.5)  =  0.479425538604203
 y''''(0.5) =  0.877582561890373
```

11.3 Richardson Extrapolation

Richardson extrapolation is a sequence acceleration method used to improve the rate of convergence of a sequence. It is named after Lewis Fry Richardson, who introduced the technique in the early 20th century. Practical applications of Richardson extrapolation include Romberg integration, which applies Richardson extrapolation to the trapezoidal rule, and the Bulirsch-Stoer algorithm for solving ordinary differential equations [69, 70].

Suppose that $A(h)$ is an estimation of order h^n for A so that $A = \lim_{h\to 0} A(h)$. In other words, $A - A(h) = a_n h^n + O(h^m)$, $a_n \neq 0$, $m > n$. By repeating this calculation with $h = th$ we obtain $A - A(th) = a_n(th)^n + O((th)^m)$. Using these two equations to eliminate the coefficient a_n and solving for A we eventually arrive at the following expression.

$$A = \frac{t^n A(h) - A(th)}{t^n - 1}$$

which is called the Richardson extrapolation formula. In practice, t is usually set to $t = 1/2$ in which case the above expression becomes

$$A = \frac{2^n A(h/2) - A(h)}{2^n - 1}$$

which gives an estimate of order h^m for A with $m > n$.

Since the derivatives are calculated in the order of $O(h^2)$ using the finite difference methods, then for our purposes $n = 2$ and the Richardson extrapolation formula reduces to

$$A = \frac{4A(h/2) - A(h)}{3}$$

The implementation of the Richardson extrapolation method in the calculation of numerical derivatives using the finite difference methods described earlier in this chapter for both an analytical function $f(x)$ and for an array of (x_i, y_i) of data values is illustrated below.

```
public static double DerivativeRichardson1(Function f, double x,
    double h, string flag)
{
    double result = double.NaN;
    switch (flag)
    {
        case "Backward":
            result = (4 * DerivativeBackward1(f, x, h / 2) -
                DerivativeBackward1(f, x, h)) / 3;
        break;
        case "Forward":
            result = (4 * DerivativeForward1(f, x, h / 2) -
                DerivativeForward1(f, x, h)) / 3;
        break;
        case "Central":
            result = (4 * DerivativeCentral1(f, x, h / 2) -
                DerivativeCentral1(f, x, h)) / 3;
        break;
        default:
            result = double.NaN;
        break;
    }
    return result;
}

public static double DerivativeRichardson1(double[] y, int i,
        double h, string flag)
{
    double result = double.NaN;
    switch (flag)
    {
        case "Backward":
            result = (4 * DerivativeBackward1(y, i, h / 2) -
                DerivativeBackward1(y, i, h)) / 3;
            break;
        case "Forward":
            result = (4 * DerivativeForward1(y, i, h / 2) -
                DerivativeForward1(y, i, h)) / 3;
            break;
        case "Central":
            result = (4 * DerivativeCentral1(y, i, h / 2) -
                DerivativeCentral1(y, i, h)) / 3;
            break;
        default:
            result = double.NaN;
            break;
    }
    return result;
}

public static double DerivativeRichardson2(Function f, double x,
    double h, string flag)
{
    double result = double.NaN;
    switch (flag)
    {
        case "Backward":
            result = (4 * DerivativeBackward2(f, x, h / 2) -
                DerivativeBackward2(f, x, h)) / 3;
            break;
        case "Forward":
            result = (4 * DerivativeForward2(f, x, h / 2) -
                DerivativeForward2(f, x, h)) / 3;
            break;
        case "Central":
            result = (4 * DerivativeCentral2(f, x, h / 2) -
                DerivativeCentral2(f, x, h)) / 3;
            break;
        default:
            result = double.NaN;
            break;
    }
    return result;
}

public static double DerivativeRichardson2(double[] y, int i, double
    h, string flag)
{
    double result = double.NaN;
    switch (flag)
    {
        case "Backward":
            result = (4 * DerivativeBackward2(y, i, h / 2) -
                DerivativeBackward2(y, i, h)) / 3;
            break;
        case "Forward":
            result = (4 * DerivativeForward2(y, i, h / 2) -
                DerivativeForward2(y, i, h)) / 3;
            break;
        case "Central":
            result = (4 * DerivativeCentral2(y, i, h / 2) -
                DerivativeCentral2(y, i, h)) / 3;
            break;
        default:
            result = double.NaN;
            break;
    }
    return result;
}
```

```
public static double DerivativeRichardson3(Function f, double x,
    double h, string flag)
{
    double result = double.NaN;
    switch (flag)
    {
        case "Backward":
            result = (4 * DerivativeBackward3(f, x, h / 2) -
                DerivativeBackward3(f, x, h)) / 3;
            break;
        case "Forward":
            result = (4 * DerivativeForward3(f, x, h / 2) -
                DerivativeForward3(f, x, h)) / 3;
            break;
        case "Central":
            result = (4 * DerivativeCentral3(f, x, h / 2) -
                DerivativeCentral3(f, x, h)) / 3;
            break;
        default:
            result = double.NaN;
            break;
    }
    return result;
}

public static double DerivativeRichardson3(double[] y, int i, double
    h, string flag)
{
    double result = double.NaN;
    switch (flag)
    {
        case "Backward":
            result = (4 * DerivativeBackward3(y, i, h / 2) -
                DerivativeBackward3(y, i, h)) / 3;
            break;
        case "Forward":
            result = (4 * DerivativeForward3(y, i, h / 2) -
                DerivativeForward3(y, i, h)) / 3;
            break;
        case "Central":
            result = (4 * DerivativeCentral3(y, i, h / 2) -
                DerivativeCentral3(y, i, h)) / 3;
            break;
        default:
            result = double.NaN;
            break;
    }
    return result;
}

public static double DerivativeRichardson4(Function f, double x,
    double h, string flag)
{
    double result = double.NaN;
    switch (flag)
    {
        case "Backward":
            result = (4 * DerivativeBackward4(f, x, h/2) -
                DerivativeBackward4(f, x, h)) / 3;
            break;
        case "Forward":
            result = (4 * DerivativeForward4(f, x, h/2) -
                DerivativeForward4(f, x, h)) / 3;
            break;
        case "Central":
            result = (4 * DerivativeCentral4(f, x, h/2) -
                DerivativeCentral4(f, x, h)) / 3;
            break;
        default:
            result = double.NaN;
            break;
    }
    return result;
}

public static double DerivativeRichardson4(double[] y, int i, double
    h, string flag)
{
    double result = double.NaN;
    switch (flag)
    {
        case "Backward":
            result = (4 * DerivativeBackward4(y, i, h / 2) -
                DerivativeBackward4(y, i, h)) / 3;
            break;
        case "Forward":
            result = (4 * DerivativeForward4(y, i, h / 2) -
                DerivativeForward4(y, i, h)) / 3;
            break;
        case "Central":
            result = (4 * DerivativeCentral4(y, i, h / 2) -
                DerivativeCentral4(y, i, h)) / 3;
            break;
        default:
            result = double.NaN;
            break;
    }
    return result;
}
//Testing the Richardson Extrapolation Method:
public delegate double Function(double x);
//Test function: f(x) = x^3 + e^x
static double f2(double x)
{
  return x * x * x + Math.Exp(x);
}

static void Main(string[] args)
{
  double h = 0.1;
  double x = 0.5;
  Console.WriteLine("Testing Richardson's Method\n");
  Console.WriteLine("Function used: f(x) = x^3 + e^x\n");
  Console.WriteLine("KEY:\n");
  Console.WriteLine("f[n]exact = f'exact,f''exact,... f''''exact\n");
  Console.WriteLine("central[n] = 1st derivative using central method
      , etc\n");
  Console.WriteLine("richardson[n] = 1st deriv using Richardson
      method, etc\n");

  Console.WriteLine("Results:\n");
  double f1exact = 3 * x * x + Math.Exp(x); //f'(x)
  double central1 = DerivativeCentral1(f2, x, h);
  double richardson1 = DerivativeRichardson1(f2, x, h, "Central");
  Console.WriteLine(" f1exact = {0,11:n8}, central1 = {1,11:n8},
  richardson1 = {2,11:n8}", f1exact, central1, richardson1);

  double f2exact = 6 * x + Math.Exp(x); //f''(x)
  double central2 = DerivativeCentral2(f2, x, h);
  double richardson2 = DerivativeRichardson2(f2, x, h, "Central");
  Console.WriteLine(" f2exact = {0,11:n8}, central2 = {1,11:n8},
  richardson2 = {2,11:n8}", f2exact, central2, richardson2);

  double f3exact = 6 + Math.Exp(x); //f'''(x)
  double central3 = DerivativeCentral3(f2, x, h);
  double richardson3 = DerivativeRichardson3(f2, x, h, "Central");
  Console.WriteLine(" f3exact = {0,11:n8}, central3 = {1,11:n8},
  richardson3 = {2,11:n8}", f3exact, central3, richardson3);

  double f4exact = Math.Exp(x); //f''''(x)
  double central4 = DerivativeCentral4(f2, x, h);
  double richardson4 = DerivativeRichardson4(f2, x, h, "Central");
  Console.WriteLine(" f4exact = {0,11:n8}, central4 = {1,11:n8},
  richardson4 = {2,11:n8}", f4exact, central4, richardson4);
}

OUTPUT:
Testing Richardson's Method. Function used: f(x) = x^3 + e^x
KEY: f[n]exact = f'exact, f''exact, ... f''''exact
central[n] = 1st derivative using central method, etc
richardson[n] = 1st derivative using Richardson method, etc

Results:
 f1exact =  2.39872127, central1 =  2.41147051, richardson1 =
     2.39872093
 f2exact =  4.64872127, central2 =  4.65009566, richardson2 =
     4.64872116
 f3exact =  7.64872127, central3 =  7.65284720, richardson3 =
     7.64872024
 f4exact =  1.64872127, central4 =  1.65147120, richardson4 =
     1.64872075
```

11.4 Derivatives by Polynomial Interpolation

The finite difference formulas have been derived under the assumption that the data points are equally spaced and adjacent to each other. When this is not possible and the data points are unevenly spaced, then polynomial interpolation provides an alternate way for calculating numerical derivatives. The basic idea behind using this method is to approximate the derivative of $f(x)$ by calculating the derivative of the interpolating polynomial. Although other polynomials such as the Laguerre, Hermite or cubic splines may also be used in conjunction with this method, the Lagrange polynomials were chosen to demonstrate this algorithm here because they are stable, continuous, well known, easy to use and have an established track record for conducting reliable interpolation calculations. As before with the Taylor series expansion, the more points we use, the greater the precision of the calculated results. Unfortunately, it also follows that the more points we use, the longer and messier the equations become. In order to reach some kind of a compromise and generate results that fall within a reasonable amount of precision while maintaining the complexity of the equations at a somewhat manageable level, the examples that follow below were chosen using a three-point Lagrange polynomial. While similar derivations using higher order Lagrange polynomials are certainly possible, such industrious undertaking will be left as an exercise for my readers. In any event, the equations for the three-point Lagrange polynomial interpolation method along with its accompanying first and second derivatives are given as follows:

$$\begin{aligned} f(x) = {} & \frac{(x-x_i)(x-x_{i+1})}{(x_{i-1}-x_i)(x_{i-1}-x_{i+1})} f(x_{i-1}) + \frac{(x-x_{i-1})(x-x_{i+1})}{(x_i-x_{i-1})(x_i-x_{i+1})} f(x_i) \\ & + \frac{(x-x_{i-1})(x-x_i)}{(x_{i+1}-x_{i-1})(x_{i+1}-x_i)} f(x_{i+1}) \end{aligned}$$

The first and second derivatives can be found by directly differentiating the above equation:

$$\begin{aligned} f'(x) = {} & \frac{(2x-x_i-x_{i+1})}{(x_{i-1}-x_i)(x_{i-1}-x_{i+1})} f(x_{i-1}) + \frac{(2x-x_{i-1}-x_{i+1})}{(x_i-x_{i-1})(x_i-x_{i+1})} f(x_i) \\ & + \frac{(2x-x_{i-1}-x_i)}{(x_{i+1}-x_{i-1})(x_{i+1}-x_i)} f(x_{i+1}) \\ f''(x) = {} & \frac{2}{(x_{i-1}-x_i)(x_{i-1}-x_{i+1})} f(x_{i-1}) + \frac{2}{(x_i-x_{i-1})(x_i-x_{i+1})} f(x_i) \\ & + \frac{2}{(x_{i+1}-x_{i-1})(x_{i+1}-x_i)} f(x_{i+1}) \end{aligned}$$

The implementation of these equations in the calculation of numerical derivatives using the finite difference methods described earlier in this chapter for both an analytical function $f(x)$ and for an array of (x_i, y_i) of data values is illustrated below.

```
public static double DerivativeByInterpolation0(double[] x,
        double[] y, double xval)
{
  double result = double.NaN;
  int n = y.Length;
  for (int i = 1; i < n-1; i++)
  {
    if (xval > x[i-1] && xval < x[i+1])
    {
      result = y[i-1]*(xval-x[i])*(xval-x[i+1])/((x[i-1]-x[i])*
      (x[i-1]-x[i+1]))+y[i]*(xval-x[i-1])*(xval-x[i+1])/((x[i]-
       x[i-1])*(x[i]-x[i+1]))+y[i+1]*(xval-x[i-1])*(xval-x[i])/
     ((x[i+1]-x[i-1])*(x[i+1]-x[i]));
    }
  }
  return result;
}

public static double DerivativeByInterpolation1(double[] x,
        double[] y, double xval)
{
  double result = double.NaN;
  int n = y.Length;
  for (int i = 1; i < n-1; i++)
  {
    if (xval > x[i-1] && xval < x[i+1])
    {
      result = y[i-1]*(2*xval-x[i]-x[i+1])/((x[i-1]-x[i])*
      (x[i-1]-x[i+1]))+y[i]*(2*xval-x[i-1]-x[i+1])/((x[i]-x[i-1])*
      (x[i]-x[i+1]))+y[i+1]*(2*xval-x[i-1]-x[i])/((x[i+1]-x[i-1])*
      (x[i+1]-x[i]));
    }
  }
  return result;
}

public static double DerivativeByInterpolation2(double[] x,
        double[] y, double xval)
{
  double result = double.NaN;
  int n = y.Length;
  for (int i = 1; i < n-1; i++)
  {
    if (xval > x[i-1] && xval < x[i+1])
    {
      result = y[i-1]*2/((x[i-1]-x[i])*(x[i-1]-x[i+1])) +
      y[i]*2 /((x[i]-x[i-1])*(x[i]-x[i+1])) +
      y[i+1]*2 /((x[i+1]-x[i-1])*(x[i+1]-x[i]));

  }
  return result;
}
```

```
//Testing the Derivative by Interpolating Polynomial Method:

static void Main(string[] args)
{
  Console.WriteLine("\n\nTesting Derivatives by Polynomial
      Interpolation Method\n");
  double x = 3.8;
  double[] xa = new double[] {2.0,2.1,2.3,2.6,2.7,4.0};
  double[] ya = new double[]
      {3.2332,3.1634,2.7220,2.1635,-2.0829,-2.6365};
  double y0 = DerivativeByInterpolation0(xa,ya,x);
  double y1 = DerivativeByInterpolation1(xa,ya,x);
  double y2 = DerivativeByInterpolation2(xa,ya,x);
  Console.WriteLine("y={0:n6}\ny'={1:n6}\ny''={2:n6}\n",y0,y1,y2);
}

OUTPUT:

Testing Derivatives by Polynomial Interpolation Method

y = -9.157326
y' = 26.598681
y'' = 60.054505
```

12

Numerical Integration

12.1 Introduction

Numerical integration constitutes a broad family of algorithms for calculating the numerical value of a definite integral and is also a much more accurate procedure than numerical differentiation. Numerical integration aspires to solve the basic problem of computing an approximate numerical solution to a general definite integral of the form

$$\int_a^b f(x)\,dx$$

If $f(x)$ is a smooth well-behaved function, integrated over a small number of dimensions and the limits of integration are bounded, there are many excellent methods to approximate the integral with excellent precision.

Motivation to perform numerical integration originates from two sources. First, the function $f(x)$ to be integrated may be too complex or even impossible to be evaluated analytically. Second, the function $f(x)$ could be described only by an array of numbers $(x, f(x))$ so that a numerical approximation may be the only way to evaluate its integral over some given range of values.

Numerical integration methods can be divided into three broad categories: Newton-Cotes formulas, Gaussian quadrature and Monte Carlo methods. The Newton-Cotes formulas are characterized by smooth, equally spaced data points $(x, f(x))$ where $f(x)$ is either known or can be easily evaluated. The Trapezoidal and Simpson's methods, for example, are perhaps the two best known methods, belonging to this group of formulas, for numerically calculating integrals based on evaluating the integrand at equally spaced points. Alternatively, if the data points $(x, f(x))$ are not equally spaced and $f(x)$ is difficult to evaluate directly then there are other methods, better known as Gaussian quadrature, which require fewer evaluations of the integrand and are better suited for performing the integration. Monte-Carlo integration, on the other hand, is a relatively recent technique compared to all these others and is usually reserved for evaluating a numerical approximation of the most difficult integrals which usually cannot be evaluated by any other means. The rise in popularity of Monte Carlo methods is most likely due to significant improvements in computer technology over the last few years.

12.2 Newton-Cotes Formulas

The central idea behind most formulas for numerically evaluating the integral

$$I(f) = \int_a^b f(x)\,dx$$

is to replace $f(x)$ by an approximating function whose integral can then be more easily evaluated. This approximating function is often chosen to be an interpolating polynomial because polynomial integration is very easy to do.

The Newton-Cotes technique consists of a very useful and straightforward family of numerical integration formulas that use the Lagrange polynomials as their source of interpolating polynomial to approximate $f(x)$. To integrate a function $f(x)$ over some interval $[a,b]$, first divide it into n equal parts such that $x_i = x_0 + ih$ where $h = (x_n - x_0)/n = (b-a)/n$ for $i = 0,1,2,\dots,n$.

Assuming that the value of a function $f(x_i)$ is known at equally spaced points x_i, for $i = 0,1,2,\dots,n$, then the closed Newton-Cotes formula of degree n is given by

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n} w_i f(x_i)$$

where the w_i are called weights and are derived from the Lagrange basis polynomials. This means they depend only on the x_i and not on the function $f(x)$ itself. If $L(x)$ represents the interpolation polynomial in the Lagrange form for the given data points $(x_0, f(x_0)),\dots,(x_n, f(x_n))$, then we can approximate the general integral $I(f)$ given above by

$$\int_a^b f(x)\,dx \approx \int_a^b L(x)\,dx = \int_a^b \sum_{i=0}^{n} f(x_i)\,l_i(x)\,dx = \sum_{i=0}^{n} f(x_i) \underbrace{\int_a^b l_i(x)\,dx}_{w_i}.$$

The open Newton-Cotes formula of degree n is stated as

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n-1} w_i f(x_i)$$

12.2.1 Rectangle Method

The rectangle method partitions the integrating interval $[a,b]$ into n equal subintervals $[x_i, x_{i+1}]$, for $i = 0,1,\dots,n$ all with width $h = (b-a)/n$. Then the area of the rectangle over $[x_i, x_{i+1}]$ is given by $h\,f(x_i) = h\,f(a+ih)$ and the total area of the n rectangle panels is

$$\int_a^b f(x)\,dx \approx h \sum_{i=0}^{n-1} f(a+ih) + O(h^2)$$

where $O(h^2)$ is an approximate expression for the error term [22]. The rectangle method is perhaps the simplest of all the known integration methods. However, its accuracy highly depends on how large n is. As n gets larger, this approximation gets more accurate. In fact, this computation is the spirit of the definition of the Riemann integral and the limit of this approximation as $n \to \infty$ is defined and equal to $\int_a^b f(x)\,dx$ if this Riemann integral is defined. The C# code to evaluate integrals by the rectangular method is given below. The first routine assumes that the integrand $f(x)$ is known and can be expressed analytically whereas the second routine assumes that $f(x)$ is only known through a finite set of data points $(x_i, f(x_i))$ where $i = 0, 1, 2, \ldots, n$.

```
public delegate double Function(double x);
public static double Rectangular(Function f,double a,double b,int n)
{
    double sum = 0.0;
    double h = (b - a) / n ;
    for (int i = 0; i < n ; i++)
    {
        sum += h * f(a + i * h);
    }
    return sum;
}

public static double Rectangular(double[] y,double a,double b,int n)
{
    double sum = 0.0;
    double h = (b - a) / n;
    for (int i = 0; i < n; i++)
    {
        sum +=  h * y[i];
    }
    return sum;
}

static double df(double x) { return Math.Cos(x); }
static double  f(double x) { return Math.Sin(x); }

static void TestRectangular()
{
  int n = 1000;
  double result; double a = 0.0; double b = 1.0;

  result = f(b) - f(a);
  Console.WriteLine("Analytic result       ="+result.ToString());

  result = Rectangular(df, a, b, n);
  Console.WriteLine("Result using function ="+result.ToString());

  double[] y = new double[n];
  double h = (b - a) / (n - 1);
  for (int i = 0; i < n; i++)
  {
      double x = i * h;
      y[i] = df(x);
  }
  result = Rectangular(y, a, b, n);
  Console.WriteLine("Result using data array ="+result.ToString());
}
OUTPUT: Testing Rectangular method for integral(df(x)) = f(x)
where f(x) = sin(x), df(x) = cos(x) and a=0 to b=1.

Analytic result          = 0.841470984807897
Result using function    = 0.84170076353238
Result using data array = 0.841399594783247
```

12.2.2 Midpoint Method

The midpoint method tends to be more accurate than the rectangle method for a finite number n. The key idea is to make the midpoint of the subintervals $[x_i, x_{i+1}]$ intersect $f(x)$ so that we then have

$$\int_a^b f(x)\,dx \approx h\sum_{i=0}^{n-1} f(a+(i+0.5)h) + O(h^2)$$

where $O(h^2)$ is an approximate expression for the error term [22]. The C# code to evaluate integrals by the rectangular method is given below. The first routine assumes that the integrand $f(x)$ is known and can be expressed analytically whereas the second routine assumes that $f(x)$ is only known through a finite set of data points $(x_i, f(x_i))$ where $i = 0, 1, 2, \ldots, n$.

```
public delegate double Function(double x);
public static double MidpointMethod(Function f, double a, double b,
    int n)
{
    double sum = 0.0;
    double h = (b - a) / n;
    for (int i = 0; i < n; i++)
    {
        sum += h * f(a + (i + 0.5) * h);
    }
    return sum;
}

public static double MidpointMethod(double[] y, double a, double b,
    int n)
{
    double sum = 0.0;
    double h = (b - a) / (n - 1);
    for (int i = 0; i < (n - 1); i++)
    {
        sum += h * 0.5 * (y[i] + y[i + 1]);
    }
    return sum;
}
```

```
static double df(double x) { return Math.Cos(x); }
static double  f(double x) { return Math.Sin(x); }

static void TestMidpointMethod()
{
  int n = 1000;
  double result; double a = 0.0; double b = 1.0;

  result = f(b) - f(a);
  Console.WriteLine("Analytic result       ="+result.ToString());

  result = MidpointMethod(df, a, b, n);
  Console.WriteLine("Result using function ="+result.ToString());

  double[] y = new double[n];
  double h = (b - a) / (n - 1);
  for (int i = 0; i < n; i++)
  {
      double x = i * h;
      y[i] = df(x);
  }
  result = MidpointMethod(y, a, b, n);
  Console.WriteLine("Result using data array ="+result.ToString());
}
OUTPUT: Testing Midpoint method for integral(df(x)) = f(x)
where f(x) = sin(x), df(x) = cos(x) and a=0 to b=1.

Analytic result         = 0.841470984807897
Result using function   = 0.841471019869188
Result using data array = 0.841470914544858
```

12.2.3 Trapezoidal Method

The trapezoidal rule assumes that $f(x)$ is nearly linear in the interval $[a,b]$ and can therefore be approximated by a linear interpolating polynomial of the form

$$f(x) \approx f(a)\frac{(x-b)}{(a-b)} + f(b)\frac{(x-a)}{(b-a)}$$

so that

$$\int_a^b f(x)\,dx \approx \int_a^b [f(a)\frac{(x-b)}{(a-b)} + f(b)\frac{(x-a)}{(b-a)}]\,dx \approx (b-a)[\frac{f(a)+f(b)}{2}]$$

To improve on this approximation for when $f(x)$ is not a linear function on $[a,b]$, break the interval $[a,b]$ into n smaller subintervals such that $x_i = x_0 + ih$ where $h = (x_n - x_0)/n = (b-a)/n$ for $i = 0,1,2,\ldots,n$. Then add up the areas of all these smaller strips to obtain an approximate value of the integral as shown below.

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\sum_{i=0}^{n}[f(x_i)+f(x_{i+1})] = \frac{h}{2}\sum_{i=0}^{n}[f(x_0+ih)+f(x_0+(i+1)h)] + O(h^3 f^{(2)}(x))$$

Here the error term, $O(h^3 f^{(2)}(x))$, signifies that the true answer differs from the estimate by an amount that is the product of some numerical coefficient times h^3 times the value of the function's second derivative somewhere in the interval of integration [22].

The C# code to evaluate integrals by the trapezoidal rule is given below. The first routine assumes that the integrand $f(x)$ is known and can be expressed analytically whereas the second routine assumes that $f(x)$ is only known through a finite set of data points $(x_i, f(x_i))$ where $i = 0, 1, 2, \ldots, n$.

```
public delegate double Function(double x);
public static double Trapezoidal(Function f,double a,double b,int n)
{
    double sum = 0.0;
    double h = (b - a) / n;
    for (int i = 0; i < n; i++)
    {
        sum += 0.5 * h * (f(a + i * h) + f(a + (i + 1) * h));
    }
    return sum;
}

public static double Trapezoidal(double[] y,double a,double b,int n)
{
    double sum = 0.0;
    double h = (b - a) / (n-1);
    for (int i = 0; i < (n-1); i++)
    {
        sum += 0.5 * h * (y[i] + y[i + 1]);
    }
    return sum;
}

static double df(double x) { return Math.Cos(x); }
static double  f(double x) { return Math.Sin(x); }

static void TestTrapezoidal()
{
    int n = 1000;
    double result;
    double a = 0.0;
    double b = 1.0;

    result = f(b) - f(a);
    Console.WriteLine("Analytic result      ="+result.ToString());

    result = Trapezoidal(df, a, b, n);
    Console.WriteLine("Result using function="+result.ToString());

    double[] y = new double[n];
    double h = (b-a) / (n-1);
    for (int i = 0; i < n; i++)
    {
        double x = i * h;
        y[i] = df(x);
    }
    result = Trapezoidal(y, a, b, n);
    Console.WriteLine("Result using data array ="+result.ToString());
}

OUTPUT: Testing Trapezoidal method for integral(df(x)) = f(x)
where f(x) = sin(x), df(x) = cos(x) and a=0 to b=1.

Analytic result          = 0.841470984807897
Result using function    = 0.841470914685314
Result using data array = 0.841470914544858
```

12.2.4 Simpson's Method

The trapezoidal method described in the last section relies on approximating the integrand by a straight line. A better approximation can perhaps be obtained by approximating the integrand with a nonlinear function that can be easily integrated. One class of such methods, called Simpson's rules or Simpson's methods, uses quadratic (Simpson's $1/3$ method) and cubic (Simpson's $3/8$ method) polynomials to approximate the integrand.

Simpson's 1/3 Method

In this method, a quadratic second-order polynomial is used to approximate the integrand. The coefficients of a quadratic polynomial can be determined from three points. For an integral over the domain $[a,b]$, the three points used are the two endpoints $x_1 = a, x_3 = b$ and the midpoint $x_2 = (a+b)/2$. The polynomial can be written in the form $P(x) = \alpha + \beta(x-x_1) + \gamma(x-x_1)(x-x_2)$ where α, β and γ are unknown constants evaluated from the condition that the polynomial passes through the points $P(x_1) = f(x_1)$, $P(x_2) = f(x_2)$ and $P(x_3) = f(x_3)$. These three conditions yield

$$\alpha = f(x_1), \quad \beta = [f(x_2) - f(x_1)]/(x_2 - x_1) \quad \text{and} \quad \gamma = \frac{f(x_3) - 2f(x_2) + f(x_1)}{2h^2}$$

Therefore the integral $I = \int_a^b f(x)\,dx$ can be integrated over the interval $[a,b]$ giving

$$I = \int_a^b f(x)\,dx \approx \int_{x_1}^{x_3} P(x)\,dx \approx \frac{h}{3}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

In the composite Simpson's 1/3 method, the interval $[a,b]$ is divided into n subintervals of equal width h where $h = (b-a)/n$. Since three points are needed for defining a quadratic polynomial, the Simpson's $1/3$ method is applied to two adjacent subintervals at a time. Consequently, the whole interval has to be divided into an even number of subintervals. Applying the last equation for I given above over two adjacent subintervals $[x_{i-1}, x_i]$ and $[x_i, x_{i+1}]$ gives

$$I_i = \int_{i-1}^{i+1} f(x)\,dx \approx \frac{h}{3}\left[f(x_{i-1}) + 4f(x_i) + f(x_{i+1})\right]$$

where $h = x_{i+1} - x_i = x_i - x_{i-1}$. By collecting similar terms, the right side of the last equation can be simplified to give the general equation for the composite Simpson's $1/3$ method for equally spaced subintervals

$$I = \int_a^b f(x)\,dx \approx \sum I_i \approx \frac{h}{3}\left[f(a) + 4\sum_{i=2,4,6,\dots}^{n} f(x_i) + 2\sum_{j=3,5,7,\dots}^{n-1} f(x_j) + f(b)\right]$$

where $h = (b-a)/n$ and the approximate expression for the error term is $O(h^5 f^4(x))$.

Simpson's 3/8 Method

The derivation for Simpson's $3/8$ method is quite similar to that of Simpson's $1/3$ method except that now a cubic third order polynomial is used instead to approximate the integrand. Consequently, the details of the derivation will be left as an exercise for the reader. The final result is given by the following equation, more commonly known as Simpson's $3/8$ method

$$I = \int_a^b f(x)\,dx \approx \frac{3h}{8}\left[f(a) + 3\sum_{i=2,5,8,\dots}^{n-1} [f(x_i) + f(x_{i+1})] + 2\sum_{j=4,7,10,\dots}^{n-2} f(x_j) + f(b)\right]$$

where $h = (b-a)/n$ and the approximate expression for the error term is $O(h^5 f^4(x))$. Note that the subintervals must be equally spaced and the number of subintervals within $[a,b]$ must be divisible by 3. Since Simpson's $1/3$ method is only valid for an even number of subintervals, and Simpson's $3/8$ method is only valid for a number of subintervals that is divisible by 3, a combination of both can be used for integration when there are any odd number of intervals. Such combined implementation in C# of Simpson's $1/3$ and $3/8$ methods is given below where $x_i = a + ih$ for $i = 0, 1, \dots, n$ with $h = (b-a)/n$ and $x_0 = a$ and $x_n = b$. The first routine assumes that the integrand $f(x)$ is known and can be expressed analytically whereas the second routine assumes that $f(x)$ is only known through a finite set of data points $(x_i, f(x_i))$ where $i = 0, 1, 2, \dots, n$.

```
public delegate double Function(double x);
public static double Simpson(Function f,double a,double b,int n)
{
    if (n < 3) return double.NaN; //Need at least 3 points
    double sum = 0.0;
    double h = (b - a) / n;
    if (n % 2 != 0)
    {
        for (int i = 0; i < n - 1; i += 2)
        {
            sum += h*(f(a+i*h)+4*f(a+(i+1)*h)+f(a+(i+2)*h))/3;
        }
    }
    else
    {
        sum = 3*h*(f(a)+3*f(a+h)+3*f(a+2*h)+f(a+3*h))/8;
        for (int i = 3; i < n - 1; i += 2)
        {
            sum += h*(f(a+i*h)+4*f(a+(i+1)*h)+f(a+(i+2)*h))/3;
        }
    }
    return sum;
}

public static double Simpson(double[] y,double a,double b,int n)
{
    double h = (b - a) / n;
    //Need at least 3 points
    if (n < 3 || h == 0) return double.NaN;

    double sum = 0.0;
    if (n % 2 != 0)
    {
        for (int i = 0; i < n - 1; i += 2)
        {
            sum += h * (y[i] + 4 * y[i + 1] + y[i + 2]) / 3;
        }
    }
    else
    {
        sum = 3 * h * (y[0] + 3 * y[1] + 3 * y[2] + y[3]) / 8;
        for (int i = 3; i < n - 1; i += 2)
        {
            sum += h * (y[i] + 4 * y[i + 1] + y[i + 2]) / 3;
        }
    }
    return sum;
}

static double df(double x) { return Math.Cos(x); }
static double  f(double x) { return Math.Sin(x); }

static void TestSimpson()
{
   int n = 1000;
   double result;

   result = f(1) - f(0);
   Console.WriteLine("Analytic result       = "+result.ToString());

   result = Simpson(df, 0, 1, n);
   Console.WriteLine("Result using function = "+result.ToString());

   double[] ya = new double[n];
   double h = 1.0 / (n - 1);
   for (int i = 0; i < n; i++)
   {
       double x = i * h;
       ya[i] = df(x);
   }
   result = Simpson(ya, h);
   Console.WriteLine("Result using data array = "+result.ToString());
}
```

```
OUTPUT: Testing Simpson method for integral(df(x)) = f(x)
where f(x) = sin(x), df(x) = cos(x) and a=0 to b=1.

Analytic result         = 0.841470984807897
Result using function   = 0.841470984807901
Result using data array = 0.841470984807901
```

12.3 Romberg Integration

Romberg's integration method consists of essentially applying Richardson extrapolation, described in Chapter 11, repeatedly on the trapezoidal rule. Consequently, this method evaluates the integrand only at equally spaced points. The integrand should also have continuous derivatives even though fairly good results have also been reported if only a few derivatives exist [22]. Romberg's integration method can be defined inductively as follows [69]

$$
\begin{aligned}
R(1,1) &= \frac{1}{2}(b-a)(f(a)+f(b)) \\
&\vdots \\
R(n,1) &= \frac{1}{2}\left[R(n-1,1)+h_{n-1}\sum_{k=1}^{2^{n-2}} f(a+(2k-1)h_n)\right] \\
&\vdots \\
R(n,m) &= R(n,m-1)+\frac{1}{4^{m-1}-1}(R(n,m-1)-R(n-1,m-1)) \\
R(n,m) &= \frac{1}{4^{m-1}-1}(4^{m-1}R(n,m-1)-R(n-1,m-1))
\end{aligned}
$$

where $n \geq 2$, $m \geq 2$ and $h_n = \frac{b-a}{2^{n-1}}$. Note that the first extrapolation $R(n,1)$ is equivalent to the trapezoidal rule with $2^{n-1}+1$ points. Also, the second extrapolation, $R(n,2)$, is equivalent to Simpson's rule with $2^{n-1}+1$ points and so on. By the recursion formula for $R(n,m)$ given above, we can write

$$R(2,2) = \frac{4}{3}R(2,1) - \frac{1}{3}R(1,1)$$

from which we can then express the results in an array of the following format

$$\begin{bmatrix} R(1,1) & \\ R(2,1) & R(2,2) \end{bmatrix}$$

For higher order terms, this matrix format can be generalized to look like this

$$\begin{bmatrix} R(1,1) & & & & \\ R(2,1) & R(2,2) & & & \\ R(3,1) & R(3,2) & R(3,3) & & \\ \vdots & \vdots & \vdots & \ddots & \\ R(i,1) & R(i,2) & R(i,3) & \cdots & R(i,i) \end{bmatrix}$$

from which we see that the latest and greatest approximation to the integral is always the last diagonal term $R(i,i)$ of the array. As a result, this process can be continued until the difference between two successive diagonal terms becomes smaller than some given tolerance factor. The implementation of Romberg's method in C# is given below. Note that this method assumes that the integrand $f(x)$ is known and can be expressed analytically.

```
public delegate double Function(double x);
public static double RombergIntegration(Function f, double a, double
    b, int maxIterations, double tolerance)
{
    int n = 2;
    double h = b - a;
    double sum = 0.0;
    int j = 0;
    double[,] R = new double[maxIterations, maxIterations];

    R[1,1] = h*(f(a)+f(b))/2.0;
    h = h / 2;
    R[2,1] = R[1,1]/2 + h*f(a+h);
    R[2,2] = (4*R[2,1]-R[1,1])/3;
    for (j = 3; j <= maxIterations; j++)
    {
        n = 2 * n;
        h = h / 2;
        sum = 0.0;
        for (int k = 1; k <= n; k += 2)
        {
            sum += f(a + k*h);
        }
        R[j,1] = R[j-1,1]/2 + h*sum;
        double factor = 4.0;
        for (int k = 2; k <= j; k++)
        {
            R[j,k] = (factor*R[j,k-1]-R[j-1,k-1])/(factor-1);
            factor = factor*4.0;
        }
        if (Math.Abs(R[j,j]-R[j,j-1]) < tolerance*Math.Abs(R[j,j]))
        {
            sum = R[j,j];
            return sum;
        }
    }
    sum = R[n, n];
    return sum;
}
```

```
static double df(double x) { return Math.Cos(x); }
static double  f(double x) { return Math.Sin(x); }

static void TestRomberg()
{
  double result = f(1) - f(0);
  Console.WriteLine("Analytic result = "+result.ToString());

  result = RombergIntegration(df, 0, 1, 10, 1e-10);
  Console.WriteLine("Result from Romberg's method = {0}",result);
}

OUTPUT: Testing Romberg's Method for integral(df(x)) = f(x)
where f(x) = sin(x), df(x) = cos(x) and a=0 to b=1.

Analytic result              = 0.841470984807897
Result from Romberg's method = 0.841470984807879
```

12.4 Gaussian Quadrature Methods

The numerical methods examined so far for calculating $I(f)$ were based on integrating linear and quadratic polynomials and the resulting formulas were applied on subdivisions of ever smaller subintervals. In this section we consider another numerical method, often called Gaussian quadrature, that is based on the exact integration of polynomials of increasingly higher degree where no subdivision of the integration interval is necessary. Instead, we now have the freedom to choose the points at which we can evaluate the function values and this feature can lead to greater accuracy in evaluating the integral. In addition, the integral can now also contain singularities as long as they are integrable.

Gaussian integration formulas have the same general appearance as the Newton-Cotes formula

$$I(f) = \int_a^b f(x)\,dx \approx \sum_{i=0}^{n} w_i f(x_i) + R_i(x)$$

However, the main difference between them is in the way that the weights w_i and x_i are determined. In Newton-Cotes integration, the data points must be equally spaced in the interval $[a,b]$ which means that their locations are predetermined. By contrast, in Gaussian integration the locations of the points and weights are chosen so that the above equation yields an exact integral if $f(x)$ is a polynomial of degree $2n+1$ or less. This means that we can now write

$$I_n(f) = \sum_{i=0}^{n} w_i f(x_i) + R_i(x)$$

and also require that the nodes $(x_0, x_1, \cdots, x_n)$ and weights $(w_0, w_1, \cdots, w_n)$ be chosen so that $I_n(f) = I(f)$ for all polynomials $f(x)$ of as large a degree as possible. The $R_i(x)$ is just an expression for the error term.

The sequence of formulas $I_n(f)$ is called the Gaussian numerical integration method. From its definition, $I_n(f)$ uses n nodes and is exact for all polynomials of degree $\leq 2n+1$. Given an integrand over $[a, b]$

$$I(f) = \int_a^b f(x)\,dx$$

we can introduce the linear change of variables

$$x = \frac{b+a+t(b-a)}{2} \quad \text{where} \quad -1 \leq t \leq 1$$

thus transforming the integral to

$$I(f) = \frac{b-a}{2}\int_{-1}^{1} F(t)\,dt \quad \text{where} \quad F(t) = f\left(\frac{b+a+t(b-a)}{2}\right)$$

The motivation for carrying out such a change of variables will be apparent later when we start using various different polynomials for doing integrations for which it can be shown that the nodes $(x_0, x_1, \cdots, x_n)$ are the zeros of the given polynomial of degree n on the interval $[-1, 1]$. We can therefore summarize Gaussian integration with the following general formula

$$\begin{aligned} I(f) = \int_a^b f(x)\,dx &= \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b+a+x(b-a)}{2}\right)dx \\ &\approx \frac{b-a}{2}\sum_{i=0}^{n-1} w_i f\left(\frac{b+a+x_i(b-a)}{2}\right) + R_i(x) \end{aligned}$$

12.4.1 Gauss-Legendre Integration

Before we can calculate the integral of some function $f(x)$ using the Gauss-Legendre integration method and the general formula just derived above, we must first calculate the values for the nodes $(x_0, x_1, \cdots, x_n)$ and the corresponding weights $w_i(x)$ of the Legendre polynomial of degree n on the interval $[-1, 1]$. The nodes are simply the roots of the Legendre polynomials and the weights are given by the formula [19]

$$w_i(x) = \frac{2(x - x_i^2)}{(n+1)^2[P_{n-1}(x)]^2}$$

where $P_n(x)$ indicates the Legendre polynomial of degree n.

Calculating the roots and weights of a Legendre polynomial is not a trivial matter [71, 72]. However, once that is accomplished and the values are stored in arrays, then computing the numerical values of the integral of some function $f(x)$ becomes a very straight forward process. The code below illustrates one way to calculate the roots and weights of a Legendre polynomial of order n.

```
public static void LegendreNodesWeights(int n, out double[] x, out
    double[] w)
{
    double c, d, p1, p2, p3, dp;

    x = new double[n];
    w = new double[n];

    for (int i = 0; i < (n + 1) / 2; i++)
    {
        c = Math.Cos(Math.PI * (4*i+3)/(4*n+2));
        do
        {
            p2 = 0;
            p3 = 1;
            for (int j = 0; j < n; j++)
            {
                p1 = p2;
                p2 = p3;
                p3 = ((2*j+1)*c*p2-j*p1)/(j+1);
            }
            dp = n * (c*p3-p2)/(c*c-1);
            d = c;
            c -= p3 / dp;
        }
        while (Math.Abs(c-d) > 1e-12);
        x[i] = c;
        x[n - 1 - i] = -c;
        w[i] = 2*(1-x[i]*x[i])/(n+1)/(n+1)/Legendre(x[i],n+1)/
            Legendre(x[i],n+1);
        w[n - 1 - i] = w[i];
    }
}
```

Once the values for the roots and weights of a Legendre polynomial have been calculated to the desired order *n*, then the integral of some function $f(x)$ may be computed using the Gauss-Legendre integration method as shown below.

```
public delegate double Function(double x);
public static double GaussLegendre(Function f, double a, double b,
    int n)
{
    double[] x, w;
    LegendreNodesWeights(n, out x, out w);

    double sum = 0.0;
    for (int i = 0; i < n; i++)
    {
        sum += 0.5*(b-a)*w[i]*f(0.5*(a+b)+0.5*(b-a)*x[i]);
    }
    return sum;
}

static double f2(double x) { return Math.Exp(-x*x); }
static void TestGaussLegendre()
{
    Console.WriteLine("\nResult from Gauss-Legendre method:\n");
    double result;
    for (int n = 1; n < 9; n++)
    {
        result = GaussLegendre(f2, 1, 2, n);
        Console.WriteLine(" n = {0},result = {1}",n,result);
    }
}
```

```
OUTPUT: Result from Gauss-Legendre method

 n = 1, result = 0.0592870638160487
 n = 2, result = 0.0555691307729124
 n = 3, result = 0.0430050989135099
 n = 4, result = 0.0351891808008212
 n = 5, result = 0.0297682119097792
 n = 6, result = 0.0257888176077846
 n = 7, result = 0.0227458498569105
 n = 8, result = 0.0203447098938804
```

12.4.2 Gauss-Hermite Integration

Before we can calculate the integral of some function $f(x)$ using the Gauss-Hermite integration method and the general formula derived earlier in this section, we must first calculate the values for the nodes $(x_0, x_1, \cdots, x_n)$ and the corresponding weights $w_i(x)$ of the Hermite polynomial of degree n on the interval $[-1,1]$. The nodes are simply the roots of the Hermite polynomials and the weights are given by the formula [19]

$$w_i(x) = \frac{2^{n+1} n! \sqrt{\pi}}{[H_{n+1}(x_i)]^2}$$

where $H_n(x)$ indicates the Hermite polynomial of degree n.

Calculating the roots and weights of a Hermite polynomial is not a trivial matter [71, 72]. However, once that is accomplished and the values are stored in arrays, then computing the numerical values of the integral of some function $f(x)$ becomes a very straight forward process. The code below illustrates one way to calculate the roots and weights of a Hermite polynomial of order n.

```
public static void HermiteNodesWeights(int n, out double[] x, out
    double[] w)
{
    double c = 0.0;
    double d, p1, p2, p3, dp;

    x = new double[n];
    w = new double[n];
    for (int i = 0; i < (n + 1) / 2; i++)
    {
        if (i == 0)
        {
            c = Math.Sqrt(2*n+1)-1.85575*Math.Pow(2*n+1,-((double)(1)
                /(double)(6)));
        }
        else
        {
            if (i == 1)
            {
                c = c-1.14*Math.Pow(n,0.426)/c;
            }
            else
            {
                if (i == 2)
                {
                    c = 1.86*c-0.86*x[0];
                }
                else
                {
                    if (i == 3)
                    {
                        c = 1.91*c-0.91*x[1];
                    }
                    else
                    {
                        c = 2*c-x[i-2];
                    }
                }
            }
        }
        do
        {
            p2 = 0;
            p3 = Math.Pow(Math.PI, -0.25);
            for (int j = 0; j < n; j++)
            {
                p1 = p2;
                p2 = p3;
                p3 = p2*c*Math.Sqrt((double)(2)/((double)(j+1))) -
                     p1*Math.Sqrt((double)(j)/((double)(j+1)));
            }
            dp = Math.Sqrt(2 * n) * p2;
            d = c;
            c -= p3 / dp;
        }
        while (Math.Abs(c - d) > 1e-12);
        x[i] = c;
        w[i] = Math.Pow(2,n+1)*Gamma(n + 1)*Math.Sqrt(Math.PI) /
               Hermite(x[i],n+1)/Hermite(x[i],n+1);
        x[n - 1 - i] = -x[i];
        w[n - 1 - i] = w[i];
    }
}
```

Once the values for the roots and weights of a Hermite polynomial have been calculated to the desired order n, then the integral of some desired function $f(x)$ may be computed using the Gauss-Hermite integration method as shown below.

```
public delegate double Function(double x);
public static double GaussHermite(Function f, int n)
{
    double[] x, w;
    HermiteNodesWeights(n, out x, out w);

    double sum = 0.0;
    for (int i = 0; i < n; i++)
    {
        sum += w[i] * f(x[i]);
    }
    return sum;
}

static double f4(double x) { return (1 - x * x) * Math.Exp(x); }

static void TestGaussHermite()
{
    Console.WriteLine("\nResult from Gauss-Hermit method:\n");
    double result;
    for (int n = 1; n < 9; n++)
    {
        result = GaussHermite(f4, n);
        Console.WriteLine(" n = {0},result = {1}",n,result);
    }
}
```

```
OUTPUT: Result from Gauss-Hermit method

 n = 1, result = 1.77245385090552
 n = 2, result = 1.11717042753117
 n = 3, result = 0.635553156861587
 n = 4, result = 0.573344494209991
 n = 5, result = 0.569162593585641
 n = 6, result = 0.568975365834694
 n = 7, result = 0.568969118057079
 n = 8, result = 0.568968952329356
```

12.4.3 Gauss-Leguerre Integration

Before we can calculate the integral of some function $f(x)$ using the Gauss-Leguerre integration method and the general formula derived earlier in this section, we must first calculate the values for the nodes $(x_0, x_1, \cdots, x_n)$ and the corresponding weights $w_i(x)$ of the Leguerre polynomial of degree n on the interval $[-1,1]$. The nodes are simply the roots of the Leguerre polynomials and the weights are given by the formula [19]

$$w_i(x) = \frac{x_i}{(n+1)^2[L_{n+1}(x_i)]^2}$$

where $L_n(x)$ indicates the Leguerre polynomial of degree n.

Calculating the roots and weights of a Leguerre polynomial is not a trivial matter [71, 72]. However, once that is accomplished and the values are stored in arrays, then computing the numerical values of the integral of some function $f(x)$ becomes a very straight forward process. The code below illustrates one way to calculate the roots and weights of a Leguerre polynomial of order n.

```
public static void LaguerreNodesWeights(int n, out double[] x, out
    double[] w)
{
    double c = 0.0;
    double d, p1, p2, p3, dp;

    x = new double[n];
    w = new double[n];
    for (int i = 0; i < n; i++)
    {
        if (i == 0)
        {
            c = 3 / (1 + 2.4 * n);
        }
        else
        {
            if (i == 1)
            {
                c += 15 / (1 + 2.5 * n);
            }
            else
            {
                c += (1+2.55*(i-1))/(1.9*(i-1))*(c-x[i-2]);
            }
        }
        do
        {
            p2 = 0;
            p3 = 1;
            for (int j = 0; j < n; j++)
            {
                p1 = p2;
                p2 = p3;
                p3 = ((-c+2*j+1)*p2-j*p1)/(j+1);
            }
            dp = (n * p3 - n * p2) / c;
            d = c;
            c = c - p3 / dp;
        }
        while (Math.Abs(c - d) > 1e-12);
        x[i]=c;
        w[i]=x[i]/(n+1)/(n+1)/Laguerre(x[i],n+1)/Laguerre(x[i],n+1);
    }
}
```

Once the values for the roots and weights of a Leguerre polynomial have been calculated to the desired order n, then the integral of some desired function $f(x)$ may be computed using the Gauss-Leguerre integration method as shown below.

```
public delegate double Function(double x);
public static double GaussLaguerre(Function f, int n)
{
    double[] x, w;
    LaguerreNodesWeights(n, out x, out w);

    double sum = 0.0;
    for (int i = 0; i < n; i++)
    {
        sum += w[i] * f(x[i]);
    }
    return sum;
}

static double f3(double x) { return Math.Sin(x); }


static void TestGaussLaguerre()
{
    Console.WriteLine("\nResult from Gauss-Laguerre method:\n");
    double result;
    for (int n = 1; n < 9; n++)
    {
        result = GaussLaguerre(f3, n);
        Console.WriteLine(" n = {0}, result = {1}", n, result);
    }
}

OUTPUT: Result from Gauss-Laguerre method

 n = 1, result = 0.841470984807897
 n = 2, result = 0.432459454679844
 n = 3, result = 0.496029827480564
 n = 4, result = 0.504879279460199
 n = 5, result = 0.498903320956064
 n = 6, result = 0.500049474797677
 n = 7, result = 0.500038911994668
 n = 8, result = 0.4999877537353
```

12.4.4 Gauss-Chebyshev Integration

Before we can calculate the integral of some function $f(x)$ using the Gauss-Chebyshev integration method and the general formula derived earlier in this section, we must first calculate the values for the nodes $(x_0, x_1, \cdots, x_n)$ and the corresponding weights $w_i(x)$ of the Chebyshev polynomial of degree n on the interval $[-1,1]$. The nodes are simply the roots of the Chebyshev polynomials and the weights are given by the formula [19]

$$x_i = cos\left[\frac{(2i+1)\pi}{2n}\right] \quad \text{where} \quad i = 0,1,2,\cdots,n-1$$

$$w_i(x) = \frac{\pi}{n} \quad \text{where} \quad i = 0,1,2,\cdots,n-1$$

Unlike its predecessors, calculating the roots and weights of a Chebyshev polynomial for the simplest case in the interval $[-1,1]$ is a trivial matter [71, 72] and so then the integral of some desired function $f(x)$ may be computed quite easily using the Gauss-Chebyshev integration method as shown below.

```
public delegate double Function(double x);
public static double GaussChebyshev(Function f, int n)
{
    double x;
    double w = Math.PI / n;

    double sum = 0.0;
    for (int i = 0; i < n; i++)
    {
        x = Math.Cos(Math.PI * (i + 0.5)/n);
        sum += f(x);
    }
    return sum * w;
}

static double f5(double x) { return (1 - x * x) * Math.Exp(x); }
static void TestChebyshev()
{
    Console.WriteLine("\nResult from Gauss-Chebyshev method:\n");
    double result;
    for (int n = 1; n < 9; n++)
    {
        result = GaussChebyshev(f5, n);
        Console.WriteLine(" n = {0}, result = {1}", n, result);
    }
}

OUTPUT: Result from Gauss-Chebyshev method
 n = 1, result = 3.14159265358979
 n = 2, result = 1.98013302639538
 n = 3, result = 1.77972865341803
 n = 4, result = 1.77553470182922
 n = 5, result = 1.77549984480925
 n = 6, result = 1.77549968964298
 n = 7, result = 1.775499689213
 n = 8, result = 1.77549968921218
```

12.5 Multiple Integration

Multiple integration can be thought of as simply an extension of the one-dimensional integration methods discussed so far. For example, a double integral can be evaluated by means of two successive applications of one of the techniques presented above for a one-dimensional integral. In the following example consider using Simpson's $1/3$ rule for $n = 2$. Then a double integral having the general format

$$\int_a^b \int_c^d f(x,y)\,dx\,dy$$

can be numerically evaluated by considering a two-dimensional rectangular grid with dimensions given by $a < x < b$ and $c < y < d$ divided into $M \times L$ rectangles with sides

$$h_x = \frac{b-a}{M}$$
$$h_y = \frac{d-c}{L}$$

where M and L are multiples of $n = 2$. Then the subarea

$$A_{ij} = \int_{y_{n(j-1)}}^{y_{n(j+1)}} dy \int_{x_{n(i-1)}}^{x_{n(i+1)}} f(x,y)\,dy$$

can be evaluated by integrating along x and then along y according to the formula

$$A_{ij} \approx \frac{h_x}{3}(g_{j-1} + 4g_j + g_{j+1})$$

where

$$g_j \approx \frac{h_y}{3}(f_{i-1,j} + 4f_{i,j} + f_{i+1,j})$$

Substituting the equation for g_j into the equation for A_{ij} we obtain

$$A_{ij} \approx \frac{h_x h_y}{9}[(f_{i+1,j+1} + f_{i+1,j-1} + f_{i-1,j+1} + f_{i-1,j-1}) \\ + 4(f_{i,j+1} + f_{i,j-1} + f_{i+1,j} + f_{i-1,j}) + 16f_{i,j}]$$

Finally, summing up the value of A_{ij} in all subareas gives

$$I = \sum_{i=1}^{M/n} \sum_{j=1}^{L/n} A_{ij}$$

The method presented above for the numerical evaluation of double integrals can also be analogously extended to the numerical evaluation of triple integrals. To evaluate

$$\int_a^b \int_c^d \int_e^f f(x,y,z)\,dx\,dy\,dz$$

using Simpson's $1/3$ rule for $n = 2$, the cubic region $a < x < b$ and $c < y < d$ and $e < z < f$ is divided into $M \times L \times P$ smaller cubic regions of sides

$$h_x = \frac{b-a}{M}$$
$$h_y = \frac{d-c}{L}$$
$$h_z = \frac{f-e}{P}$$

where M, L and P are multiples of $n = 2$. Now the subvolume $A_{i,j,k}$ is evaluated by integrating along x to obtain

$$g_{j,k} \approx \frac{h_x}{3}(f_{i+1,j,k} + 4f_{i,j,k} + f_{i-1,j,k})$$

then along y to obtain

$$g_k \approx \frac{h_y}{3}(g_{j+1,k} + 4g_{j,k} + g_{j-1,k})$$

and finally along z to obtain

$$A_{i,j,k} \approx \frac{h_z}{3}(g_{k+1} + 4g_k + g_{k-1})$$

Combining these three last equations together results in a huge messy final equation for $A_{i,j,k}$ given by

$$\begin{aligned} A_{ij} \approx \frac{h_x h_y h_z}{27}[&(f_{i-1,j-1,k+1} + 4f_{i-1,j,k+1} + f_{i-1,j+1,k+1}) \\ &+ (4f_{i,j-1,k+1} + 16f_{i,j,k+1} + 4f_{i,j+1,k+1}) \\ &+ (f_{i+1,j-1,k+1} + 4f_{i+1,j,k+1} + f_{i+1,j+1,k+1}) \\ &+ (4f_{i-1,j-1,k} + 16f_{i-1,j,k} + 4f_{i-1,j+1,k}) \\ &+ (16f_{i,j-1,k} + 64f_{i,j,k} + 16f_{i,j+1,k}) \\ &+ (4f_{i+1,j-1,k} + 16f_{i+1,j,k} + 4f_{i+1,j+1,k}) \\ &+ (f_{i-1,j-1,k-1} + 4f_{i-1,j,k-1} + f_{i-1,j+1,k-1}) \\ &+ (4f_{i,j-1,k-1} + 16f_{i,j,k-1} + 4f_{i,j+1,k-1}) \\ &+ (f_{i+1,j-1,k-1} + 4f_{i+1,j,k-1} + f_{i+1,j+1,k-1})] \end{aligned}$$

Finally, summing up the value of $A_{i,j,k}$ in all subareas now gives

$$I = \sum_{i=1}^{M/n} \sum_{j=1}^{L/n} \sum_{k=1}^{P/n} A_{i,j,k}$$

Perhaps the most obvious conclusion that can be drawn from these results for the numerical evaluation of double and triple integrals is that the equations become increasingly messier as the dimensionality of the integrals increase. Fortunately, there are other numerical methods, such as Monte Carlo, that can be used to numerically evaluate higher order integrals.

12.6 Monte Carlo Methods

Monte Carlo methods were first developed in the 1940s as a method for estimating integrals that could not be evaluated analytically. Although many sophisticated techniques are now included under the category of *Monte Carlo methods*, the presentation given here will be limited to just a very brief introduction of this topic in order to illustrate how Monte Carlo methods may be used as an alternative technique for obtaining numerical approximations of integrals and especially those integrals which cannot be solved analytically or by some other means [73].

12.6.1 Monte Carlo Integration

One of the many applications of Monte Carlo methods is the numerical evaluation of definite integrals. Consider the one dimensional integral

$$I = \int_a^b f(x)dx$$

By application of the mean value theorem of calculus, an estimate of the integral I can be obtained by

$$I \approx (b-a)\langle f \rangle$$

where $\langle f \rangle$ denotes the mean value of $f(x)$ over the interval $a \le x \le b$. In the limit of a large number of points N, I_N can be shown to approximate the exact value given by I. If points are selected at random over the interval $[a,b]$, then the Monte Carlo estimate of the integral can be approximated by

$$I_N \approx (b-a)\langle f \rangle \approx \frac{(b-a)}{N}\sum_{i=1}^{N} f(x_i) + O\left(\frac{1}{\sqrt{N}}\right)$$

where $\langle f \rangle$ denotes the mean value of f over the set of sampled points $\{x_i\}$. By the central limit theorem, the set of all possible sums over different $\{x_i\}$ will have a Gaussian distribution. The standard deviation σ_N of the different values of I_N is a measure of the uncertainty in the integral's value and is given by

$$\sigma_N = \sqrt{\frac{\frac{1}{N}\sum_{i=1}^{N} f(x_i)^2 - \left(\frac{1}{N}\sum_{i=1}^{N} f(x_i)\right)^2}{N-1}}$$

The probability that I is within $I_N \pm \sigma_N$ is ≈ 0.68 and the probability of being with $2\sigma_N$ is ≈ 0.95. This error decays as $1/\sqrt{N}$ independent of the dimensionality of the integral, unlike grid based methods which have a strong dimensional dependence.

The accuracy of the Monte Carlo method can be enhanced by using information about the function. For example, if $g(x) \approx f(x)$ and if we can integrate $g(x)$, then we can write

$$I = \int_a^b f(x)dx = \int_a^b \frac{f(x)}{g(x)}g(x)dx = \int_{y^{-1}(a)}^{-1(b)} \frac{f(x)}{g(x)}dy$$

where $y(x) = \int^x g(t)dt$. So instead of uniformly sampling x to integrate $f(x)$ we can uniformly sample y and integrate $f(x)/g(x)$. This technique is known as *importance* *sampling* and has the effect of placing a large number of sample points where the function is large, thus yielding a better estimate of the integral.

Consider the following very simple example, $\int_0^1 x^2 dx = 1/3 = 0.3333\ldots$ that was designed to illustrate the ideas discussed so far for using Monte Carlo methods to numerically evaluate an integral.

```
// An example of integration with direct Monte
// Carlo scheme with integrand f(x) = x*x from 0 to 1.

public static void MonteCarloIntegrationDemo()
{
  Console.WriteLine("Monte Carlo Integration Example");
  Console.WriteLine("Integrating function f(x)=x*x from 0 to 1\n");
  Random rand = new Random();
  int nPts = 1000000;
  double s0 = 0;
  double sigma = 0;
  double x = 0.0;
  double fcn;
  for (int i = 0; i < nPts; ++i)
  {
      x = rand.NextDouble();
      fcn = x * x;
      s0 += fcn;
      sigma += fcn * fcn;
  }
  s0 /= nPts;
  sigma /= nPts;
  sigma = Math.Sqrt(Math.Abs(sigma - s0 * s0) / nPts);
  Console.WriteLine("Analytical result     = 0.3333333....333");
  Console.WriteLine("Integral result = " + s0 + " +- " + sigma);
}

OUTPUT: Monte Carlo Integration Example
Integrating function f(x) = x * x from 0 to 1

Analytical result = 0.3333333....333
Integral result   = 0.332668255235276 +- 0.000297724348776569
```

In many real-world applications of the Monte Carlo method, there will be a need to generate a random sample from a distribution that is either very complex or perhaps even unknown. When simpler methods for generating a random sample do not work, there are many alternative sampling techniques of various levels of difficulty that can sometimes be helpful.

12.6.2 The Metropolis Algorithm

The Metropolis algorithm [74] essentially generates a random walk of points distributed according to a required probability distribution. From an initial *position* in phase or configuration space, a proposed *move* is generated and the move either accepted or rejected according to the Metropolis algorithm. By taking a sufficient number of trial steps all of phase space is explored and the Metropolis algorithm ensures that the points are distributed according to the required probability distribution.

The Metropolis algorithm may be derived by considering the probability density, ρ at two points in configuration space X and X'. Configuration space is specified by the integration variables and limits. For one dimensional integration, $X = \{x\}$ for $x \in [a,b]$. Averaged over many trial steps, the average number of samples at X and X' should be equal, $\rho(X) = \rho(X')$.

The probability for a trial move from X to X' is defined by $T(X \to X')$ and we require

$$T(X \to X') = T(X' \to X)$$

If the probability of accepting a move from X to X' is $P(X \to X')$ then the total probability of accepting a move from X to X' is $P(X \to X')T(X \to X')$. Therefore, at equilibrium

$$\rho(X)P(X \to X')T(X \to X') = \rho(X')P(X' \to X)T(X' \to X) \quad .$$

The Metropolis algorithm corresponds to choosing

$$P(X \to X') = \min\left\{1, \frac{\rho(X')}{\rho(X)}\right\} \quad .$$

For the Metropolis algorithm to be valid, it is essential that the random walk is ergodic, that is any point X' in configuration space may be reached from any other point X. In some applications of the Metropolis algorithm, parts of configuration space may be difficult to reach. Long simulations or a modification of the algorithm are then necessary. Many methods have been developed to cope with this slowing down, but for the applications presented here the Metropolis algorithm has been found both adequate and sufficient for the problems investigated.

In order to demonstrate the power of the Metropolis algorithm and also directly compare the results obtained with those from straight random sampling which was calculated earlier, let us again embrace our very simple example where $f(x) = x^2$ so that $\int_0^1 f(x)dx = 1/3 = 0.3333\ldots$. However, this time let us apply the concepts behind importance sampling and the metropolis algorithm. First we need to obtain some weighted distribution function $w(x)$.

$$\int_0^1 f(x)dx = \int_0^1 w(x)g(x)dx$$

We can choose a distribution function $w(x)$ such that

$$w(x) = \frac{1}{C}(\exp(x^2) - 1)$$

and obtain a normalization constant

$$C = \int_0^1 (\exp(x^2) - 1)dx = 0.46265167$$

Therefore, the corresponding function $g(x)$ is

$$g(x) = \frac{f(x)}{g(x)} = \frac{0.46265167\,x^2}{\exp(x^2) - 1}$$

Although it may look like we are trying to calculate a very simple integral by the hardest method possible, we should keep in mind that it is a Monte Carlo calculation involving sampling data instead of an analytical evaluation of the integral. The following code in C# illustrates the methods and ideas we have discussed so far.

```
public static void MetropolisAlgorithmDemo()
{
    Console.WriteLine("Metropolis Integration Algorithm Example");
    Console.WriteLine("Integrating function f(x)=x*x from 0 to 1\n");
    int nsize = 100000;
    int nskip = 15;
    int ntotal = nsize * nskip;
    int neq = 10000;
    int iaccept = 0;
    double x, w, h = 0.4, z = 0.46265167;
    Random r = new Random();

    x = r.NextDouble();
    w = weight(x);
    for (int i = 0; i < neq; ++i)
    {
        double rand = r.NextDouble();
        Metropolis(ref x, ref  h, ref r, ref iaccept, ref w);
    }

    double s0 = 0; double ds = 0; iaccept = 0;
    for (int i = 0; i < ntotal; ++i)
    {
        double rand = r.NextDouble();
        Metropolis(ref  x, ref h, ref r, ref iaccept, ref w);

        if (i % nskip == 0)
        {
            double f = g(x);
            s0 += f;
            ds += f * f;
        }
    }
    s0 /= nsize; ds /= nsize;
    ds = Math.Sqrt(Math.Abs(ds - s0 * s0) / nsize);
    s0 *= z;
    ds *= z;
    double accept = 100.0 * iaccept / ntotal;
    Console.WriteLine("Analytical result  = 0.3333333....333");
    Console.WriteLine("Integral result    = " + s0 + " +- " + ds);
    Console.WriteLine("Acceptance rate    = " + accept + "%");
}

public static void Metropolis(ref double x, ref double h, ref Random
    rand, ref int iaccept, ref double w)
{
    double xold = x;
    x = x + 2 * h * (rand.NextDouble() - 0.5);
    if ((x < 0) || (x > 1)) x = xold;
    else
    {
            double wnew = weight(x);
            if (wnew > w * rand.NextDouble())
            {
                w = wnew;
                ++iaccept;
            }
            else x = xold;
        }
}

public static double weight(double x) { return Math.Exp(x*x)-1; }
public static double g(double x) { return x*x/(Math.Exp(x*x)-1); }

OUTPUT: Metropolis Integration Algorithm Example
Integrating function f(x) = x * x from 0 to 1

Analytical result = 0.3333333....333
Integral result   = 0.333222846428068 +- 0.000151405466715815
Acceptance rate   = 49.5926666666667%
```

12.7 Convolution Integrals

Convolution is an important mathematical concept and a useful tool in the area of linear systems theory, particularly in image and signal processing applications. In image processing, for example, convolution can be used as a filter to change the characteristics of the image, sharpen the edges, blur the image or remove the high or low frequency noise. In signal processing, on the other hand, convolution is a method of describing how any linear system $h[n]$ acts on any input $x[n]$ to generate the corresponding output $y[n]$. Convolution can also be used to suppress unwanted portions of the signal or to separate the signal into different parts. More specifically, the output $y[n]$ is said to be the convolution of the input signal $x[n]$ with the characteristic system response function $h[n]$ and is written as $y[n] = h[n] * x[n]$.

In general, the continuous convolution of two functions, say $h(x)$ and $g(x)$, is defined by the convolution integral [75]

$$y(x) = h(x) * g(x) = \int_{-\infty}^{+\infty} h(\alpha)g(\alpha - \beta)d\alpha$$

It can be easily shown that $h(x) * g(x) = g(x) * h(x)$, which means that the convolution operation is commutative, meaning that we can therefore also write

$$\int_{-\infty}^{+\infty} h(\alpha)g(\alpha - \beta)d\alpha = \int_{-\infty}^{+\infty} h(\alpha - \beta)g(\alpha)d\alpha$$

Using the rectangle method for approximating integrals, the convolution integral above may be approximated by

$$y(x) = h(x) * g(x) = \int_{-\infty}^{+\infty} h(\alpha)g(\alpha-\beta)d\alpha \approx \sum_{j=-\infty}^{\infty} h[j]\, g[i-j]\Delta\alpha$$

where $\Delta\alpha$ is just a scaling factor more commonly known as the sampling interval. In practice we cannot store arrays or vectors of infinite length, but rather treat all of the entries outside of our finite length vectors as if they were 0. Generally speaking, our convolution integrals may therefore be approximated by the summation of the product of a pair of discrete (but displaced) values and then have the final result multiplied by some scaling factor.

By definition, if $x[n]$ is an N-point digital signal running from 0 to $N-1$, and $h[n]$ is a M-point signal running from 0 to $M-1$, the convolution of the two signals $y[n] = x[n] * h[n]$ is an $N+M-1$ point signal running from 0 to $N+M-2$, given by [76]

$$y[i] = \sum_{j=0}^{M-1} h[j]\, x[i-j]$$

This equation is called the convolution sum and allows each point in the output signal to be calculated independently of all other points in the output signal. $x[n]$ is called the input signal, $h[n]$ is called the response and $y[n]$ is called the output signal. Since it can be easily shown that $h[n] * x[n] = x[n] * h[n]$, there are two ways to calculate the above sum. The input side algorithm loops through each sample in the input signal whereas the output side algorithm loops through each sample in the output signal. The code below illustrates how these two methods of calculating the discrete convolution of two signals may be implemented in C#.

```
public static double[] Convolution1(double[] x, double[] h)
{
    //Uses the input side algorithm

    double[] y = new double[x.Length + h.Length - 1];

    for (int i = 0; i < y.Length; i++)
    {
      y[i] = 0.0;
    }

    for (int i = 0; i < x.Length; i++)
    {
        for (int j = 0; j < h.Length; j++)
        {
            y[i+j] = y[i+j] + x[i] * h[j];
        }
    }
    return y;
}
```

```
public static double[] Convolution2(double[] x, double[] h)
{
    //Uses the output side algorithm

    double[] y = new double[x.Length + h.Length - 1];

    for (int i = 0; i < y.Length; i++)
    {
        y[i] = 0.0;
        for (int j = 0; j < h.Length; j++)
        {
            if ((i - j) < 0) continue;
            if ((i - j) > (x.Length - 1)) continue;
            y[i] = y[i] + h[j] * x[i - j];
        }
    }
    return y;
}
```

As an example to demonstrate how to numerically calculate the convolution of two discrete signals and the equivalence of the results obtained from two methods given above, consider calculating the convolution of the following two input signals, $x[n] = 2,2,3,3,4$ and $h[n] = 1,1,2$ as shown below.

```
static void Main(string[] args)
{
   double[] x = { 2, 2, 3, 3, 4 };
   double[] h = { 1, 1, 2 };

   double[] y = new double[x.Length + h.Length - 1];

   y = Convolution1(x, h);

   for (int i = 0; i < y.Length; i++)
   {
      Console.WriteLine("y[{0}] = {1}", i, y[i]);
   }

   y = Convolution2(x, h);

   for (int i = 0; i < y.Length; i++)
   {
      Console.WriteLine("y[{0}] = {1}", i, y[i]);
   }

   Console.WriteLine("Expected 2 4 9 10 13 10 8");
   Console.ReadLine();
}

OUTPUT (Convolution1): 2 4 9 10 13 10 8
OUTPUT (Convolution2): 2 4 9 10 13 10 8
```

Note how the output of these convolution calculations match exactly as expected.

Finally, let us also look at a simple practical example of calculating the convolution of two square wave signals that can be described mathematically by the following expressions

$$g(x) = \begin{cases} 1 & 0 \leq x < 1 \\ 0 & \text{elsewhere} \end{cases} \quad \text{and} \quad \mathrm{h(x)} = \begin{cases} 1 & 1 \leq x < 2 \\ 0 & \text{elsewhere} \end{cases}$$

Their convolution can be calculated as shown below.

$$\begin{aligned} y(x) = h(x) * g(x) &= \int_{-\infty}^{0} 0\, d\tau + \int_{0}^{x} 1\, d\tau + \int_{x-1}^{1} 1\, d\tau + \int_{1}^{+\infty} 0\, d\tau \\ &= \begin{cases} x & 0 \leq x < 1 \\ 2 - x & 1 \leq x < 2 \end{cases} \end{aligned}$$

These results seem to indicate that the convolution of two square wave pulses will produce a triangular shaped pulse, the size of which depends on the scaling factor of the convolution integral described earlier. The coding of this example in C# is given below.

```
static void Main(string[] args)
{
   double[] x = {  0, 1, 1, 0, 0};
   double[] h = {  0, 0, 0, 1, 1, 0, 0 };

   double[] y = new double[x.Length + h.Length - 1];

   y = Convolution1(x, h);
   for (int i = 0; i < y.Length; i++)
   {
      Console.WriteLine("y[{0}] = {1}", i, y[i]);
   }

   y = Convolution2(x, h);
   for (int i = 0; i < y.Length; i++)
   {
      Console.WriteLine("y[{0}] = {1}", i, y[i]);
   }

   Console.WriteLine("Expected 2 4 9 10 13 10 8");
   Console.ReadLine();
}

OUTPUT (Convolution1): 0 0 0 1 2 1 0 0 0 0
OUTPUT (Convolution2): 0 0 0 1 2 1 0 0 0 0
```

As expected, the convolution of two square wave pulses does produce a triangular shaped pulse. Choosing a more suitable scaling factor, if any, is left for the reader.

13

Statistical Functions

13.1 Introduction

Statistics is a vast subfield of mathematics pertaining to the collection, analysis, interpretation and presentation of data. In addition, it provides the mathematical tools necessary for the prediction and forecasting of future outcomes of random events based on the collection, analysis and interpretation of data. As a result, statistics is used by a wide variety of disciplines that includes not only the natural sciences and engineering, but also the social sciences and the humanities.

Statistics can be further subdivided into two broad categories, applied and theoretical, that focus on different aspects of this particular discipline. In descriptive statistics, for example, methods are used to summarize or describe a set of data. Inferential statistics studies patterns in the data which may be modeled in a way that accounts for randomness and uncertainty in observations. These models are then used to draw inferences about the process or population under study. Together descriptive and inferential statistics comprise a subfield of statistics called applied statistics. Mathematical statistics, on the other hand, is concerned with studying the rigorous theoretical aspects of the subject.

Given that there are already many excellent books on the subject [77, 64] and that statistics consists of an enormous amount of material, the focus of this chapter will be limited to developing just the essential software tools in C# needed to organize and analyze data using basic descriptive statistics.

13.2 Some Useful Tools

In statistics, a population consists of the entire set of all possible entities from which statistical inferences are to be drawn. For practical reasons, however, a smaller subset of the population, based on a random sample, is usually taken to be analyzed instead. If the sample is representative of the population, inferences and conclusions made from the sample can be extended to include the population as a whole.

Before doing any statistical calculations using C#, one must first decide what kind of data structure to use for storing the values drawn from a random sample. Although arrays are usually chosen for such a task, the .NET Framework also provides the `System.Collections` namespace which contains interfaces and classes that define various collections of objects, such as lists, queues, bit arrays, hash tables and dictionaries.

When working with data sets there is always some possibility that some entries may have undesired problems that, if left unchecked, could potentially halt the execution of the program by throwing some kind of exception. For example, large data sets collected during some long laboratory experiment may have some random `NaN` entries due to malfunctioning equipment. These entries may then go undetected until data is later analyzed and scrutinized more carefully. However, by then it may be too late as the `NaN` entry may cause the program to crash by throwing an exception thus leaving the user to then try and manually locate and fix the source of the problem. As a result, once the sample data has been collected and stored in an array, it would be nice to be able to verify that the data values in the array are indeed *clean* without any unwanted entries before doing any further processing. Moreover, it would be even nicer if those unwanted entries could also be automatically *fixed* without any manual intervention by the user. The following two routines were designed to do just that. The first one checks for the existence of some unwanted entry, such as `NaN`, in the input data set and returns a boolean variable. The second routine not only searches for an unwanted entry, but if one is found, the offending entry is then promptly removed from the input data set. Note that with slight modifications in the code shown below, these routines can be adapted to search for and remove any other type of unwanted data entries.

```
//Tests for existence of unwanted NaN values in a data set
public static bool NaNTest(double[] x)
{
   for (int i = 0; i < x.Length; i++)
   {
      if (Double.IsNaN(x[i]))
      {
         return true;
      }
   }
   return false;
}

//Removes all NaN values that may be found in a data set
public static double[] NaNDelete(double[] x)
{
   int n = 0;
   for (int i = 0; i < x.Length; i++)
   {
      if (Double.IsNaN(x[i]))
      { n++; }
   }
   double[] removed = new double[x.Length - n];
   int index = 0;
   for (int i = 0; i < x.Length; i++)
   {
      if (!Double.IsNaN(x[i]))
      {
         removed[index] = x[i];
         index++;
      }
   }
   return removed;
}
```

In the course of doing statistical calculations there may be times where one may desire to run some special test on the data values while preserving the integrity of the original data set. Perhaps the most common way of doing this is to simply make an extra copy of your data file thereby creating a second file that is expendable for you to mess with. Another way for you to do this is to have your code copy the contents of your original data array into some other expendable data array which you can then use for whatever purpose you want. However, the .NET Framework provides a far more elegant way of doing all of this while at the same time using far less overhead. The clone method creates a shallow copy of the array that copies only the elements of the array, whether they are reference types or value types, but does not copy the objects that the references refer to. The references in the new array point to the same objects that the references in the original array point to. As a result, you end up only messing with pointers to the data but not the actual data itself. Unlike C or C++ where you had to keep careful track of pointers and what they were pointing to, C# does all the hard work with pointers internally and seamlessly for you. In contrast, a deep copy of an array copies the entire data elements *and* everything directly or indirectly referenced by the elements. This action creates a lot of undesired overhead which can be somewhat problematic particularly when working with large data sets. As a result, the following code snippet illustrates how to apply the clone method to sort the contents of an array. For simplicity, I'm using the built-in sorting routine found in array data structures instead of one of the other fancier ones described in an earlier chapter in this book on the topic of sorting and searching.

```
//Returns a cloned sorted copy of a data array
enum SortOrder { Ascending, Descending };
static double[] Sort(double[] x,SortOrder order)
{
  return SortNoClone((double[])x.Clone(),order);
}

static double[] Sort(double[] x)
{
  return SortNoClone((double[])x.Clone(),SortOrder.Ascending);
}

static double[] SortNoClone(double[] x,SortOrder order)
{
    Array.Sort(x);
    if (order == SortOrder.Descending)
    { Array.Reverse(x); }
    return x;
}
```

13.3 Basic Statistical Functions

13.3.1 Mean and Weighted Mean

In statistics, the central tendency of a data set refers to calculating the *middle* or *expected* value of the data set. There are many different descriptive methods that can be chosen as a measurement of the central tendency of a data set but the most common one of these is known as the arithmetic mean. The arithmetic mean is calculated by adding up all the elements of the data array and then dividing the result by the total number of elements in the array as shown in the equation and code snippet that follows.

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

```
public static double Mean(double[] data)
{
   double total = 0.0d;
   for (int i = 0; i < data.Length; i++)
   {
      total += data[i];
   }
   return (total/data.Length);
}
```

It is important to remember that the mean can also refer to the expected value of a randomly selected variable from a data set which, by the way, is also called the *population mean*. As a result, there exists some confusion with regards to using the proper and correct terminology and this has led some people to claim that the *mean* value of a data set is equivalent to calculating its *average*. This is technically incorrect if the *mean* is taken to be the *arithmetic mean* as there are different types of averages: the *mean*, *median*, and *mode* as well as different types of mean.

For example, if one wants to combine average values from samples of the same population with different sample sizes, the weighed arithmetic mean is usually used. The weighed mean is similar to an arithmetic mean, where instead of each of the data points contributing equally to the final result, some data points contribute more than others. If all the weights are equal, then the weighed mean is the same as the arithmetic mean. In general, if a set of weights $w_1, \cdots, w_N$ is associated to a set of data points given by $x_1, \cdots, x_N$, then the weighed arithmetic mean is defined by

$$\bar{x} = \frac{\sum_{i=1}^{N} w_i \cdot x_i}{\sum_{i=1}^{N} w_i}$$

The weights w_i represent the bounds of the partial sample. In other applications they represent a measure for the reliability of the influence upon the mean by the respective weighed values. Note that the arithmetic mean is the special case where all weights are equal to 1.

```
public static double WeightedMean(double[] x, double[] weights)
{
    if (x.Length != weights.Length)
    {
        throw new Exception("Data and Weight
        arrays are not the same size");
    }
    double numerator = 0.0d;
    double denominator = 0.0d;
    for (int i = 0; i < x.Length; i++)
    {
        numerator += x[i] * weights[i];
        denominator += weights[i];
    }
    return numerator / denominator;
}
```

13.3.2 Geometric and Weighted Geometric Mean

The geometric mean is just another type of mean which measures the central tendency or typical value of a set of numbers. It is similar to the arithmetic mean except that instead of adding the set of numbers and then dividing the sum by the count of numbers in the set, n, the numbers are multiplied and then the *n*th root of the resulting product is taken. Note that the input data values x_i must be both non-zero and positive.

$$\overline{x} = \left(\prod_{i=1}^{N} x_i \right)^{1/N}$$

```
public static double GeometricMean(double[] data)
{
   if (data.Length == 0)
   {
      throw new Exception("Geometric Mean requires
      at least one value in the data.");
   }
   double product = 1.0;
   for (int i = 0; i < data.Length; i++)
   {
      product *= data[i];
   }
   return Math.Pow(product,(1.0/data.Length));
}
```

If a set of weights $w_1, \cdots, w_N$ is associated to the dataset $x_1, \cdots, x_N$, then the weighed geometric mean is defined by

$$\overline{x} = \left(\prod_{i=1}^{N} x_i^{w_i} \right)^{1/\sum_{i=1}^{N} w_i} \quad = \quad \exp\left(\frac{\sum_{i=1}^{N} w_i \ln x_i}{\sum_{i=1}^{N} w_i} \right)$$

Note that for the weighed geometric mean, the geometric mean is the special case where all weights are equal to 1.

```
public static double WeighedGeometricMean(double[] x,
                                          double[] weights)
{
    if (x.Length != weights.Length)
    {
        throw new Exception("Data and Weight
        arrays are not the same size");
    }
    double product = 1.0;
    double wtotal = 0.0;
    for (int i = 0; i < x.Length; i++)
    {
        product += weights[i] * Math.Log(x[i]);
        wtotal += weights[i];
    }
    return Math.Exp(product / wtotal);
}
```

13.3.3 Harmonic and Weighted Harmonic Mean

The harmonic mean is yet another type of mean which measures the central tendency or typical value of a set of numbers. It is useful for sets of numbers which are defined in relation to some unit or in situations when the average of rates is desired.

$$\overline{x} = \frac{N}{\left(\sum_{i=1}^{N} \frac{1}{x_i}\right)}$$

```
public static double HarmonicMean(double[] x)
{
   double total = 0.0d;
   for (int i = 0; i < x.Length; i++)
   {
      total += 1.0d / x[i];
   }
   return x.Length/total;
}
```

If a set of weights $w_1, \cdots, w_n$ is associated to the dataset $x_1, \cdots, x_n$, then the weighed harmonic mean is defined by

$$\overline{x} = \frac{\sum_{i=1}^{N} w_i}{\left(\sum_{i=1}^{N} \frac{w_i}{x_i}\right)}$$

Note that for the weighed harmonic mean, the harmonic mean is the special case where all weights are equal to 1.

```
public static double WeighedHarmonicMean(double[] x,double[] weights)
{
      if (x.Length != weights.Length)
      {
         throw new Exception("Data and Weight
         arrays are not the same size");
      }
      double xtotal = 0.0d;
      double ytotal = 0.0d;
      for (int i = 0; i < x.Length; i++)
      {
         xtotal += (weights[i]/x[i]);
         ytotal += weights[i];
      }
      return ytotal/xtotal;
}
```

13.3.4 Truncated Mean

The *truncated* mean or *trimmed* mean is a statistical measure of central tendency, much like the mean and median. However, it involves the calculation of the mean *after* discarding given parts of a probability distribution or sample at the high and low end, and typically discarding an equal amount on both ends. In the example that follows, `TruncatedMean()` calculates the mean of a data set *after* the specified truncation by discarding a certain percentage of the lowest and highest values of the input data array and then computing the mean of the remaining values. For example, a mean truncated by 10% is computed by discarding the lower and higher 5% of the values and taking the mean of the remaining values. `TruncatedMean()` takes both the input data array and the truncation parameter `trcValue`, which is a value between 0.0 and 1.0.

```
public static double TruncatedMean(double[] x, double trcValue)
{
    int remove = (int)Math.Floor((trcValue * x.Length) / 2.0);
    if (remove == 0)
    {
        return Mean(x);
    }
    if (remove == x.Length)
    {
        return Median(x);
    }
    double[] sorted = Sort(x,SortOrder.Ascending);
    double sum = 0.0d;
    int count = 0;
    for (int i = remove; i < (sorted.Length - remove); i++)
    {
        sum += sorted[i];
        count++;
    }
    return sum / count;
}
```

13.3.5 Root Mean Square

The quadratic mean, more commonly known as the root mean square (abbreviated RMS or rms) is a statistical measure of the magnitude of a varying quantity and is especially useful when variates are both positive and negative. The RMS of a collection of n values $\{x_1, x_2, \ldots, x_n\}$ is given by

$$x_{\text{rms}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2} = \sqrt{\frac{x_1^2 + x_2^2 + \cdots + x_N^2}{N}}.$$

and the C# code for calculating the RMS of a data set stored in an array is given by

```
public static double RMS(double[] x)
{
    double sum = 0.0d;
    for (int i = 0; i < x.Length; i++)
    {
        sum += (x[i] * x[i]);
    }
    return Math.Sqrt(sum / x.Length);
}
```

13.3.6 Median, Range and Mode

As for the median, it essentially measures the middle position of a frequency distribution for a data set. More formally, consider a sample of n variates $x_1, \cdots, x_N$. If we reorder them so that $Y_1 < Y_2 < \cdots < Y_N$, then Y_i is called the ith order statistic. Special cases include the minimum

$$Y_1 = \min(x_1, \cdots, x_N) \quad \text{and the maximum} \quad Y_N = \max(x_1, \cdots, x_N)$$

Important functions of order statistics also include the statistical range

$$R = Y_N - Y_1 \quad \text{and the midrange} \quad MR = \frac{1}{2}(Y_1 + Y_N)$$

The median of a set of numbers is simply the value where half the numbers are less than the median and half the numbers are more than the median. If the count of numbers is odd the mid point is used if the count of numbers is even the average of the two values around the midpoint is used. Given the notation just described, we can therefore express a general equation for calculating the median as

$$\bar{x} = \begin{cases} Y_{((N+1)/2)} & \text{if N is odd} \\ \frac{1}{2}(Y_{N/2} + Y_{1+N/2}) & \text{if N is even} \end{cases}$$

```
public static double Range(double[] x)
{
  double[] sorted = Sort(x);
  return (sorted[sorted.Length]-sorted[0]);
}

public static double MidRange(double[] x)
{
  double[] sorted = Sort(x);
  return 0.5 * (sorted[sorted.Length]+sorted[0]);
}

public static double Median(double[] x)
{
  double[] sorted = Sort(x);
  double position = 0.5*(sorted.Length-1);
  int lower = (int)System.Math.Floor(position);
  int upper = (int)System.Math.Ceiling(position);
  if (lower == upper)
  {
    return sorted[lower];
  }
  return sorted[lower] + ((position-lower) *
        (sorted[upper] - sorted[lower]));
}
```

The mode of a data sample is the element that occurs most often in the collection. Like the statistical mean and the median, the mode is just another way of obtaining important information about a random variable or a population in a single quantity. In general, the mode is different from mean and median, and may be very different for strongly skewed distributions. In addition, the mode is not necessarily unique, since the same maximum frequency may be attained at different values.

```
public static double Mode(double[] x)
{
  int[] counts = new int[x.Length];
  double[] temp = new double[x.Length];
  int index = 0;
  for (int i = 0; i < x.Length; i++)
  {
    int found = Array.IndexOf(temp,x[i],0,index);
    if (found >= 0)
    {
      // already seen
      counts[found]++;
    }
    else
    {
      // new value
      temp[index] = x[i];
      counts[index] = 1;
      index++;
    }
  }
  int maxValue = 0;
  int maxIndex = 0;
  for (int i = 0; i < index; i++)
  {
    if (counts[i] > maxValue)
    {
      maxValue = counts[i];
      maxIndex = i;
    }
  }
  return temp[maxIndex];
}
```

13.3.7 Mean Deviation

In statistics, the term *deviation* is used to indicate a measurement of the difference between the observed value and the mean. The sign of deviation, either positive or negative, indicates whether the observation is larger than or smaller than the mean. The magnitude of the deviation value is a measure of how different an observation is from the mean. One of the features of the mean is that the sum of the deviations across the entire set of all observations is always zero. The average deviation is calculated using the absolute value of the actual deviation and it is the sum of absolute values of the deviations divided by the number of observations.

$$\overline{d} = \frac{1}{N}\sum_{i=1}^{N}|x_i - \overline{x}|$$

In practice, however, the standard deviation is used more frequently and will be addressed in greater detail later on in this chapter. Nevertheless, the C# code for calculating the average deviation of a sample population whose data is stored in an array is presented below.

```
public static double MeanDeviation(double[] x)
{
    double sum = 0.0d;
    for (int i = 0; i < x.Length; i++)
    {
        sum += x[i];
    }
    double mean = sum / x.Length;
    sum = 0.0d;
    for (int i = 0; i < x.Length; i++)
    {
        sum += Math.Abs(x[i] - mean);
    }
    return (sum / x.Length);
}
```

13.3.8 Mean Deviation of the Mean

A closely related statistical function is the median absolute deviation (MAD) of the mean which calculates the median of the absolute deviations from the mean and offers a measure of the variability of a univariate sample. For a univariate data set $x_1, x_2, \cdots, x_n$, the MAD is defined as

$$\text{MAD} = \text{median}_i \left(\left| x_i - \text{median}_j(x_j) \right| \right)$$

In other words, starting with the deviations from the data's median, the *MAD* is the median of their absolute values. The median absolute deviation is a measure of statistical dispersion. It is a more robust estimator of scale than the sample variance or standard deviation. In addition, it exists for some statistical distributions which may not have a mean or variance.

```
public static double MedianDeviationOfTheMean(double[] x)
{
    double sum = 0.0d;
    for (int i = 0; i < x.Length; i++)
    {
        sum += x[i];
    }
    double mean = sum / x.Length;
    double[] means = new double[x.Length];
    for (int i = 0; i < means.Length; i++)
    {
        means[i] = Math.Abs(x[i] - mean);
    }
    return Median(means);
}
```

13.3.9 Mean Deviation of the Median

Another closely related statistical function is the median of the absolute deviations from the median. This function is similar to MAD except that it calculates the median deviation of the median instead of the median deviation of the mean. The proposed C# code for calculating the median deviation of the median is shown below.

```
public static double MedianDeviationOfTheMedian(double[] x)
{
    double median = Median(x);
    double[] medians = new double[x.Length];
    for (int i = 0; i < medians.Length; i++)
    {
        medians[i] = Math.Abs(x[i]-median);
    }
    return Median(medians);
}
```

13.3.10 Variance and Standard Deviation

The expected value of a random variable, $E(x)$, is defined to be its mean value, $\bar{x}$. Unfortunately, the expected value of a random variable does not provide any indication of how all the other data values of the sample are dispersed. As a result, the concepts of both variance and standard deviation have been introduced in order to provide a measurement of the dispersion for a set of values taken by a random variable. The variance of a sample is calculated by averaging the squared distance of its possible values from its mean value. Whereas the mean is a way to describe the location of a distribution, the variance is a way to capture its scale or degree of being spread out. The standard deviation is defined to be the positive square root of the variance and so it is not uncommon to find the variance denoted by σ^2 while the standard deviation is denoted by σ. If x is a random variable having an expected value $E(x) = \overline{x}$, the variance $Var(x) = \sigma^2$ of x is defined by

$$Var(x) = \sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2$$

where $\bar{x}$ is the population mean.

As an alternative way to measure dispersion about a mean value, the standard deviation is perhaps the most commonly used method for measuring statistical dispersion. A low standard deviation indicates that the data is clustered around the mean, whereas a high standard deviation indicates that the data is widely spread with significantly higher/lower figures than the mean. The standard deviation of a discrete random variable is the root-mean-square (RMS) deviation of its values from the mean. If the random variable X takes on N values $x_1, \cdots, x_N$ which are real numbers with equal probability, then its standard deviation σ is described by the following formula:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

where $\bar{x}$ is the arithmetic mean of the values $x_1, \cdots, x_N$ as described earlier in this chapter.

In the real world, though, finding the standard deviation of an entire population is unrealistic except in certain cases where every member of a population is able to be sampled. As a result, when only a sample of data from a population is available, the population standard deviation can be estimated by a slightly modified standard deviation of the sample. In most cases, however, the standard deviation is estimated by examining a random sample taken from the population. Using the definition given above for a data set and applying it to a small or moderately-sized sample results in an estimate that tends to be somewhat too low. As a result, the most commonly used formula for σ is an adjusted version, called the sample standard deviation, which is defined by

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

where $x_1, x_2, \cdots, x_N$ is the sample and $\bar{x}$ is the mean of the sample. The denominator $N-1$ is the number of degrees of freedom in the sample which, in statistics, is used to describe the number of values in the final calculation of a statistic that are free to vary. Standard deviation calculations where the term $N-1$ is used are said to produce an *unbiased* statistic, otherwise it is called a *biased* statistic. An unbiased statistic is one which, if applied in all possible circumstances, would yield an equal number of values above and below the mean statistic. An unbiased statistic is usually preferred since it tends to avoid both underestimating and overestimating results. The C# code shown below can be specified to produce both biased and unbiased calculations of variance and standard deviation.

```
enum BiasType { Biased, Unbiased };
static double Variance(double[] data, BiasType type)
{
    double sum = 0.0d;
    for (int i = 0; i < data.Length; i++)
    {
        sum += data[i];
    }
    double mean = sum / data.Length;
    sum = 0.0d;
    for (int i = 0; i < data.Length; i++)
    {
        sum += Math.Pow((data[i] - mean), 2);
    }
    double variance;
    if (type == BiasType.Biased)
    {
        variance = sum / data.Length;
    }
    else
    {
        variance = sum / (data.Length - 1);
    }
    return variance;
}

public static double StandardDeviation(double[] data)
{
    return Math.Sqrt(Variance(data, BiasType.Unbiased));
}

static double StandardDeviation(double[] data, BiasType btype)
{
    return Math.Sqrt(Variance(data,btype));
}
```

13.3.11 Moments About the Mean

In statistics, the *k*th central moment μ_k about the mean of a real valued random variable x is defined as

$$\mu_k = E(x - E(x))^k = \frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^k$$

where $E(x)$ is the expected value of x.

The first few central moments have familiar and intuitive interpretations. The first central moment is equal to zero. The second central moment about the mean is called the variance. The third central moment is called the skewness. The fourth central moment is called the kurtosis. Beyond that, it is just called the *k*th moment. The proposed C# code for calculating the central moment of a specified order *k* is shown below.

```
public static double CentralMoment(double[] data, int k)
{
   double sum = 0.0d;
   for (int i = 0; i < data.Length; i++)
   {
      sum += data[i];
   }
   double mean = sum/data.Length;
   sum = 0.0d;
   for (int i = 0; i < data.Length; i++)
   {
      sum += Math.Pow((data[i]-mean),k);
   }
   return sum/data.Length;;
}
```

13.3.12 Skewness

Skewness is a measure of the degree of asymmetry of a distribution. It is also associated with the normalized form of the third central moment μ_3 of a distribution. For a sample of N values the sample skewness is given by the formula

$$g_1 = \frac{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^3}{\left(\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2\right)^{3/2}}$$

If the left tail (tail at small end of the distribution) is longer than the right tail (tail at the large end of the distribution), the function is said to have negative skewness. If the reverse is true, it has positive skewness. If the two are equal, it has zero skewness. As with the variance and standard deviation calculations which are associated with the normalized form of the second central moment μ_2 of a distribution, the skewness calculation also depends on whether it is biased or unbiased. The C# code below calculates the skewness of the given data, according to the specified bias type.

```
static double Skewness(double[] data, BiasType type)
{
    if (data.Length < 3)
    {
        throw new Exception("Skewness requires
        at least three elements.");
    }
    // Calculate Mean
    double sum = 0.0d;
    for (int i = 0; i < data.Length; i++)
    {
        sum += data[i];
    }
    double mean = sum / data.Length;
    // Third and second moments
    double second = 0.0d;
    double third = 0.0d;
    double deviation;
    for (int i = 0; i < data.Length; i++)
    {
        deviation = data[i] - mean;
        second += (deviation * deviation);
        third += Math.Pow(deviation, 3);
    }
    // Standardized moment
    double variance;
    if (type == BiasType.Biased)
    {
        variance = second / data.Length;
    }
    else
    {
        variance = second / (data.Length - 1);
    }
    double standardizedMoment=third/Math.Pow(Math.Sqrt(variance),3);
    // Skewness
    double d = (double)data.Length;
    double skewness;
    if (type == BiasType.Unbiased)
    {
        skewness = (d/((d-1)*(d-2)))*standardizedMoment;
    }
    else
    {
        skewness = standardizedMoment / d;
    }
    return skewness;
}
```

13.3.13 Kurtosis

Kurtosis measures the degree of the peakedness of a distribution of real-valued random variables. It is also associated with the normalized form of the fourth central moment μ_4 of a distribution. For a sample of N values the sample kurtosis is given by the formula

$$g_2 = \frac{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^4}{\left(\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2\right)^2} - 3$$

Higher kurtosis means more of the variance is due to infrequent extreme deviations, as opposed to frequent modestly-sized deviations. A high kurtosis distribution has a sharper peak and longer, fatter tails, while a low kurtosis distribution has a more rounded peak and shorter thinner tails. As with the variance and standard deviation calculations which are associated with the normalized form of the second central moment μ_2 of a distribution, the kurtosis calculation also depends on whether it is biased or unbiased. The C# code below calculates the skewness of the given data, according to the specified bias type.

```
static double Kurtosis(double[] data, BiasType type)
{
  if (data.Length < 4)
  {
   throw new Exception("Kurtosis requires at least four elements");
  }
  // Mean
  double sum = 0.0d;
  for (int i = 0; i < data.Length; i++)
  {
   sum += data[i];
  }
  double mean = sum / data.Length;

  // Fourth and second moments
  double second = 0.0d;
  double fourth = 0.0d;
  double deviation;
  for (int i = 0; i < data.Length; i++)
  {
   deviation = data[i] - mean;
   second += (deviation * deviation);
   fourth += Math.Pow(deviation, 4);
  }

  // Standardized moment
  double variance;
  if (type == BiasType.Biased)
  {
   variance = second / data.Length;
  }
  else
  {
   variance = second / (data.Length - 1);
  }
  double standardizedMoment=fourth/(variance*variance);

  // Kurtosis
  double d = (double)data.Length;
  double kurtosis;
  if (type == BiasType.Unbiased)
  {
   double term1 = d*(d+1);
   double term2 = (d-1)*(d-2)*(d-3);
   double term3 = (3*(d-1)*(d-1))/((d-2)*(d-3));
   kurtosis = ((term1/term2)*standardizedMoment)-term3;
  }
  else
  {
   kurtosis = (standardizedMoment/d)-3;
  }
  return kurtosis;
}
```

13.3.14 Covariance and Correlation

Covariance provides a measure of the strength of the correlation between two or more sets of random variates. The covariance between two real-valued random variables X and Y, each with sample size N, and expected values $E(X) = \mu_X$ and $E(Y) = \mu_Y$ is defined as

$$Cov(X,Y) = \mathrm{E}((X-\mu_X)(Y-\mu_Y)) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})(y_i - \overline{y})$$

where E is the expected value operator. For uncorrelated variates, $Cov(X,Y) = 0$. However, if the variables are correlated in some way, then their covariance will be nonzero. In fact, if $cov(X,Y) > 0$, then Y tends to increase as X increases, and if $cov(X,Y) < 0$, then Y tends to decrease as X increases. Note that while statistically independent variables are always uncorrelated, the converse is not necessarily true. In the special case of $X = Y$, $Cov(X,X)$ can be easily shown to reduce to σ_X^2 and so the covariance reduces to the usual variance $\sigma_X^2 = Var(X)$. This motivates the use of the symbol $\sigma_{XY} = Cov(X,Y)$, which then provides a consistent way of denoting the variance as $\sigma_{XX} = \sigma_X^2$ where σ_X is the standard deviation. Consequently, the derived quantity

$$Cor(X,Y) = \frac{Cov(X,Y)}{\sigma_X\sigma_Y} = \frac{\sigma_{XY}}{\sigma_X\sigma_Y}$$

The C# code for calculating covariance is shown below.

```
static double Covariance(double[] data1, double[] data2,
                         BiasType type)
{
  if (data1.Length != data2.Length)
  {
   throw new Exception("The two data sets being tested
                        are of different sizes");
  }
  if (data1.Length == 0)
  {
   throw new Exception("Data in covariance method has zero length");
  }
  // Sum
  double sum1 = 0.0d;
  double sum2 = 0.0d;
  for (int i = 0; i < data1.Length; i++)
  {
    sum1 += data1[i];
    sum2 += data2[i];
  }

  // Mean
  double mean1 = sum1 / data1.Length;
  double mean2 = sum2 / data2.Length;

  // Covariance
  double total = 0.0d;
  for (int i = 0; i < data1.Length; i++)
  {
    total += ((data1[i] - mean1) * (data2[i] - mean2));
  }
  if (type == BiasType.Biased)
  {
    return total / data1.Length;
  }
  else
  {
    return total / (data1.Length - 1);
  }
}
```

The C# code for calculating correlation is shown below.

```
static double Correlation(double[] data1, double[] data2)
{
    if (data1.Length != data2.Length)
    {
        throw new Exception("The two data sets being
        tested are of different sizes");
    }
    if (data1.Length == 0)
    {
        throw new Exception("Data in correlation
        method has zero length.");
    }
    if (data1.Length == 1)
    {
        throw new Exception("Single data point
        is insufficient for correlation.");
    }
    // Sum
    double sum1 = 0.0d;
    double sum2 = 0.0d;
    for (int i = 0; i < data1.Length; i++)
    {
        sum1 += data1[i];
        sum2 += data2[i];
    }
    // Mean
    double mean1 = sum1 / data1.Length;
    double mean2 = sum2 / data2.Length;
    // Covariance
    double total = 0.0d;
    for (int i = 0; i < data1.Length; i++)
    {
     total += ((data1[i] - mean1) * (data2[i] - mean2));
    }
    double covariance = total / data1.Length;
    // Standard deviation
    sum1 = 0.0d;
    sum2 = 0.0d;
    for (int i = 0; i < data1.Length; i++)
    {
        sum1 += ((data1[i] - mean1) * (data1[i] - mean1));
        sum2 += ((data2[i] - mean2) * (data2[i] - mean2));
    }
    double stdev1 = Math.Sqrt(sum1 / data1.Length);
    double stdev2 = Math.Sqrt(sum2 / data2.Length);
    if ((stdev1 * stdev2) == 0)
    {
     throw new Exception("One of the standard deviations is zero");
    }
    return (covariance / (stdev1 * stdev2));
}
```

13.3.15 Miscellaneous Utilities

Some statistical calculations, such as those to determine percentiles or ranks, often require running some supplementary calculations on the side in support of extracting the desired information. As a result, the following self-explanatory utility routines are listed below in support of further examples to be described later.

```
//Calculates the minimum value in the given data set
public static double MinValue(double[] data)
{
   double minimum = data[0];
   double d;

   for (int i = 1; i < data.Length; i++)
   {
      d = data[i];
      if ( d < minimum )
      {
         minimum = d;
      }
   }
   return minimum;
}

//Calculates the minimum absolute value in the given data set
public static double MinAbsValue(double[] data)
{
   double minimum = Math.Abs(data[0]);
   double d;

   for (int i = 1; i < data.Length; i++)
   {
      d = Math.Abs(data[i]);
      if ( d < minimum )
      {
         minimum = d;
      }
   }
   return minimum;
}
```

```
//Calculates the index of the minimum value in a data set
public static int MinIndex(double[] data)
{
   double minimum = Double.MaxValue;
   int index = -1;
   double d;

   for (int i = 0; i < data.Length; i++)
   {
      d = data[i];
      if (d < minimum)
      {
         index = i;
         minimum = d;
      }
   }
   return index;
}

//Calculates the index of the minimum absolute value in a data set
public static int MinAbsIndex(double[] data)
{
   double minimum = Double.MaxValue;
   int index = -1;
   double d;

   for (int i = 0; i < data.Length; i++)
   {
      d = Math.Abs(data[i]);
      if ( d < minimum )
      {
         index = i;
         minimum = d;
      }
   }
   return index;
}

//Calculates the maximum value in the given data set
public static double MaxValue(double[] data)
{
   double maximum = data[0];
   double d;

   for (int i = 1; i < data.Length; i++)
   {
      d = data[i];
      if (d > maximum)
      {
         maximum = d;
      }
   }
   return maximum;
}
```

```
//Calculates the maximum absolute value in the given data set
public static double MaxAbsValue(double[] data)
{
   double maximum = Math.Abs( data[0] );
   double d;

   for (int i = 1; i < data.Length; i++)
   {
      d = Math.Abs( data[i] );
      if (d > maximum)
      {
         maximum = d;
      }
   }
   return maximum;
}

//Calculates the index of the maximum value in the given data set
public static int MaxIndex(double[] data)
{
   double maximum = Double.MinValue;
   int index = -1;
   double d;

   for (int i = 0; i < data.Length; i++)
   {
      d = data[i];
      if (d > maximum)
      {
         index = i;
         maximum = d;
      }
   }
   return index;
}

//Calculates the index of the maximum absolute value in the
//given data set
public static int MaxAbsIndex(double[] data)
{
   double maximum = Double.MinValue;
   int index = -1;
   double d;

   for (int i = 0; i < data.Length; i++)
   {
      d = Math.Abs(data[i]);
      if (d > maximum)
      {
         index = i;
         maximum = d;
      }
   }
   return index;
}
```

```
//Returns the number of elements contained in a data set
public static int Count(double[] data)
{
   return data.Length;
}

//Calculates the sum of the elements in the given data set
public static double Sum(double[] data)
{
   double sum = 0.0d;
   for (int i = 0; i < data.Length; i++)
   {
      sum += data[i];
   }
   return sum;
}
```

13.3.16 Percentiles and Rank

A percentile is the value of a variable below which a certain percent of observations fall. So the 20th percentile is the value below which 20 percent of the observations may be found. The routine that follows, `Percentile()`, calculates the value at the *n*th percentile of the elements in a data set, where $0 \leq n \leq 1$.

```
public static double Percentile(double[] data, double n)
{
  if ((n < 0) || (n > 1))
  {
   throw new Exception("Percentile must be between zero and one");
  }
  if (n == 0)
  {
   return MinValue(data);
  }
  if (n == 1.0)
  {
   return MaxValue(data);
  }
  double[] sorted = Sort(data);
  double position = n * (sorted.Length - 1);
  int lower = (int)Math.Floor(position);
  int upper = (int)Math.Ceiling(position);
  if (lower == upper)
  {
   return sorted[lower];
  }
  return sorted[lower] +
         ((position-lower)*(sorted[upper]-sorted[lower]));
}
```

The routine that follows, `PercentileRank()`, calculates the percentile in which a given value would fall, if it were in the given data set.

```
public static double PercentileRank(double[] data,double val)
{
    double[] sorted = Sort(data);
    if (val < sorted[0])
    {
        return 0.0;
    }
    if (val > sorted[sorted.Length-1])
    {
        return 1.0;
    }
    int[] bounds = Bounds(sorted,0,sorted.Length-1,val);
    int lower = bounds[0];
    int upper = bounds[1];
    if (val == sorted[lower])
    {
        return ((double)lower) / (sorted.Length-1);
    }
    else if (val == sorted[upper])
    {
        return (lower + 1.0d) / (sorted.Length-1);
    }
    //Interpolate between bound percentiles.
    return (lower+((val-sorted[lower]) /
           (sorted[upper]-sorted[lower])))/(sorted.Length-1);
}

// Assumes sorted ascending.
private static int[] Bounds(double[] data, int lower,
               int upper, double val)
{
    if (upper == (lower + 1))
    {
        int[] bounds = new int[2];
        bounds[0] = lower;
        bounds[1] = upper;
        return bounds;
    }
    int mid = (lower + upper) / 2;
    if (val <= data[mid])
    {
        return Bounds(data, lower, mid, val);
    }
    else
    {
        return Bounds(data, mid, upper, val);
    }
}
```

As a practical example to better illustrate how one might apply the statistical functions in C# presented in this chapter, I chose to analyze an array of 20 randomly chosen data points between 0 and 10. The results are presented below along with the C# source code that was used to create them.

```
static void Main(string[] args)
{
  int nPoints = 20;
  double magnitude = 10.0;
  double[] ydata = new double[nPoints];
  double[] xdata = new double[nPoints];

  for (int i = 0; i < ydata.Length; i++)
  {
    ydata[i] = magnitude * randObj.NextDouble();
    xdata[i] = ydata[i];
    Console.WriteLine("{0}", Math.Round(ydata[i],2));
  }

  double[] weights = new double[] { 1.0, 2.0, 3.0, 4.0, 5.0,
     6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0,
     16.0, 17.0, 18.0, 19.0, 20.0 };

  Console.WriteLine("\n\nStatistical Tests Summary\n");

  Console.WriteLine("Mean = {0}", Mean(ydata));
  Console.WriteLine("Weighted Mean = {0}",
                    WeightedMean(ydata, weights));
  Console.WriteLine("Geometric Mean = {0}",
                    GeometricMean(ydata));
  Console.WriteLine("Weighed Geometric Mean = {0}",
                    WeighedGeometricMean(ydata, weights));
  Console.WriteLine("Harmonic Mean = {0}", HarmonicMean(ydata));
  Console.WriteLine("Weighed Harmonic Mean = {0}",
                    WeighedHarmonicMean(ydata, weights));
  Console.WriteLine("Truncated Mean = {0}",
                    TruncatedMean(ydata, 2.0));
  Console.WriteLine("RMS = {0}", RMS(ydata));
  Console.WriteLine("Range = {0}", Range(ydata));
  Console.WriteLine("Mid Range = {0}", MidRange(ydata));
  Console.WriteLine("Median = {0}", Median(ydata));
  Console.WriteLine("Mode = {0}", Mode(ydata));
  Console.WriteLine("Mean Deviation = {0}",
                    MeanDeviation(ydata));
  Console.WriteLine("Median Deviation Of The Mean = {0}",
                    MedianDeviationOfTheMean(ydata));
  Console.WriteLine("Median Deviation Of The Median = {0}",
                    MedianDeviationOfTheMedian(ydata));
  Console.WriteLine("Variance (biased) = {0}", Variance(ydata,
                    BiasType.Biased));
  Console.WriteLine("Variance (unbiased) = {0}", Variance(ydata,
                    BiasType.Unbiased));
  Console.WriteLine("Standard Deviation = {0}",
                    StandardDeviation(ydata));
  Console.WriteLine("Central Moment = {0}",
                    CentralMoment(ydata, 2));
  Console.WriteLine("Skewness (biased) = {0}",
                    Skewness(ydata, BiasType.Biased));
  Console.WriteLine("Skewness (unbiased) = {0}",
                    Skewness(ydata, BiasType.Unbiased));
  Console.WriteLine("Kurtosis (biased) = {0}",
                     Kurtosis(ydata, BiasType.Biased));
  Console.WriteLine("Kurtosis (unbiased) = {0}",
                     Kurtosis(ydata, BiasType.Unbiased));
  Console.WriteLine("Covariance (biased) = {0}",
                     Covariance(ydata, xdata, BiasType.Biased));
  Console.WriteLine("Covariance (unbiased) = {0}",
                     Covariance(ydata, xdata, BiasType.Unbiased));
  Console.WriteLine("Correlation = {0}", Correlation(ydata, ydata));
  Console.WriteLine("MinValue = {0}", MinValue(ydata));
  Console.WriteLine("MinAbsValue = {0}", MinAbsValue(ydata));
  Console.WriteLine("MinIndex = {0}", MinIndex(ydata));
  Console.WriteLine("MinAbsIndex = {0}", MinAbsIndex(ydata));
  Console.WriteLine("MaxValue = {0}", MaxValue(ydata));
  Console.WriteLine("MaxAbsValue = {0}", MaxAbsValue(ydata));
  Console.WriteLine("MaxIndex = {0}", MaxIndex(ydata));
  Console.WriteLine("MaxAbsIndex = {0}", MaxAbsIndex(ydata));
  Console.WriteLine("Count = {0}", Count(ydata));
  Console.WriteLine("Sum = {0}", Sum(ydata));
  Console.WriteLine("Percentile = {0}", Percentile(ydata, 0.25));
  Console.WriteLine("Percentile Rank = {0}",
                     PercentileRank(ydata, 10.0));
  Console.WriteLine("\nPress ENTER key to continue...");
  Console.ReadLine();
}
```

```
OUTPUT: Statistical Tests for Chapter 13
Input Data: 20 random numbers between 0 and 10

5.64
5.49
4.82
8.1
6.01
4.84
2.15
9.99
1.13
8.63
1.79
0.1
6.65
3.68
3.62
9.7
9.68
6.27
3.43
4.04
```

```
Statistical Tests Summary

Mean = 5.28713731411245
Weighted Mean = 5.26448884501154
Geometric Mean = 3.96252438735796
Weighed Geometric Mean = 3.86297096047409
Harmonic Mean = 1.31327759771638
Weighed Harmonic Mean = 1.20013376376921
Truncated Mean = 5.16260406009974
RMS = 5.99864039745514
Range = 9.8953838739057
Mid Range = 5.04349275959818
Median = 5.16260406009974
Mode = 5.63562247698923
Mean Deviation = 2.3281402181965
Median Deviation Of The Mean = 1.76548644353891
Median Deviation Of The Median = 1.6409531895262
Variance (biased) = 8.02986563970048
Variance (unbiased) = 8.45249014705314
Standard Deviation = 2.9073166575131
Central Moment = 8.02986563970048
Skewness (biased) = 0.0780457942200665
Skewness (unbiased) = 0.0845218117489187
Kurtosis (biased) = -0.880587253499232
Kurtosis (unbiased) = -0.775667693288213
Covariance (biased) = 8.02986563970048
Covariance (unbiased) = 8.45249014705314
Correlation = 1
MinValue = 0.095800822645333
MinAbsValue = 0.095800822645333
MinIndex = 11
MinAbsIndex = 11
MaxValue = 9.99118469655103
MaxAbsValue = 9.99118469655103
MaxIndex = 7
MaxAbsIndex = 7
Count = 20
Sum = 105.742746282249
Percentile = 3.56974738001346
Percentile Rank = 1
```

14

Special Functions

14.1 Introduction

There is really nothing exceptional about *special functions* other than the fact that they originate outside the familiar collection of functions, such as trigonometric and exponential functions, which are usually found in elementary mathematics. Instead, special functions arise naturally in many areas of more advanced mathematics, the natural sciences and engineering [19]. Some special functions are defined explicitly while others originate from the solution of specific equations that were developed to explain certain physical phenomena. This chapter will therefore focus its attention on ways to implement special functions in C#.

14.2 Factorials

Factorials are not really special functions in the traditional sense but since some special functions contain factorials as part of their mathematical expression, factorials can therefore be considered important enough to be included in this chapter on special functions. In mathematics, the factorial of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than or equal to n. The factorial function is formally defined by

$$n! = \prod_{k=1}^{n} k \qquad \forall n \in \mathbb{N}$$

or recursively defined by

$$n! = \begin{cases} n \leq 1 & 1 \\ n > 1 & n(n-1)! \end{cases} \qquad \forall n \in \mathbb{N}.$$

Both of the above definitions incorporate the instance $0! = 1$.

Implementing $n!$ in C# is a bit tricky since even small integer values, such as $n > 12$, will quickly throw a numerical overflow exception. Even calculating $n!$ by recursion works well only for small values of n. As n increases in value, calculating $n!$ by recursion becomes increasingly slower.

```
//Calculates the factorial n! using recursion
public static double Factorial(double number)
{
   double Result;
   if (number < 0)
   {
      throw new Exception("Input value must be > 0");
   }
   if (number == 0.0) Result = 1; //0! = 1
   else
   //Recursively call the Factorial function
   Result = (number * Factorial(number - 1));
   return Result;
}
```

A much better approach to calculating $n!$ is to first calculate the natural logarithm of $n!$ and then take the natural exponent of the result so that we finally end up with the more efficient expression $n! = \exp(\ln n!)$. The following C# code implements this improved method for calculating $n!$. Now one is able to calculate $n!$ to as high as $170! \approx 7.25741561530799E+306$ before a numerical overflow exception is thrown. Unfortunately, because of the physical hardware limitations for handling very large numbers, computers can only provide approximate results for calculating factorials for $n > 23$. In addition, this new method for calculating $n!$ runs much faster if you just use simple table lookups for $n <= 23$ and so using the formula $n! = \exp(\ln n!)$ is recommended only for values $n > 23$. In any event, the C# code for calculating $n!$ using $n! = \exp(\ln n!)$ with the requirement of $n \leq 170$ is given below.

```
private static readonly double[] factorialLookup =
{
   1.0,
   1.0,
   2.0,
   6.0,
   24.0,
   120.0,
   720.0,
   5040.0,
   40320.0,
   362880.0,
   3628800.0,
   39916800.0,
   479001600.0,
   6227020800.0,
   87178291200.0,
   1307674368000.0,
   20922789888000.0,
   355687428096000.0,
   6402373705728000.0,
   121645100408832000.0,
   2432902008176640000.0,
   51090942171709440000.0,
   1124000727777607680000.0,
   25852016738884976640000.0
};
```

```
//Calculates n! using a lookup table up to n < 24
//For n > 24, it calculates n! = exp(Ln(n!))
public static double Factorial(int n)
{
   if (n < 0)
   {
      throw new Exception("Input value must be a > 0");
   }
   else if (n < 24)
   {
      return factorialLookup[n];
   }
   else
   {
      return Math.Floor(System.Math.Exp(FactorialLn(n))+0.5);
   }
}

public static double FactorialLn(int n)
{
   if (n < 0)
   {
      throw new Exception("Input value must be > 0");
   }
   else
   {
      return GammaLn(n+1.0);
   }
}

static void Main(string[] args)  //Testing of n!
{
   try
   {
      Console.WriteLine("Factorial(5) = {0}", Factorial(5));
      Console.WriteLine("Factorial(-5) = {0}", Factorial(-5));
      Console.ReadLine();
   }
   catch(Exception ex)
   {
      Console.WriteLine("Fatal error: " + ex.Message);
   }
}
OUTPUT:
Factorial(5) = 120
Factorial(-5) = Fatal error: Input value must be > 0
```

14.3 Combinations and Permutations

Combinatorics is the branch of mathematics studying the *enumeration*, *combination*, and *permutation* of sets of elements and the mathematical relations that characterize their properties. Mathematicians sometimes use the term combinatorics to refer to a larger subset of discrete mathematics that includes *graph theory*. In addition to applications in pure and applied mathematics, engineering and the physical sciences, combinatorial mathematics has also many useful applications in computer science, particularly in the study of data structures and network design and analysis. More recently, concepts involving combinations and permutations of sets of elements have also found many important applications in the area of software testing.

14.3.1 Combinations

A *combination* is an unordered collection of distinct elements, usually of a prescribed size and taken from a given set. The number of k combinations that can be chosen from n elements is given by the binomial coefficient, known as the *choose function*:

$$_nC_k = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

The C# code to calculate this value is given by

```
public static double Combination(int n, int k)
{
 if ((n < 0 || k < 0) || (n < k))
 throw new Exception("Input value must be > 0");

 //Cleans up round off error for small values of n and k
 return Math.Floor(0.5+(Math.Exp(FactorialLn(n)-
                       FactorialLn(k)-FactorialLn(n-k))));
}
```

While the number of k combinations that can be chosen from n elements of a set is an important piece of data to have, it would be even more useful if we could have the computer generate in lexicographical order what all these combinations might actually look like. The lexicographic or lexicographical order, also known as dictionary order, alphabetic order or lexicographic product, is a natural order structure of the Cartesian product of two ordered sets. Given two partially ordered sets A and B, the lexicographical order on the Cartesian product $A \times B$ is defined as $(a,b) = (a',b')$ if and only if $a < a'$ or $(a = a'$ and $b = b')$.

A mathematical combination lends itself nicely to implementation as a class. Since a mathematical combination represents a subset of k items selected from a set of integers from 0 through $n-1$, you need to store those values as well as an array to hold the combinations individual integer values. The letters n and k are often used in mathematical literature related to combinations and permutations and so we will also use them here. The class `combinationlex` given below is heavily commented and as such, it is quite self explanatory. A simple demonstration is also provided along with its accompanying output where the computer is asked to generate all the possible combinations of a set of 6 letters *ABCDEF* taken 3 at a time. The code can also be easily adapted for use with various other data types such as numbers, binary digits, words and so forth.

```
public class combinationlex
{
    private long n = 0;
    private long k = 0;
    private long[] combarray = null;

    //Accessors to private variables
    public long N
    {
        get { return n; }
        set { n = value; }
    }

    public long K
    {
        get { return k; }
        set { k = value; }
    }

    public long[] combArray
    {
        get { return combarray; }
        set { combarray = value; }
    }

    //Constructor
    public combinationlex(long n, long k)
    {
        // Both n and k must be positive
        if (n < 0 || k < 0)
            throw new Exception("Negative argument in constructor");
        // Assign n, k and combarray their initial default values
        this.n = n;
        this.k = k;
        this.combArray = new long[k];
        for (long i = 0; i < k; ++i)
            this.combarray[i] = i;
    }

    public double Chose(int n, int k)
    {
        if ((n < 0 || k < 0) || (n < k))
            throw new Exception("Input value must be > 0");

        //Cleans up round off error for small values of n and k
        return Math.Floor(0.5 + (Math.Exp(FactorialLn(n) -
                     FactorialLn(k) - FactorialLn(n - k))));
    }
    public combinationlex next()
    {
        //Check to see if you are at the last combination element
        //and if so, return a null
        if (this.combArray[0] == this.n - this.k) return null;

        //Create a combination object to hold the result
        combinationlex result = new combinationlex(this.n, this.k);

        //Copy the combination array data to the combination
        //object that holds the result
        for (long i = 0; i < this.k; ++i)
            result.combArray[i] = this.combArray[i];

        //Find the rightmost atom to be incremented
        //Use an index to start at the last rightmost position within
        //the array and work towards the left (decrementing) until
        //you find a false result to the condition
        //result.combArray[rIndex] == this.n - this.k + rIndex
        //or hit the start of the array
        long rIndex;
        for (rIndex = this.k - 1; rIndex > 0 &&
           result.combArray[rIndex]==this.n-this.k+rIndex; --rIndex);

        //Increment atom at that position
        ++result.combArray[rIndex];

        //Then increment every atom to the right of that atom as well
        for (long j = rIndex; j < this.k - 1; ++j)
            result.combArray[j + 1] = result.combArray[j] + 1;

        //and return the result
        return result;
    }
}

static void Main(string[] args)
{
    string[] dataArray = new string[] {"A","B","C","D","E","F"};
    int n = dataArray.Length;
    int k = 3;

    Console.WriteLine("Input data array = {A,B,C,D,E,F}");
    Console.WriteLine("Generating all lexicographical combinations
        for a set of " + dataArray.Length.ToString() +
        " elements taken " + k.ToString() + " at a time.\n");

    //Create a combination object to hold the data
    combinationlex combObj = new combinationlex(n, k);
    //and a string array to hold the current combination results
    string[] currComb = new string[n];

    long combCount = (long)combObj.Chose(n, k);
    Console.WriteLine("There are " + combCount + " possible
        combinations for a set of " + dataArray.Length.ToString() +
        " elements taken " + k.ToString() + " at a time and those
        combinations are:\n");

    //Loop through all possible combinations printing out
    //all possible combination results
    int i = 0;
    while (combObj != null)
    {
        string[] result = new string[combObj.K];
        for (long j = 0; j < result.Length; ++j)
            currComb[j] = dataArray[combObj.combArray[j]];

        Console.Write("[" + i + "] ");
        for (long m = 0; m < result.Length; ++m)
        {
            Console.Write(currComb[m] + " ");
        }
        Console.WriteLine();
        combObj = combObj.next();
        ++i;
    }
}

OUTPUT: Input data array = {A,B,C,D,E,F}
Find all lexicographical combinations for a set of 6 elements taken 3
at a time. There are 20 possible combinations those combinations are:
[00] A B C
[01] A B D
[02] A B E
[03] A B F
[04] A C D
[05] A C E
[06] A C F
[07] A D E
[08] A D F
[09] A E F
[10] B C D
[11] B C E
[12] B C F
[13] B D E
[14] B D F
[15] B E F
[16] C D E
[17] C D F
[18] C E F
[19] D E F
```

14.3.2 Permutations

In mathematics, the term *permutation* is used with different but closely related meanings that largely depends on context. The common underlying concept is that permutation relates to the notion of mapping the elements of a set to other elements of the same set by exchanging or *permuting* the elements of the set. In order to avoid ambiguity here, permutation will be defined as a sequence containing each element from a finite set once, and only once. The concept of sequence is distinct from that of a set, in that the elements of a sequence appear in some well defined order. In contrast, the elements in a set have no specific order. For example, (1, 2, 3) and (3, 2, 1) denote different sequences and also denote different ways to display the elements of the same set.

The number of distinct r permutations that can be extracted from n elements is given by the following expression

$$P(n,r) =_n P_r = \frac{n!}{(n-r)!}$$

Note that for the case where $r = n$, we have $P(n,r) = n!$. The C# code to calculate this value is given below. Note that direct calculation of $n!$ can lead very quickly to a numeric overflow for the variable n. However, using the notion that $x = e^{\ln x}$, a little trick is used that instead calculates the natural logarithm of $n!$ followed by the exponentiation of the result.

```
public static double Permutation(int n, int r)
{
 if ((n < 0 || r < 0) || (n < r))
 throw new Exception("Input value must be > 0");

 //Cleans up round off error for small values of n and k
 return Math.Floor(0.5+(Math.Exp(FactorialLn(n)-FactorialLn(n-r))));
}
```

There are many ways to list all permutations of a set of some given length n. The most natural approach builds permutations of growing length successively starting with the shortest one. For example, a one element set, say $\{1\}$, has only one trivial permutation: $\{1\}$. In a two element set, say $\{1\ 2\}$, the additional element $\{2\}$ can be appended to the permutation of $\{1\}$ in either one of two ways: on the left and on the right. Thus we get two possible permutations: $\{1\ 2\}$ and $\{2\ 1\}$. Similarly, in a set of three elements we obtain: $\{3\ 1\ 2,\ 1\ 3\ 2,\ 1\ 2\ 3\}$ and $\{3\ 2\ 1,\ 2\ 3\ 1,\ 2\ 1\ 3\}$ and so on.

The method just described is simple and appeals directly to the definition of a permutation without resorting to any fancy tricks. It has a drawback, though. You must compute a whole pyramid of permutations shorter than the ones you need. However, the Johnson-Trotter algorithm [78, 79] offers a clever and efficient way to directly generate permutations of the required length without first having to compute all the shorter permutations that precede it.

The Johnson-Trotter algorithm is set up with the idea that only one set of neighbors needs to swap positions and that there need only be one swap to generate the next permutation. To accommodate this, there needs to be an extra data element added: direction of mobility. That is, the direction of the swap. This direction is either left or right, but is initialized to the left. An integer is said to be mobile if, in the direction of its mobility, the nearest integer is less than the current integer. Note that if an integer is to the far left and its mobility is to the left, it is not mobile. Similarly, if an integer is to the far right and its mobility is to the right, it is also not mobile. The following implementation of the Johnson-Trotter algorithm was translated into

C# from a Java version of the same algorithm published by Sedgewick [80] who reportedly translated it from a C version of the algorithm published by Ruskey [81]. McCaffrey [82] has also recently published a somewhat different version of this algorithm. With slight modifications, which are left as an exercise for the reader, this algorithm can be adapted for use with characters, strings, bits or bytes.

```
public static void Permutation(int N)
{
    int[] p = new int[N];       // permutation
    int[] pi = new int[N];      // inverse permutation
    int[] dir = new int[N];     // direction = +1 or -1
    for (int i = 0; i < N; i++)
    {
        dir[i] = -1;
        p[i] = i;
        pi[i] = i;
    }
    Permutation(0, p, pi, dir);
}

public static void Permutation(int n, int[] p, int[] pi, int[] dir)
{
    // base case - print out permutation
    if (n >= p.Length)
    {
        for (int i = 0; i < p.Length; i++)
            Console.Write(p[i]);
        Console.Write("\n");
        return;
    }

    // swap
    Permutation(n + 1, p, pi, dir);
    for (int i = 0; i <= n - 1; i++)
    {
        int z = p[pi[n] + dir[n]];
        p[pi[n]] = z;
        p[pi[n] + dir[n]] = n;
        pi[z] = pi[n];
        pi[n] = pi[n] + dir[n];
        Permutation(n + 1, p, pi, dir);
    }
    dir[n] = -dir[n];
}

static void Main(string[] args)
{
   Console.WriteLine("\nTesting lexicographical permutation for 3
       elements");
   Permutation(3);
   Console.WriteLine("\n\nTesting lexicographical permutation for 4
       elements");
   Permutation(4);
}
```

```
Testing lexicographical permutation for 3 elements
012
021
201
210
120
102

Testing lexicographical permutation for 4 elements
0123
0132
0312
3012
3021
0321
0231
0213
2013
2031
2301
3201
3210
2310
2130
2103
1203
1230
1320
3120
3102
1302
1032
1023
```

14.4 Gamma Function

The gamma function was first introduced by the Swiss mathematician Leonhard Euler in 1729 while attempting to generalize the factorial to non-integer values. As a result, the gamma function is now sometimes also called the Euler integral of the second kind. In a nutshell, the gamma function, $\Gamma(z)$, is merely a generalization of the factorial function $z!$ to include both complex and positive real numbers. For a complex number z with a positive real part, the gamma function is defined by

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t}\, dt$$

where t^{z-1} is interpreted as $e^{(z-1)log t}$ if z is not an integer. If the real part of the complex number z is positive so that $(\mathrm{Re}[z] > 0)$, then the integral above can be shown to converge. Using integration by parts, it can be shown that the gamma function satisfies the recurence relation

$$\Gamma(z+1) = z\Gamma(z)$$

In addition, when the argument z is an integer, the gamma function can be shown to reduce to the familiar factorial function $n!$ offset by 1:

$$\Gamma(n+1) = n!$$

There is a variety of methods for calculating the gamma function $\Gamma(z)$ numerically but perhaps the most popular one seems to be the approximation first derived by Lanczos [83] in 1964. The result, as formulated by Press et al. [22] is that for certain choices of rational γ and integer N, for certain coefficients $c_1, c_2, \cdots, c_N$ and for $z > 0$ the gamma function can be very well approximated by the following expression

$$\Gamma(z+1) = (z+\gamma+0.5)^{z+0.5} e^{-(z+\gamma+0.5)} \sqrt{2\pi} \left[c_0 + \frac{c_1}{z+1} + \frac{c_2}{z+2} + \cdots + \frac{c_N}{z+N} + \varepsilon \right]$$

The error term is parameterized by ε. For $N = 14$, and a certain set of c's and γ Press et al. [22] report an error of $|\varepsilon| < 10^{-15}$ which should be good enough for most numerical projects. To avoid obtaining overflows for even small input values, Press et al. [22] also suggest implementing the natural logarithm of the gamma function, that is $\ln[\Gamma(x)]$, instead of $\Gamma(x)$ directly. Then, as in the case of factorials, the actual gamma function can be calculated by $\Gamma(x) = \exp(\ln \Gamma(x))$.

```
public static double GammaLn(double x)
{
   if (x <= 0) throw
   new Exception("Input value must be > 0");

   double[] coef = new double[14]
   {57.1562356658629235,
   -59.5979603554754912,
    14.1360979747417471,
   -0.491913816097620199,
     0.339946499848118887E-4,
     0.465236289270485756E-4,
   -0.983744753048795646E-4,
     0.158088703224912494E-3,
   -0.210264441724104883E-3,
     0.217439618115212643E-3,
   -0.164318106536763890E-3,
     0.844182239838527433E-4,
   -0.261908384015814087E-4,
     0.368991826595316234E-5};

   double denominator = x;
   double series = 0.999999999999997092;
   double temp = x + 5.24218750000000000;
   temp = (x + 0.5) * Math.Log(temp) - temp;
   for (int j = 0; j < 14; j++)
```

```
      series += coef[j] / ++denominator;
   return (temp+Math.Log(2.5066282746310005*series/x));
}


public static double Gamma(double x)
{
   if (x <= 0) throw
   new Exception("Input value must be > 0");
   return Math.Exp(GammaLn(x));
}
```

14.5 Beta Function

The beta function, also called the Euler integral of the first kind, is a special function defined by

$$\beta(x,y) = \int_0^1 t^{x-1}(1-t)^{y-1}\,dt = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$$

for $\mathrm{Re}(x), \mathrm{Re}(y) > 0$. The C# code for the beta function is given by

```
public static double Beta(double x, double y)
{
   if ((x <= 0) || (y <= 0)) throw
   new Exception("Input values must both be > 0");

   return Math.Exp(GammaLn(x)+GammaLn(y)-GammaLn(x+y));
}
```

14.6 Error Function

The error function, also called the Gauss error function, is a special function defined as

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$$

The complementary error function, denoted by $erfc$, is defined in terms of the error function as

$$\mathrm{erfc}(x) = 1-\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}\,dt$$

Although there are several different methods for numerically calculating the error function, Press et al. [22] have described a particularly elegant and allegedly faster technique that takes advantage of an approximation of the form

$$\operatorname{erfc}(x) \approx t \exp[-x^2 + \mathscr{P}(t)] \quad \text{where} \quad t = \frac{2}{2+x}$$

and $\mathscr{P}(t)$ is a polynomial for $0 \le t < 1$ that can be found by using Chebyshev methods. The code below represents a translation into C# from Press's original C++ code and seems to work quite well as expected.

```
static double erf(double x)
{
   if (x >= 0)
      return 1.0 - ErfcCheb(x);
   else
      return ErfcCheb(-x) - 1.0;
}

static double erfc(double x)
{
   if (x >= 0)
      return ErfcCheb(x);
   else
      return 2.0 - ErfcCheb(-x);
}

private static double ErfcCheb(double x)
{
   int j;
   double tmp;
   double d = 0.0;
   double dd = 0.0;
   const int ncoef = 28;

   if (x < 0.0) throw new Exception("ErfcCheb requires nonnegative
       argument");
   double[] coef = new double[ncoef]{-1.3026537197817094,
   6.4196979235649026e-1, 1.9476473204185836e-2,
   -9.561514786808631e-3, -9.46595344482036e-4,
   3.66839497852761e-4, 4.2523324806907e-5,
   -2.0278578112534e-5, -1.624290004647e-6,
   1.303655835580e-6, 1.5626441722e-8, -8.5238095915e-8,
   6.529054439e-9, 5.059343495e-9, -9.91364156e-10,
   -2.27365122e-10, 9.6467911e-11, 2.394038e-12,
   -6.886027e-12, 8.94487e-13, 3.13092e-13,
   -1.12708e-13, 3.81e-16, 7.106e-15, -1.523e-15,
   -9.4e-17, 1.21e-16, -2.8e-17};
   double t = 2.0/(2.0+x);
   double ty = 4.0 * t - 2.0;
   for (j=ncoef-1;j>0;j--)
   {
      tmp = d;
      d = ty*d - dd + coef[j];
      dd = tmp;
   }
   return t*Math.Exp(-x*x + 0.5*(coef[0]+ty*d)-dd);
}
```

14.7 Sine and Cosine Integral Functions

The trigonometric integrals consist of a family of integrals which involve trigonometric functions that are also expressible as a family of infinite series expansions as shown below.

$$\mathrm{Si}(x) = \int_0^x \frac{\sin t}{t}\,dt = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)(2n+1)!}$$

$$\mathrm{Ci}(x) = \int_0^x \frac{1-\cos t}{t}\,dt = \gamma + \ln x + \sum_{n=1}^{\infty} \frac{(-1)^n x^{2n}}{2n(2n)!}$$

where γ is called the Euler constant and equals

$$\gamma = \lim_{n\to\infty}\left[\left(\sum_{k=1}^{n}\frac{1}{k}\right) - \ln(n)\right] = -\int_0^{\infty} e^{-x}\ln x\,dx = 0.57721566490153286060\cdots$$

The implementation of these integral functions in C# is quite straight forward as shown below.

```
public static double Si(double x)
{
   double sum = 0.0;
   double t = 0.0;
   const double epsilon = 1.0E-10;
   int n = 0;
   do
   {
      t=Math.Pow(-1,n)*Math.Pow(x,2*n+1)/(2*n+1)/Gamma(2*n+2);
      sum += t;
      n++;
   }
   while (Math.Abs(t) > epsilon);
   return sum;
}

public static double Ci(double x)
{
   double sum = 0.0;
   double t = 0.0;
   const double epsilon = 1.0E-10;
   int n = 1;
   do
   {
      t=Math.Pow(-1,n)*Math.Pow(x,2*n)/(2*n)/Gamma(2*n+1);
      sum += t;
      n++;
   }
   while (Math.Abs(t) > epsilon);
   return 0.57721566490153286060 + Math.Log(x) + sum;
}
```

14.8 Laguerre Polynomials

The Laguerre polynomials arise from solutions of the Laguerre's second-order linear differential equation

$$x\frac{\mathrm{d}^2y}{\mathrm{dx}^2}+(1-x)\frac{\mathrm{d}y}{\mathrm{dx}}+ny=0$$

The Laguerre differential equation has nonsingular solutions only if n is a non-negative integer. These polynomials, usually denoted $L_0, L_1, \cdots, L_n$, are a polynomial sequence which may be defined by the Rodrigues' formula

$$L_n(x)=\frac{e^x}{n!}\frac{\mathrm{d}^n}{\mathrm{d}x^n}\left(e^{-x}x^n\right)$$

It is possible to also define the Laguerre polynomials recursively. Starting with the first three Laguerre polynomials

$$\begin{aligned}L_0(x)&=1\\L_1(x)&=1-x\\L_2(x)&=\frac{1}{2}(x^2-4x+2)\\&\vdots\end{aligned}$$

it can be shown that the following recurrence relation holds for any n + 1

$$L_{n+1}(x)=\frac{1}{n+1}\left((2n+1-x)L_n(x)-nL_{n-1}(x)\right).$$

The following code illustrates how one might go about implementing Laguerre polynomials in C#.

```
public static double Laguerre(double x, int deg)
{
   double L0 = 1.0; double L1 = 1.0 - x;
   double L2 = (x*x - 4*x + 2.0)/2.0;
   int n = 1;
   if (deg < 0)
    throw new Exception("Bad Laguerre polynomial: deg < 0");
   if (deg == 0) return L0;
   else if (deg == 1) return L1;
   else
   {
      while (n < deg)
      {
         L2 = ((2.0*n + 1.0 - x)*L1 - n*L0)/(n+1);
         L0 = L1; L1 = L2; n++;
      }
      return L2;
   }
}
```

14.9 Hermite Polynomials

The Hermite polynomials arise from the solution of the Hermite second-order linear differential equation

$$\frac{\mathrm{d}^2y}{\mathrm{dx}^2} - 2x\frac{\mathrm{d}y}{\mathrm{dx}} + 2ny = 0$$

and are explicitly given by the general formula

$$H_n(x) = (-1)^n e^{x^2} \frac{\mathrm{d}^n}{\mathrm{d}x^n} e^{-x^2}$$

The first three Hermite polynomials are

$$\begin{aligned} H_0(x) &= 1 \\ H_1(x) &= 2x \\ H_2(x) &= 4x^2 - 2 \\ &\vdots \end{aligned}$$

In addition, the Hermite polynomials satisfy the following recursion equation

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x)$$

The following code illustrates how one might go about implementing Hermite polynomials in C#.

```
public static double Hermite(double x, int deg)
{
   double H0 = 1.0;
   double H1 = 2*x;
   double H2 = 4*x*x - 2;
   int n = 1;
   if (deg < 0)
      throw new Exception("Bad Hermite polynomial: deg < 0");
   if (deg == 0)
      return H0;
   else if (deg == 1)
      return H1;
   else
   {
      while (n < deg)
      {
         H2 = 2.0*x*H1 - 2.0*n*H0;
         H0 = H1;
         H1 = H2;
         n++;
      }
      return H2;
   }
}
```

14.10 Chebyshev Polynomials

The Chebyshev polynomials of the first kind arise from the solution of the Chebyshev second order linear differential equation

$$(1-x^2)\frac{\mathrm{d}^2y}{\mathrm{d}x^2} - x\frac{\mathrm{d}y}{\mathrm{d}x} + n^2y = 0$$

and are generated by the recurrence relation

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

where $n = 1, 2, \ldots$. The first few Chebyshev polynomials of the first kind are given by

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ &\vdots \end{aligned}$$

Alternatively, the Chebyshev polynomials of the first kind can also be expressed by the trigonometric identity

$$T_n(\cos(\theta)) = \cos(n\theta)$$

where $n = 1, 2, \ldots$. The following code illustrates how one might go about implementing Chebyshev polynomials of the first kind in C#.

```
public static double ChebyshevT(double x, int deg)
{
   double T0 = 1.0; double T1 = x;
   double T2 = 2.0*x*x - 1.0;
   int n = 1;
   if (deg < 0)
      throw new Exception("Bad Chebyshev polynomial: deg < 0");
   if (deg == 0)
      return T0;
   else if (deg == 1)
      return T1;
   else
   {
      while (n < deg)
      {
         T2 = 2.0*x*T1 - T0;
         T0 = T1; T1 = T2; n++;
      }
      return T2;
   }
}
```

The Chebyshev polynomials of the second kind arise from the solution of the Chebyshev second order linear differential equation

$$(1-x^2)\frac{\mathrm{d}^2y}{\mathrm{d}x^2} - 3x\frac{\mathrm{d}y}{\mathrm{d}x} + n(n+2)y = 0$$

and are generated by the recurrence relation

$$U_{n+1}(x) = 2xU_n(x) - U_{n-1}(x)$$

where $n = 1,2,\ldots$. The first few Chebyshev polynomials of the second kind are given by

$$\begin{aligned}
U_0(x) &= 1 \\
U_1(x) &= 2x \\
U_2(x) &= 4x^2 - 1 \\
U_3(x) &= 8x^3 - 4x \\
&\vdots
\end{aligned}$$

Alternatively, the Chebyshev polynomials of the second kind can also be expressed by the trigonometric identity

$$U_n(\cos(\theta)) = \frac{\sin((n+1)\theta)}{\sin\theta}$$

where $n = 1,2,\ldots$. The following code illustrates how one might go about implementing Chebyshev polynomials of the second kind in C#.

```
public static double ChebyshevU(double x, int deg)
{
   double U0 = 1.0;
   double U1 = 2.0*x;
   double U2 = 4*x*x - 1.0;
   int n = 1;
   if (deg < 0)
      throw new Exception("Bad Chebyshev polynomial: deg < 0");
   if (deg == 0)
      return U0;
   else if (deg == 1)
   return U1;
   else
   {
      while (n < deg)
      {
         U2 = 2.0*x*U1 - U0;
         U0 = U1;
         U1 = U2;
         n++;
      }
      return U2;
   }
}
```

14.11 Legendre Polynomials

The Legendre polynomials arise from the solution of the Legendre second order linear differential equation

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[(1-x^2)\frac{\mathrm{d}}{\mathrm{d}x}P_n(x)\right]+n(n+1)P_n(x)=0$$

and are generated by the general expression

$$P_n(x)=\frac{1}{2^n n!}\left(\frac{\mathrm{d}}{\mathrm{d}x}\right)^n(x^2-1)^n$$

Equivalently, the Legendre polynomials may also be generated by the recurrence relation

$$P_{n+1}(x)=2xP_n(x)-P_{n-1}(x)-[xP_n(x)-P_{n-1}(x)]/(n+1)$$

where $n=1,2,3,\ldots$. The first few Legendre polynomials are given by

$$\begin{aligned}
P_0(x) &= 1\\
P_1(x) &= x\\
P_2(x) &= \frac{1}{2}(3x^2-1)\\
P_3(x) &= \frac{1}{2}(5x^3-3x)\\
&\vdots
\end{aligned}$$

The following code illustrates how one might go about implementing Legendre polynomials in C#.

```
public static double Legendre(double x, int deg)
{
   double P0 = 1.0; double P1 = x;
   double P2 = (3.0*x*x - 1)/2.0; int n = 1;
   if (deg < 0)
      throw new Exception("Bad Hermite polynomial: deg < 0");
   if (deg == 0) return P0;
   else if (deg == 1) return P1;
   else
   {
      while (n < deg)
      {
         P2 = 2.0*x*P1 - P0 - (x*P1-P0)/(deg+1);
         P0 = P1; P1 = P2; n++;
      }
      return P2;
   }
}
```

14.12 Bessel Functions

Bessel functions arise from the canonical solutions $y(x)$ of Bessel's differential equation

$$x^2\frac{\mathrm{d}^2y}{\mathrm{d}x^2}+x\frac{\mathrm{d}y}{\mathrm{d}x}+(x^2-\alpha^2)y=0$$

for an arbitrary real or complex number α which is referred to as the order of the Bessel function. The most common and important special case is where α is an integer. Since this is a second-order differential equation, there must be two linearly independent solutions. Bessel functions of the first kind, denoted as $J_\alpha(x)$, are solutions of Bessel's differential equation that are finite at the origin $(x=0)$ for non-negative integer α, and diverge as x approaches zero for negative non-integer α. It is possible to define Bessel functions of the first kind by their Taylor series expansion around $x=0$:

$$J_\alpha(x)=\sum_{n=0}^{\infty}\frac{(-1)^n}{n!\Gamma(n+\alpha+1)}\left(\frac{x}{2}\right)^{2n+\alpha}$$

```
public static double BesselJ(double x, double a)
{
  double sum = 0.0;
  double t = 0.0;
  const double epsilon = 1.0E-10;
  int n = 0;
  do
  {
   t = Math.Pow(-1,n)*Math.Pow(0.5*x,2*n+a)/Gamma(n+1)/Gamma(n+a+1);
   sum += t;
   n++;
  }
  while (Math.Abs(t) > epsilon);
  return sum;
}
```

Bessel functions of the second kind, denoted by $Y_\alpha(x)$, are also solutions of the Bessel differential equation. They are singular (infinite) at the origin $(x=0)$. $Y_\alpha(x)$ is sometimes also called the Neumann function, and is occasionally denoted instead by $N_\alpha(x)$. For non-integer α, Bessel functions of the second kind are related to $J_\alpha(x)$ by

$$Y_\alpha(x)=\frac{J_\alpha(x)\cos(\alpha\pi)-J_{-\alpha}(x)}{\sin(\alpha\pi)}.$$

```
public static double BesselY(double x, double a)
{
   return (BesselJ(x,a)*Math.Cos(a*Math.PI) -
           BesselJ(x,-a))/Math.Sin(a*Math.PI);
}
```

The spherical Bessel functions, $j_n(x)$ and $y_n(x)$ arise in the context of solving the Helmholtz partial differential equation $\nabla^2 A + k^2 A = 0$ in spherical coordinates. These alternate ways of expressing Bessel functions appear often enough in scientific and engineering numerical applications that they also merit some attention here. When solving the Helmholtz equation in spherical coordinates by separation of variables, the radial equation has the form

$$x^2 \frac{\mathrm{d}^2 y}{\mathrm{d}x^2} + 2x \frac{\mathrm{d}y}{\mathrm{d}x} + [x^2 - n(n+1)]y = 0.$$

The two linearly independent solutions to this equation are called the spherical Bessel functions $j_n(x)$ and $y_n(x)$, and are related to the ordinary Bessel functions J_n and Y_n by the following expressions

$$j_n(x) = \sqrt{\frac{\pi}{2x}} J_{n+1/2}(x)$$
$$y_n(x) = \sqrt{\frac{\pi}{2x}} Y_{n+1/2}(x)$$

Accordingly, their implementations in C# are shown below.

```
public static double Besselj(double x, double a)
{
   if (x == 0.0)
      throw new Exception("Attempted division by zero.");

   return Math.Sqrt(Math.PI/2.0/x) * BesselJ(x,a+0.5);
}

public static double Bessely(double x, double a)
{
   if (x == 0.0)
      throw new Exception("Attempted division by zero.");

   return Math.Sqrt(Math.PI/2.0/x) * BesselY(x,a+0.5);
}
```

15

Curve Fitting Methods

15.1 Introduction

Proper analysis of empirical data is arguably one of the most important tasks that both scientists and engineers are routinely asked to do. While a computer usually records and analyzes the data obtained from an experiment, it is ultimately up to people to program the computer and eventually interpret the results. Typical data collected from experiments usually contain a lot of inexact and noisy values which arise from both random and systematic errors. Random errors arise from natural limitations of making physical measurements whereas systematic errors arise from blunders in the measuring process. Either way, statistical methods of various levels of difficulty and sophistication must be used to properly extract information and interpret empirical results.

The ultimate goal of doing data analysis is to represent empirical data using a model based on mathematical equations that can best describe the physical phenomena under study. Therefore, one needs a good understanding of the underlying physics, chemistry, electronics, and other properties of a problem in order to choose or develop the most appropriate model and interpretation possible. No graphing or analysis software can pick a model for the given data. Instead, such software can only help to differentiate between models. However, once a model is picked, one can then do a rough but quick assessment of its suitability by plotting the data. In order to have some confidence in selecting a model and interpreting the results, at least some agreement is needed between the empirical data and the expected values predicted by the model. For example, if the model is supposed to represent exponential growth and the data points are monotonically decreasing, then the model is obviously wrong.

With the correct model, one can determine important characteristics of the data, such as the rate of change anywhere on the curve (first derivative), the local minimum and maximum points of the function (zeros of the first derivative), and the area under the curve (integral). Therefore, one important goal of empirical data analysis is to find the best set of parameter values of either a linear or nonlinear equation that most closely matches the data. When this happens, the expression obtained is said to be the *best fit* function to those data points.

15.2 Least Squares Fit

The least squares method was first described by the famous German mathematician Carl Friedrich Gauss around 1794 and is still perhaps the most popular curve fitting procedure. The best fit in the least-squares sense is that instance of the model for which the sum of squared residuals has its least value, a residual being the difference between an observed value and the value given by the model. The sum of the squares of the offsets is used, instead of the absolute values of the actual offset, because this allows the residuals to be treated as a continuous differentiable quantity. However, because squares of the offsets are used, outlying points can have a disproportionate effect on the fit, a property which may or may not be desirable depending on the problem at hand.

A general equation for least squares fitting can be derived by assuming we have a data set consisting of n data points: (x_i, y_i) where $i = 0, 1, 2, \ldots, n-1$ and x_i is an independent variable and y_i is a dependent variable. A general model function for n data points can then be defined by [84]

$$f(x; \mathbf{a}) = f(x; a_0, a_1, a_2, \ldots, a_{n-1})$$

where $a_0, a_1, a_2, \ldots, a_{n-1}$ are variable parameters. The objective is to find those parameter values for which the model best fits the data. The form of the actual model function is determined ahead of time and is usually based on the theory associated with the experiment for which the data is being taken. The least squares method defines *best* as when the sum, S, of squared residuals

$$S = \sum_{i=0}^{n} r_i^2$$

is a minimum. A residual is defined as the difference between the values of the dependent variable y_i and the predicted values from the estimated model function $f(x_i, \mathbf{a})$,

$$r_i = y_i - f(x_i, \mathbf{a})$$

Least squares problems fall into two categories, linear and nonlinear. The linear least squares problem has a closed form solution, but the nonlinear problem does not and is usually solved by iterative refinement where at each iteration the system is approximated by a linear one, so the core calculation is similar in both cases.

The minimum of the sum of squares is found by setting the gradient of S to zero. Since the model contains n parameters there are $0, 1, 2, \ldots, n-1$ gradient equations.

$$\frac{\partial S}{\partial a_j} = 2 \sum_{i=0}^{n} r_i \frac{\partial r_i}{\partial a_j} = 0, \quad j = 0, \ldots, n-1$$

and since $r_i = y_i - f(x_i, \mathbf{a})$ the gradient equations become

$$\frac{\partial S}{\partial a_j} = -2 \sum_{i=0}^{n} \frac{\partial f(x_i, \mathbf{a})}{\partial a_j} r_i = 0, \quad j = 0, \ldots, n-1$$

15.2.1 Straight-Line Fit

The simplest least squares problem consists of finding the best fit equation to a straight line. Using the slope-intercept form of a straight line, $y = mx + b$, we can immediately identify the coefficients as $a_0 = b = y$-intercept and $a_1 = m =$ slope so that the model function becomes $f(x,\mathbf{a}) = a_0 + a_1 x$ or equivalently, $y = mx + b$. We can therefore calculate the sum of the squares of the residuals S as

$$S(m,b) = \sum_{i=0}^{n} [y_i - f(x_i;\mathbf{a})]^2 = \sum_{i=0}^{n} [y_i - b_i - m_i x_i]^2$$

The minimum of the sum of squares is found by setting the gradient of S to zero: $\frac{\partial S}{\partial b} = 0$ and $\frac{\partial S}{\partial m} = 0$ from which we obtain the following set of equations

$$\begin{array}{rcrcl} nb & + & \left(\sum_{i=0}^{n} x_i\right) m & = & \sum_{i=0}^{n} y_i \\ \left(\sum_{i=0}^{n} x_i\right) b & + & \left(\sum_{i=0}^{n} x_i^2\right) m & = & \sum_{i=0}^{n} x_i y_i \end{array}$$

which can be written in matrix form as

$$\begin{bmatrix} n & \sum_{i=0}^{n} x_i \\ \sum_{i=0}^{n} x_i & \sum_{i=0}^{n} x_i^2 \end{bmatrix} \begin{bmatrix} b \\ m \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^{n} y_i \\ \sum_{i=0}^{n} x_i y_i \end{bmatrix}$$

from which we can solve for b and m to give

$$b = \frac{\left(\sum_{i=0}^{n} x_i^2\right)\left(\sum_{i=0}^{n} y_i\right) - \left(\sum_{i=0}^{n} x_i\right)\left(\sum_{i=0}^{n} x_i y_i\right)}{n\sum_{i=0}^{n} x_i^2 - \left(\sum_{i=0}^{n} x_i\right)^2}$$

$$m = \frac{n\sum_{i=0}^{n} x_i y_i - \left(\sum_{i=0}^{n} x_i\right)\left(\sum_{i=0}^{n} y_i\right)}{n\sum_{i=0}^{n} x_i^2 - \left(\sum_{i=0}^{n} x_i\right)^2}$$

These equations can be further simplified by recognizing and substituting the expressions for the mean values of x and y as

$$\overline{x} = \frac{1}{n}\sum_{i=0}^{n} x_i \qquad \text{and} \qquad \overline{y} = \frac{1}{n}\sum_{i=0}^{n} y_i$$

from which we finally obtain

$$b = \frac{1}{n}\left(\sum_{i=0}^{n} y_i - m\sum_{i=0}^{n} x_i\right) = \overline{y} - m\overline{x} \qquad \text{and} \qquad m = \frac{\sum_{i=0}^{n} y_i(x_i - \overline{x})}{\sum_{i=0}^{n} x_i(x_i - \overline{x})}$$

Implementations in C# of these two sets of best-fit formulas are given below. The `LeastSquaresBestFitLine1` algorithm follows the more compact version, whereas the `LeastSquaresBestFitLine2` follows the longer and more traditional version of the two equations. An example illustrating how both of these algorithms may be implemented follows immediately below and includes the results from a sample set of data points.

```
public static double[] LeastSquaresBestFitLine1(double[] x,double[]y)
{
    //Calculates equation of best-fit line using shortcuts
    int n = x.Length;
    double xMean = 0.0;
    double yMean = 0.0;
    double numeratorSum = 0.0;
    double denominatorSum = 0.0;
    double bestfitYintercept = 0.0;
    double bestfitSlope = 0.0;
    double sigma = 0.0;
    double sumOfResidualsSquared = 0.0;

    //Calculates the mean values for x and y arrays
    for (int i = 0; i < n; i++)
    {
        xMean += x[i] / n;
        yMean += y[i] / n;
    }

    //Calculates the numerator and denominator for best-fit slope
    for (int i = 0; i < n; i++)
    {
        numeratorSum += y[i] * (x[i] - xMean);
        denominatorSum += x[i] * (x[i] - xMean);
    }

    //Calculate the best-fit slope and y-intercept
    bestfitSlope = numeratorSum / denominatorSum;
    bestfitYintercept = yMean - xMean * bestfitSlope;

    //Calculate the best-fit standard deviation
    for (int i = 0; i < n; i++)
    {
     sumOfResidualsSquared +=
        (y[i]-bestfitYintercept-bestfitSlope*x[i]) *
        (y[i]-bestfitYintercept-bestfitSlope*x[i]);
    }
    sigma = Math.Sqrt(sumOfResidualsSquared/(n-2));
    return new double[] { bestfitYintercept, bestfitSlope, sigma };
}

public static double[] LeastSquaresBestFitLine2(double[] x,double[]y)
{
    //Calculates equation of best-fit line using sums
    int n = x.Length;
    double xSum = 0.0;
    double ySum = 0.0;
    double xySum = 0.0;
    double xSqrSum = 0.0;
    double denominator = 0.0;
    double bNumerator = 0.0;
    double mNumerator = 0.0;
    double bestfitYintercept = 0.0;
    double bestfitSlope = 0.0;
    double sigma = 0.0;
    double sumOfResidualsSquared = 0.0;

    //calculate sums
    for (int i = 0; i < n; i++)
    {
        xSum += x[i];
        ySum += y[i];
        xySum += x[i] * y[i];
        xSqrSum += x[i] * x[i];
    }

    denominator = n * xSqrSum - xSum * xSum;
    bNumerator = xSqrSum * ySum - xSum * xySum;
    mNumerator = n * xySum - xSum * ySum;

    //calculate best-fit y-intercept
    bestfitYintercept = bNumerator / denominator;

    //calculate best-fit slope
    bestfitSlope = mNumerator / denominator;

    //calculate best-fit standard deviation
    for (int i = 0; i < n; i++)
    {
     sumOfResidualsSquared +=
        (y[i]-bestfitYintercept-bestfitSlope*x[i]) *
        (y[i]-bestfitYintercept-bestfitSlope*x[i]);
    }
    sigma = Math.Sqrt(sumOfResidualsSquared/(n-2));
    return new double[] {bestfitYintercept,bestfitSlope,sigma};
}

private static void TestLeastSquaresBestFitLine()
{
    double[] xdata = new double[] {2.0,3.0,4.0,5.0,6.0,7.0 };
    double[] ydata = new double[] {2.9,4.7,6.3,8.4,10.5,12.5};
    double[] results1 = LeastSquaresBestFitLine1(xdata,ydata);
    double[] results2 = LeastSquaresBestFitLine2(xdata,ydata);

    Console.WriteLine("Testing straight line best-fit method \n");
    Console.WriteLine("Results1: (compact version)");
    Console.WriteLine("Best-fit y-intercept = {0}", results1[0]);
    Console.WriteLine("Best-fit slope = {0}", results1[1]);
    Console.WriteLine("Best-fit sigma = {0}", results1[2]);
    Console.WriteLine("Equation for best-fit line:
      y = mx + b = {0}x + {1}\n", results1[1], results1[0]);

    Console.WriteLine("Results2: (traditional version)");
    Console.WriteLine("Best-fit y-intercept = {0}", results2[0]);
    Console.WriteLine("Best-fit slope = {0}", results2[1]);
    Console.WriteLine("Best-fit sigma = {0}", results2[2]);
    Console.WriteLine("Equation for best-fit line:
      y = mx + b = {0}x + {1}", results2[1], results2[0]);
    Console.ReadLine();
}
```

```
RESULTS:

Testing straight line best-fit method

Results1: (compact version)
Best-fit y-intercept = -1.12857142857143
Best-fit slope = 1.92857142857143
Best-fit sigma = 0.190862703084106
Equation for best-fit line:
y = mx + b = 1.92857142857143x + -1.12857142857143

Results2: (traditional version)
Best-fit y-intercept = -1.12857142857143
Best-fit slope = 1.92857142857143
Best-fit sigma = 0.190862703084105
Equation for best-fit line:
y = mx + b = 1.92857142857143x + -1.12857142857143
```

15.3 Weighted Least Squares Fit

The general expressions for a least squares fit have been derived based on the implicit assumption that the errors are uncorrelated with each other and with the independent variables. In other words, the derivation for least squares fit was made under the assumption that all errors have the same significance. There are situations, however, where the confidence in the accuracy of the data values may vary from point to point meaning that the measurements are still uncorrelated but have different uncertainties. For example, there may be some errors that are more important than others or perhaps there might be some kind of unexpected drift in the precision of the measurements being taken due to undetected or spontaneously occurring equipment malfunction. In order to take these factors into consideration, a modified approach needs to be adopted where a weight factor is introduced for each data point. The general expression for the sum of squares of the the residuals is then given by

$$S = \sum_{i=0}^{n} w_i r_i^2 = \sum_{i=0}^{n} w_i [y_i - f(x_i, \mathbf{a})]^2$$

The minimum of the weighted sum of squares is found by setting the gradient of S to zero which leads to the following expressions:

$$-2\sum_{i=0}^{n} w_i \frac{\partial f(x_i, \mathbf{a})}{\partial a_j} r_i = 0, \qquad j = 0, 1, 2, \ldots, n-1$$

15.3.1 Weighted Straight-Line Fit

Finding the best fit straight-line equation to a set of weighted data points (x_i, y_i) with corresponding weights w_i follows immediately in an analogous fashion to the derivation given earlier for a straight-line fit. The weighted sum of squares S of the residuals is now

$$S(m,b) = \sum_{i=0}^{n} w_i [y_i - f(x_i;\mathbf{a})]^2 = \sum_{i=0}^{n} w_i [y_i - b_i - m_i x_i]^2$$

The minimum of the weighted sum of squares is found by setting the gradient of S to zero: $\frac{\partial S}{\partial b} = 0$ and $\frac{\partial S}{\partial m} = 0$ from which we obtain the following set of equations

$$\begin{aligned}
\left(\sum_{i=0}^{n} w_i\right) b \quad &+ \quad \left(\sum_{i=0}^{n} w_i x_i\right) m \quad = \quad \sum_{i=0}^{n} w_i y_i \\
\left(\sum_{i=0}^{n} w_i x_i\right) b \quad &+ \quad \left(\sum_{i=0}^{n} w_i x_i^2\right) m \quad = \quad \sum_{i=0}^{n} w_i x_i y_i
\end{aligned}$$

which can be written in matrix form as

$$\begin{bmatrix} \sum_{i=0}^{n} w_i & \sum_{i=0}^{n} w_i x_i \\ \sum_{i=0}^{n} w_i x_i & \sum_{i=0}^{n} w_i x_i^2 \end{bmatrix} \begin{bmatrix} b \\ m \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^{n} w_i y_i \\ \sum_{i=0}^{n} w_i x_i y_i \end{bmatrix}$$

from which we can solve for b and m to give

$$b = \frac{\left(\sum_{i=0}^{n} w_i x_i^2\right)\left(\sum_{i=0}^{n} w_i y_i\right) - \left(\sum_{i=0}^{n} w_i x_i\right)\left(\sum_{i=0}^{n} w_i x_i y_i\right)}{\left(\sum_{i=0}^{n} w_i\right)\left(\sum_{i=0}^{n} w_i x_i^2\right) - \left(\sum_{i=0}^{n} w_i x_i\right)^2}$$

$$m = \frac{\left(\sum_{i=0}^{n} w_i\right)\sum_{i=0}^{n} w_i x_i y_i - \left(\sum_{i=0}^{n} w_i x_i\right)\left(\sum_{i=0}^{n} w_i y_i\right)}{\left(\sum_{i=0}^{n} w_i\right)\left(\sum_{i=0}^{n} w_i x_i^2\right) - \left(\sum_{i=0}^{n} w_i x_i\right)^2}$$

which can also be expressed as

$$b = \overline{y_w} - m\overline{x_w} \qquad \text{and} \qquad m = \frac{\sum_{i=0}^{n} w_i y_i (x - \overline{x_w})}{\sum_{i=0}^{n} w_i x_i (x_i - \overline{x_w})}$$

where the weighted average of x and y are given by

$$\overline{x_w} = \frac{\sum_{i=0}^{n} w_i x_i}{\sum_{i=0}^{n} w_i} \qquad \text{and} \qquad \overline{y_w} = \frac{\sum_{i=0}^{n} w_i y_i}{\sum_{i=0}^{n} w_i}$$

Implementations in C# of these two sets of best-fit formulas are given below. The `LeastSquaresWeightedBestFitLine1` algorithm follows the more compact version, whereas the `LeastSquaresWeightedBestFitLine2` follows the longer and more traditional version of the two equations. An example illustrating how both of these algorithms may be implemented follows immediately below and includes the results from a sample set of data points.

```
public static double[] LeastSquaresWeightedBestFitLine1(double[] x,
    double[] y, double[] w)
{
    //Calculates equation of best-fit line using short cuts
    int n = x.Length;
    double wxMean = 0.0;
    double wyMean = 0.0;
    double wSum = 0.0;
    double wnumeratorSum = 0.0;
    double wdenominatorSum = 0.0;
    double bestfitYintercept = 0.0;
    double bestfitSlope = 0.0;
    double sigma = 0.0;
    double sumOfResidualsSquared = 0.0;

    //Calculates the sum of the weights w[i]
    for (int i = 0; i < n; i++)
    { wSum += w[i]; }

    //Calculates the mean values for x and y arrays
    for (int i = 0; i < n; i++)
    {
        wxMean += w[i] * x[i] / wSum;
        wyMean += w[i] * y[i] / wSum;
    }

    //Calculates the numerator and denominator for best-fit slope
    for (int i = 0; i < n; i++)
    {
        wnumeratorSum += w[i] * y[i] * (x[i] - wxMean);
        wdenominatorSum += w[i] * x[i] * (x[i] - wxMean);
    }

    //Calculate the best-fit slope and y-intercept
    bestfitSlope = wnumeratorSum / wdenominatorSum;
    bestfitYintercept = wyMean - wxMean * bestfitSlope;

    //Calculate the best-fit standard deviation
    for (int i = 0; i < n; i++)
    {
        sumOfResidualsSquared +=
           w[i]*(y[i]-bestfitYintercept-bestfitSlope*x[i]) *
                (y[i]-bestfitYintercept-bestfitSlope*x[i]);
    }
    sigma = Math.Sqrt(sumOfResidualsSquared / (n - 2));
    return new double[] { bestfitYintercept, bestfitSlope, sigma };
}

public static double[] LeastSquaresWeightedBestFitLine2(double[] x,
    double[] y, double[] w)
{
    //Calculates equation of best-fit line using sums
    int n = x.Length;
    double wSum = 0.0;
    double wxSum = 0.0;
    double wySum = 0.0;
    double wxySum = 0.0;
    double wxSqrSum = 0.0;
    double denominator = 0.0;
    double bNumerator = 0.0;
    double mNumerator = 0.0;
    double bestfitYintercept = 0.0;
    double bestfitSlope = 0.0;
    double sigma = 0.0;
    double sumOfResidualsSquared = 0.0;

    //calculate sums
    for (int i = 0; i < n; i++)
    {
        wSum += w[i];
        wxSum += w[i]*x[i];
        wySum += w[i]*y[i];
        wxySum += w[i]*x[i] * y[i];
        wxSqrSum += w[i]* x[i] * x[i];
    }

    denominator = wSum * wxSqrSum - wxSum * wxSum;
    bNumerator = wxSqrSum * wySum - wxSum * wxySum;
    mNumerator = wSum * wxySum - wxSum * wySum;

    //calculate best-fit y-intercept
    bestfitYintercept = bNumerator / denominator;

    //calculate best-fit slope
    bestfitSlope = mNumerator / denominator;

    //calculate best-fit standard deviation
    for (int i = 0; i < n; i++)
    {
        sumOfResidualsSquared +=
           w[i]*(y[i]-bestfitYintercept-bestfitSlope*x[i]) *
                (y[i]-bestfitYintercept-bestfitSlope*x[i]);
    }
    sigma = Math.Sqrt(sumOfResidualsSquared / (n - 2));
    return new double[] { bestfitYintercept, bestfitSlope, sigma };
}

private static void TestLeastSquaresWeightedBestFitLine()
{
  double[] xdata = new double[] { 0.0, 2.0, 4.0, 6.0 };
  double[] ydata = new double[] { 10.0, 15.0, 18.0, 25.0 };
  double[] weights = new double[] { 1.0, 5.0, 10.0, 1.0 };
  double[] results1 =
    LeastSquaresWeightedBestFitLine1(xdata, ydata, weights);
  double[] results2 =
    LeastSquaresWeightedBestFitLine2(xdata, ydata, weights);
  Console.WriteLine("Testing weighed straight line best-fit method");
  Console.WriteLine("Results1: (compact version)");
  Console.WriteLine("Best-fit y-intercept = {0}", results1[0]);
  Console.WriteLine("Best-fit slope = {0}", results1[1]);
  Console.WriteLine("Best-fit sigma = {0}", results1[2]);
  Console.WriteLine("Equation for weighted best-fit line:
    y = mx + b = {0}x + {1}\n", results1[1], results1[0]);
  Console.WriteLine("Results2: (traditional version)");
  Console.WriteLine("Best-fit y-intercept = {0}", results2[0]);
  Console.WriteLine("Best-fit slope = {0}", results2[1]);
  Console.WriteLine("Best-fit sigma = {0}", results2[2]);
  Console.WriteLine("Equation for weighted best-fit line:
    y = mx + b = {0}x + {1}", results2[1], results2[0]);
  Console.ReadLine();
}

RESULTS:

Testing weighted straight line best-fit method

Results1: (compact version)
Best-fit y-intercept = 10.2985074626866
Best-fit slope = 2.05223880597015
Best-fit sigma = 1.78967609395506
Equation for weighted best-fit line:
y = mx + b = 2.05223880597015x + 10.2985074626866

Results2: (traditional version)
Best-fit y-intercept = 10.2985074626866
Best-fit slope = 2.05223880597015
Best-fit sigma = 1.78967609395506
Equation for weighted best-fit line:
y = mx + b = 2.05223880597015x + 10.2985074626866
```

15.4 Linear Regression

In general, regression analysis refers to a collection of techniques for the modeling and analyzing of numerical data consisting of a dependent variable and one or more independent variables. The dependent variable in the regression equation is modeled as a function of the independent variables, and any corresponding parameters. The parameters are estimated so as to give a *best fit* of the data and are usually evaluated by using the least squares method. More advanced techniques of regression analysis, which are beyond the scope of this book, can be used for prediction, including forecasting of time-series data, inference, hypothesis testing, and modeling of causal relationships.

Linear regression is a form of regression analysis in which the relationship between one or more independent variables and another variable, called the dependent variable, is modeled by a least squares function, called a linear regression equation. This function is a linear combination of one or more model parameters, called regression coefficients. A linear regression equation with one independent variable represents a straight line when the predicted, or dependent variable from the regression equation, is plotted against the independent variable. However, note that *linear* does not refer to this straight line, but rather to the way in which the regression coefficients occur in the regression equation.

The primary application of linear least squares is in data fitting. Given a set of n data points $x_0, x_1, \ldots, x_{n-1}$ of an independent variables where x_i may be scalar or vector quantities, and given a model function $y = f(x, \mathbf{a})$, with $m-1$ parameters $\mathbf{a} = (a_0, a_1, \ldots, a_m)$, it is desired to find the parameters a_j such that the model function *best* fits the data. In linear least squares, linearity is meant to be with respect to parameters a_j, so

$$f(x, \mathbf{a}) = \sum_{j=0}^{m} a_j f_j(x)$$

where the functions $f_j(x)$ may be nonlinear with respect to the variable x. Ideally, the model function fits the data exactly, so $y_i = f(x_i, \mathbf{a}) = \sum_{j=0}^{m} a_j f_j(x_i)$ for all $i = 0, 1, \ldots, n-1$. This is usually not possible in practice, as there are often more data points than there are parameters to be determined. The typical approach that is usually chosen is to find the minimal possible value of the sum of squares of the residuals $r_i(\mathbf{a}) = y_i - f(x_i, \mathbf{a}),\ (i = 0, 1, \ldots, n)$ so as to minimize the function

$$S(\mathbf{a}) = \sum_{i=0}^{n} r_i^2(\mathbf{a})$$

After substituting for r_i and then for f, this minimization problem becomes the quadratic minimization problem above with $X_{ij} = f_j(x_i)$, and the best fit can be found by solving the normal equations.

S is minimized when its gradient with respect to each parameter is equal to zero. The elements of the gradient vector are the partial derivatives of S with respect to the parameters:

$$\frac{\partial S}{\partial a_k} = 2 \sum_{i=0}^{n} r_i \frac{\partial r_i}{\partial a_k} = 0$$

Since $r_i = y_i - f(x_i, \mathbf{a}) = y_i - \sum_{j=0}^{m} X_{ij} a_j$, the partial derivatives are

$$\frac{\partial r_i}{\partial a_k} = \frac{\partial}{\partial a_k}(y_i - \sum_{j=0}^{m} X_{ij} a_j) = -X_{ik}$$

Substitution of the expressions for the residuals and the derivatives into the gradient equations gives

$$\frac{\partial S}{\partial a_k} = 2 \sum_{i=0}^{n} (y_i - \sum_{j=0}^{m} X_{ij} a_j)(-X_{ik}) = 0$$

Upon some rearrangement, we arrive at the normal equations

$$\sum_{j=0}^{m} \sum_{i=0}^{n} X_{ij} X_{ik} a_j = \sum_{i=0}^{n} X_{ik} y_i$$

Since $X_{ij} = f_j(x_i)$, we can simplify the expressions above even further to read

$$\sum_{j=0}^{m}\sum_{i=0}^{n} f_j(x_i) f_k(x_i) a_j = \sum_{i=0}^{n} f_k(x_i) y_i$$

The above expression can be rewritten in a more compact matrix form by recognizing the matrix elements written as

$$F_{jk} = \sum_{i=0}^{n} f_j(x_i) f_k(x_i) \qquad \text{and} \qquad B_k = \sum_{i=0}^{n} f_k(x_i) y_i$$

so that

$$\sum_{j=0}^{m} F_{jk} A_j = B_k \text{ or more simply } FA = B \text{ and therefore, } A = F^{-1}B$$

The implementation of linear regression in C# is now a very straight forward process. First, we need to define a delegate function to act as our model function that takes a double variable `x` as its input parameter. Then we construct a public static method, called `LinearRegression`, that returns a `RVector` object whose components are the coefficients of a user supplied set of basis functions. Contained inside the `LinearRegression` method, are the coefficient `RMatrix` F and `RVector` B. The equation $A = F^{-1}B$ is then solved using the `GaussJordan` method discussed in Chapter 8. Also included is a calculation of the standard deviation in order to provide us with a rough estimation of the error.

```
public delegate double ModelFunction(double x);

public static RVector LinearRegression(double[] x, double[] y,
    ModelFunction[] f, out double sigma)
{
    //m = number of data points
    int m = f.Length;
    RMatrix Fmatrix = new RMatrix(m, m);
    RVector Bvector = new RVector(m);
    // n = number of linear terms in the regression equation
    int n = x.Length;

    //Calculate the B vector entries
    for (int k = 0; k < m; k++)
    {
        Bvector[k] = 0.0;
        for (int i = 0; i < n; i++)
        { Bvector[k] += f[k](x[i]) * y[i]; }
    }

    //Calculate the F matrix entries
    for (int j = 0; j < m; j++)
    {
        for (int k = 0; k < m; k++)
        {
            Fmatrix[j, k] = 0.0;
            for (int i = 0; i < n; i++)
            { Fmatrix[j, k] += f[j](x[i]) * f[k](x[i]); }
        }
    }

    // FA = B so A = F^{-1}B
    RVector Avector = GaussJordan(Fmatrix, Bvector);

    // Calculate the standard deviation to estimate error
    double sumOfResidualsSquared = 0.0;
    for (int i = 0; i < n; i++)
    {
        double sum = 0.0;
        for (int j = 0; j < m; j++)
        { sum += Avector[j] * f[j](x[i]); }
        sumOfResidualsSquared += (y[i] - sum) * (y[i] - sum);
    }
    sigma = Math.Sqrt(sumOfResidualsSquared / (n - m));
    return Avector;
}

private static double f0(double x) {return 1.0;}
private static double f1(double x) {return x;}
private static double f2(double x) {return x * x;}
private static double f3(double x) {return x * x * x;}

private static void TestLinearRegression()
{
  Console.WriteLine("Testing Linear Regression Fit for basis");
  Console.WriteLine("function of a polynomial of order 3\n");
  double[] x = new double[] { 0, 1, 2, 3, 4, 5 };
  double[] y = new double[] { 4, 2, 8, 8, 6, 4 };
  double sigma = 0.0;
  ModelFunction[] f = new ModelFunction[] { f0, f1, f2, f3 };
  RVector results = LinearRegression(x, y, f, out sigma);
  Console.WriteLine("Order of polynomial m = 3:
    A0 + A1*x + A2*x^2 + A3*x^3 where");
  Console.WriteLine("A0 = {0}", results[0]);
  Console.WriteLine("A1 = {0}", results[1]);
  Console.WriteLine("A2 = {0}", results[2]);
  Console.WriteLine("A3 = {0}", results[3]);
  Console.WriteLine("Standard deviation = {0}\n\n", sigma);
}
```

RESULTS:

```
Testing Linear Regression Fit for a set of basis
functions consisting of a polynomial of order 3

Order of polynomial m = 3: A0 + A1*x + A2*x^2 + A3*x^3 where
A0 = 3.34920634920648
A1 = -0.351851851852306
A2 = 1.3730158730161
A3 = -0.259259259259288
Standard deviation = 2.31626409657434
```

15.4.1 Polynomial Fit

The linear regression algorithm, discussed in the previous section, is a general algorithm for finding best fits to a given set of data points provided we can specify a set of appropriate basis functions and linear parameters under which the prescribed calculations can take place. For example, a polynomial fit is just a special case of the more general linear regression method where the basis functions are now given by $f_j(x) = x^j$ and $j = 0,1,2,\ldots,n-1$ while the linear coefficients are given by $a_0, a_1, ...a_{n-1}$ so that in general the model function is given by $f(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$. In this case, the matrix F and vector B in the normal function are given by

$$F_{jk} = \sum_{i=0}^{n} x_i^{j+k} \qquad \text{and} \qquad B_k = \sum_{i=0}^{n} x_i^k \, y_i$$

An implementation of the polynomial fit in C# can then be written down as shown below.

```
public static RVector PolynomialFit(double[] x, double[] y, int m,
    out double sigma)
{
    //m = number of data points which in this case
    //for polynomials = order or degree of polynomial P_m(x)
    m++; //minor adjust
    RMatrix Fmatrix = new RMatrix(m, m);
    RVector Bvector = new RVector(m);
    // n = number of linear terms in the regression equation
    int n = x.Length;

    //Calculate the B vector entries
    for (int k = 0; k < m; k++)
    {
        Bvector[k] = 0.0;
        for (int i = 0; i < n; i++)
        { Bvector[k] += Math.Pow(x[i], k) * y[i]; }
    }

    //Calculate the F matrix entries
    for (int j = 0; j < m; j++)
    {
        for (int k = 0; k < m; k++)
        {
            Fmatrix[j, k] = 0.0;
            for (int i = 0; i < n; i++)
            { Fmatrix[j, k] += Math.Pow(x[i], j + k); }
        }
    }

    // FA = B so A = F^{-1}B
    RVector Avector = GaussJordan(Fmatrix, Bvector);

    // Calculate the standard deviation to estimate error
    double sumOfResidualsSquared = 0.0;
    for (int i = 0; i < n; i++)
    {
        double sum = 0.0;
        for (int j = 0; j < m; j++)
        { sum += Avector[j] * Math.Pow(x[i], j); }
        sumOfResidualsSquared += (y[i] - sum) * (y[i] - sum);
    }
    sigma = Math.Sqrt(sumOfResidualsSquared / (n - m));
    return Avector;
}

private static void TestPolynomialFit()
{
    Console.WriteLine("Testing Polynomial Fit");
    Console.WriteLine("using an actual polynomial of order 3");
    double[] x = new double[] { 0, 1, 2, 3, 4, 5 };
    double[] y = new double[] { 4, 2, 8, 8, 6, 4 };
    int polynomialOrder = 3;
    double sigma = 0.0;
    RVector results=PolynomialFit(x,y,polynomialOrder,out sigma);
    Console.WriteLine("Order of polynomial m = 3:
         A0 + A1*x + A2*x^2 + A3*x^3 where");
    Console.WriteLine("A0 = {0}", results[0]);
    Console.WriteLine("A1 = {0}", results[1]);
    Console.WriteLine("A2 = {0}", results[2]);
    Console.WriteLine("A3 = {0}", results[3]);
    Console.WriteLine("Standard deviation = {0}\n\n", sigma);
}

RESULTS:

Testing Polynomial Fit
using an actual polynomial of order 3

Order of polynomial m = 3: A0 + A1*x + A2*x^2 + A3*x^3 where
A0 = 3.34920634920648
A1 = -0.351851851852306
A2 = 1.3730158730161
A3 = -0.259259259259288
Standard deviation = 2.31626409657434
```

15.4.2 Exponential Fit

In an exponential function fit, the model function $f(x)$ takes on the general form: $f(x) = y = ae^{cx}$. Since the coefficients a and c are nonlinear, we cannot directly apply the linear regression methods discussed earlier in this chapter. However, by taking the natural logarithm of both sides, we get $\ln y = cx + \ln a$ which has the general form of the slope-intercept equation of a straight line: $Y = mx + b$ where slope $= m = c$ and y-intercept $= b = \ln a$. Therefore, by using $(x_i, \ln y_i)$ instead of (x_i, y_i), the linear regression methods described in this chapter can now also be used to obtain the best fit solution of exponential functions.

With a sufficiently large number of data points, an argument can be made for calculating exponential fits with weights. In this scenario, the fit function becomes

$$F(x) = \ln(f(x)) = \ln(ae^{cx}) = \ln a + cx$$

The residuals of the logarithm fit are then given by

$$R_i = \ln y_i - \ln F(x_i) = \ln y_i - \ln a - cx_i$$

The residuals r_i used in fitting the original data are given by

$$r_i = y_i - f(x_i) = y_i - ae^{cx}$$

Combining these two equations for r_i and R_i yields

$$R_i = \ln\left(1 - \frac{r_i}{y_i}\right)$$

In the limit where $r_i << y_i$ we can use a Taylor series expansion to approximate $R_i \approx r_i/y_i$. As a result, in minimizing $\sum_{i=0}^{n} R_i^2$, we must introduce a weight factor $1/y_i$. Therefore, we need to apply the weights $w_i = y_i$ when fitting the model function to the data points $(x_i, \ln y_i)$. An implementation of an exponential fit in C# can therefore be done as shown below.

```
private static void TestExponentialFit()
{
  Console.WriteLine("Testing Exponential Fit");
  Console.WriteLine("  Original equation: y  = a e^{cx}");
  Console.WriteLine("Linearized equation: ln(y) = cx + ln(a)\n");
  Console.WriteLine("which has the form: Y=mx+b of a straight line");
  double[] x = new double[] { 1, 2, 3, 4, 5 };
  double[] y = new double[] { 3.5, 6.2, 9.5, 15.3, 20.4 };
  double[] logy = new double[]
      {1.25276,1.82455,2.25129,2.72785,3.01553};

  double[] results = LeastSquaresBestFitLine1(x, logy);
  Console.WriteLine("Results without weights, just (x,ln(y)): ");
  Console.WriteLine("Best-fit y-intercept b=ln(a)= {0}",results[0]);
  Console.WriteLine("Best-fit slope = m = c ={0}", results[1]);
  Console.WriteLine("Best-fit sigma = {0}", results[2]);
  Console.WriteLine("Equation for weighted best-fit exponential:");
  Console.WriteLine("y = a e^(cx) = {0}exp({1}x)\n",
        Math.Exp(results[0]),results[1]);

  double[] resultsWt = LeastSquaresWeightedBestFitLine1(x, logy, y);
  Console.WriteLine("Results with weights w=y. Calc (x,ln(y),w)):");
  Console.WriteLine("Best-fit y-intercept b=ln(a)={0}",resultsWt[0]);
  Console.WriteLine("Best-fit slope = m = c = {0}",resultsWt[1]);
  Console.WriteLine("Best-fit sigma = {0}",resultsWt[2]);
  Console.WriteLine("Equation for weighted best-fit exponential:");
  Console.WriteLine("y = a e^(cx) = {0}exp({1}x)\n",
        Math.Exp(resultsWt[0]),resultsWt[1]);
  Console.ReadLine();
}
```

```
RESULTS:
Testing Exponential Fit
Original equation: y  = a e^{cx}
Linearized equation: ln(y) = cx + ln(a)
which is in the form: Y = mx + b for a straight line

Results without weights, just (x,ln(y)):
Best-fit y-intercept b = ln(a) = 0.885744
Best-fit slope = m = c =0.442884
Best-fit sigma = 0.0857772094051406
Equation for weighted best-fit exponential:
y = a e^(cx) = 2.4247877626437exp(0.442884x)

Results with weights w = y. Calc (x,ln(y),w)):
Best-fit y-intercept b = ln(a)= 0.971492075896069
Best-fit slope = m = c = 0.419085390333843
Best-fit sigma = 0.0979232048349948
Equation for weighted best-fit exponential:
y = a e^(cx) = 2.64188341057344exp(0.419085390333843x)
```

Other types of model functions which are nonlinear in nature can sometimes also be artificially linearized so that we may then apply the linear regression methods introduced in this chapter in order to obtain a best fit solution to a set of corresponding data points. For example, consider the another model function $f(x)$ that takes on the form

$$f(x) = y = ax^c$$

By taking the logarithm of both sides, we get $\ln y = c\ln x + \ln a$ which is in the intercept-slope form of an equation for a straight line $Y = mX + b$ where $Y = \ln y$, $X = \ln x$, slope $= m = c$ and y-intercept $b = \ln a$.

As a final example of data linearization, consider the nonlinear model function given by

$$f(x) = y = \frac{1}{a + cx}$$

This nonlinear equation can be converted to linear form by substituting $Y = 1/y$ in which case we obtain $Y = cx + a$ which is in the intercept-slope form of an equation for a straight line $Y = mx + b$ where $Y = 1/y$, slope $= m = c$ and y-intercept $b = a$.

15.5 The χ^2 Test for Goodness of Fit

There are two types of χ^2 tests:

1. The χ^2 test for goodness of fit establishes whether or not an observed frequency distribution differs from a theoretical distribution. In other words, the

χ^2 test for goodness of fit compares the expected and observed values to determine how well an experimenter's predictions fit the data or how well a statistical model fits a set of observations. Such measures can be used in statistical hypothesis testing such as to test for normality of residuals, or to test whether two samples are drawn from identical distributions.

2. The χ^2 test for independence assesses whether paired observations on two variables are independent from each other. In other words, the χ^2 for independence is used to determine how independent two variables of a sample really are. In this context independence means that the two variables are not related.

Data used in a χ^2 test must

1. be randomly drawn from the population
2. be reported in raw counts of frequency
3. have independently measured variables
4. have observed frequencies which are not too small, and
5. be mutually exclusive.

The first step in the chi-square test is to calculate the chi-square statistic. The chi-square statistic is calculated by finding the difference between each observed and theoretical frequencies for each possible outcome, squaring them, dividing each by the theoretical frequency, and taking the sum of the results. To calculate frequency, the results of any repeated measurements are first grouped in k bins, where $k = 1,2,\ldots,n$. Let O_k denote the number of results observed in bin k, based on some assumed distribution. Let E_k be the expected number of measurements in bin k. Then the value of the χ^2 test statistic is given by

$$\chi^2 = \sum_{k=1}^{n} \frac{(O_k - E_k)^2}{E_k}$$

In actual practice, the values for O_k are obtained from a histogram of the observed frequency values. If n measurements, x_k, are made of the quantity x, we can truncate the data to a common least count and group the observations into frequencies of identical observations to make a histogram. Let us assume that k runs from 1 to n so there are n possible different values of x_k and let us call the frequency of observations, or number of counts in each histogram bin, $h(x_k)$ for each different measured value of x_k. If the probability for observing the value x_k in any random measurement is denoted by $P(x_k)$, then the expected number of such observations is $E(x_k) = nP(x_k)$ where n is the total number of bins contained in the corresponding frequency histogram. Using $O_k = h(x_k)$ then χ^2 can also be expressed as

$$\chi^2 = \sum_{k=1}^{n} \frac{[h(x_k) - nP(x_k)]^2}{nP(x_k)}$$

If $\chi^2 = 0$ then the agreement between the expected and observed distributions is perfect. That is, $O_k = E_k$ for all bins k, a situation most unlikely to occur. If $\chi^2 >> n$, which physically means that χ^2 is significantly greater than the total number of bins, then the observed and expected numbers differ significantly and the agreement between O_k and E_k is unacceptable and we reject the assumed distribution. If $\chi^2 \leq n$ then the observed and expected distributions agree about as well as can be expected.

To illustrate the use of the chi-square statistic, I chose to generate 2000 randomly drawn data points that supposedly follow a binomial distribution and then compare the result against another set of 2000 data points calculated directly from the binomial distribution. The frequency histogram used to distribute the data consists of a total of 20 bins. The C# code to carry out this calculation is presented below where I also utilized C# code from Chapters 10 and 14 on random probability distributions and special functions, respectively. Because of the randomness of the calculations, a slightly different χ^2 is obtained each time the program runs. For completeness, the result for a reduced χ^2 is also given. The reduced χ^2 is just the regular χ^2 divided by the number of degrees of freedom of the experiment. The number of degrees of freedom is just the number of sample frequencies minus the number of constraints or parameters in the experiment. In the example below, a frequency histogram was created containing 20 bins and so the number of degrees of freedom is 20. The number of constraints or parameters is 1. The resulting $\chi^2 = 19.913$, or equivalently, the reduced χ^2 value of 1.04807 supports the hypothesis that the observed data is indeed comparable to the expected data.

```
private static void TestChiSquareStatistic()
{
  int nBins = 20;      //number of bins for histogram
  int nPoints = 2000; //number of data points

  //Create an array of 2000 random data points
  //that supposedly follow a binomial distribution
  double[] RandomData = NextBinomial(20, 0.5, nPoints);

  //Distribute the 2000 random data points into a
  //histogram consisting of 20 bins
  double[] RandomDistributionValues=makeHistogram(RandomData,nBins);

  //Setup arrays to hold data
  double[] xdata = new double[nBins];
  double[] ydata = new double[nBins];

  double[] Observed = new double[nBins];
  double[] Expected = new double[nBins];

  double[] ydistribution = new double[nBins];

  //Assign random data and binomial distribution data
  //to their proper arrays
  for (int i = 0; i < nBins; i++)
  {
    xdata[i] = i;
    ydata[i] = (double)RandomDistributionValues[i];
    ydistribution[i] = BinomialPDF(i, 20, 0.5);
  }

  //Calculate the normalization factor
  double normalizationFactor = dataMax(ydata)/dataMax(ydistribution);

  //Assign observed and expected data arrays and adjust
  //the expected data array by the normalization factor
  for (int i = 0; i < nBins; i++)
  {
    Observed[i] = ydata[i];
    Expected[i] = ydistribution[i] * normalizationFactor;
  }

  //Calculate the number of degrees of freedom
  int NDegreesOfFreedom = nBins - 1;

  //Calculate chi-square value
  double chi_square = 0.0;
  for (int i = 0; i < nBins; i++)
  {
    chi_square += ((Observed[i] - Expected[i]) * (Observed[i] -
        Expected[i])) / (Expected[i]);
  }

  Console.WriteLine("\nChi-Square Statistic Test\n");
  Console.WriteLine("A random sample of 2000 points supposedly ");
  Console.WriteLine("following a Binomial Distribution is ");
  Console.WriteLine("compared against a set of calculated ");
  Console.WriteLine("binomial distribution values using the ");
  Console.WriteLine("chi-square statistic. ");
  Console.WriteLine("Frequency histogram used has 20 bins.");
  Console.WriteLine("Results: ");
  Console.WriteLine("chi-square = {0}", chi_square);
  Console.WriteLine("Reduced chi-square = {0}", chi_square/
      NDegreesOfFreedom);
  Console.ReadLine();
}

OUTPUT:
A random sample of 2000 points supposedly following a Binomial
Distribution is compared against a set of calculated binomial
distribution values using the chi-square statistic.
Frequency histogram used has 20 bins.
Results:
        chi-square = 19.913500568919
Reduced chi-square = 1.04807897731153
```

16

Ordinary Differential Equations

16.1 Introduction

Differential equations consist of a broad field of study in pure and applied mathematics, the natural sciences and engineering. All of these disciplines are concerned with the properties of differential equations of various types. For example, pure mathematics focuses on the existence and uniqueness of solutions, while applied mathematics emphasizes the rigorous justification of the methods for approximating solutions. Moreover, differential equations play an important role in modeling virtually every physical, technical, or biological process, from celestial motion to bridge design, to interactions between neurons. In general, differential equations may be broken down into several categories:

- An *ordinary differential equation* (ODE) is a differential equation in which the unknown function is a function of a single independent variable.
- A *partial differential equation* (PDE) is a differential equation in which the unknown function is a function of multiple independent variables and their partial derivatives.
- A *delay differential equation* (DDE) is a differential equation in which the derivative of the unknown function at a certain time is given in terms of the values of the function at previous times.
- A *stochastic differential equation* (SDE) is a differential equation in which one or more of the terms is a stochastic process, thus resulting in a solution which is itself a stochastic process.
- A *differential algebraic equation* (DAE) is a differential equation comprising differential and algebraic terms, given in implicit form.

Each of these categories can be divided into linear and nonlinear sub-categories. A differential equation is said to be *linear* if the dependent variable and all its derivatives appear to the power 1 and there are no products or functions of the dependent variable. Otherwise the differential equation is said to be *nonlinear*. Unfortunately and with a few exceptions, most differential equations of interest that originate from describing or modeling natural phenomena and other real-life problems are not always directly solvable or have nice closed form analytic solutions and therefore cannot be solved exactly. Instead, such solutions can only be approximated using numerical methods. As a result, the contents of this chapter will focus on the most popular and fundamental numerical methods for solving differential equations and illustrate how these methods may be coded in C#.

In mathematics, an ordinary differential equation is a relation that contains functions of only one independent variable, and one or more of its derivatives with respect to that variable. A simple example is Newton's second law of motion, which leads to the differential equation

$$m\frac{d^2x(t)}{dt^2} = F(x(t))$$

for the motion of a particle of mass m. In general, the force F depends upon the position of the particle $x(t)$ at time t, and thus the unknown function $x(t)$ appears on both sides of the differential equation, as is indicated in the notation $F(x(t))$. Ordinary differential equations are distinguished from partial differential equations, which involve partial derivatives of several variables and will be addressed to some extent in the next chapter.

When a differential equation of order n has the form

$$F\left(x, y, y', y'', \ldots, y^{(n)}\right) = 0$$

it is called an *implicit* differential equation whereas the form

$$F\left(x, y, y', y'', \ldots, y^{(n-1)}\right) = y^{(n)}$$

is called an *explicit* differential equation.

A boundary value problem (BVP) is an ordinary differential equation together with a set of additional restraints, called the boundary condition, which has values assigned on the physical boundary of the domain in which the problem is specified. A solution to a boundary value problem is a solution to the differential equation which also satisfies the boundary conditions. An initial value problem (IVP) is an ordinary differential equation together with specified value, called the initial condition, of the unknown function at a given point in the domain of the solution. A more mathematical way to picture the difference between an initial value problem and a boundary value problem is that an initial value problem has all of the conditions specified at the same value of the independent variable in the equation whereas a boundary value problem has conditions specified at the extremes of the independent variable. A major difference between IVPs and BVPs is that there may be an issue of existence and uniqueness.

16.2 Euler Method

The Euler method, named after the famous Swiss mathematician Leonhard Euler, is a first order numerical procedure for solving ordinary differential equations with a given initial value. It is the most basic kind of explicit method for numerical integration of ordinary differential equations. The Euler method seeks to approximate the solution of the initial value problem

$$y'(x) = f(x, y(x)) \qquad \text{where} \qquad y(x_0) = y_0$$

by using the first two terms of the Taylor expansion of $y(x)$ about $x = x_0$, which represents the linear approximation around the point $(x_0, y(x_0))$

$$y(x) \approx y(x_0) + \frac{y'(x_0)}{1!}(x - x_0) + \frac{y''(x_0)}{2!}(x - x_0)^2 + \frac{y^3(x_0)}{3!}(x - x_0)^3 + \cdots$$

By truncating the series after two terms, we obtain the approximation

$$y(x) \approx y(x_0) + y'(x_0)(x - x_0)$$

By setting $h = x - x_0$ and observing that $y'(x_0) = f(x_0, y(x_0)) = f(x_0, y_0)$ we then obtain the expression

$$y(x_0 + h) \approx y_0 + f(x_0, y_0)\,h$$

which is the first step in the Euler approximation method. Proceeding in a similar manner, the equations can be further generalized for the n and $n+1$ term to read

$$\begin{aligned} x_{n+1} &= x_n + h \\ y_{n+1} &= y_n + f(x_n, y_n)\,h \end{aligned}$$

The Euler method is explicit, meaning that the solution y_{n+1} is an explicit function of y_i for $i \leq n$. An implementation of the Euler method in C# is given below. For flexibility we again defined a general delegate function $f(x,y)$ so that users may then input any function they choose. This version of the Euler algorithm takes the delegate function f, the initial values x_0 and y_0, the increment h along the x-axis and the point position x where we want to calculate a solution as input parameters. For testing purposes, I chose the differential equation given by $y' = y\cos(x)$ with the initial condition $x_0 = 0$ and $y_0 = 1$ for which the exact solution is easily calculated to be $y = \exp(\sin(x))$. In order to compare differences in the output of additional upcoming solution methods, these same test equations and initial conditions were also used unless indicated otherwise.

```
public delegate double Function(double x, double y);

static double f(double x, double y)
{ return y*Math.Cos(x); }
```

```
public static double ODE_Euler(Function f, double x0,
                double y0, double h, double x)
{
    double xnew, ynew, result = double.NaN;
    if (x <= x0) result = y0;
    else if (x > x0)
    {
        do
        {
            if (h > x - x0) h = x - x0;
            ynew = y0 + f(x0, y0) * h;
            xnew = x0 + h;
            x0 = xnew;
            y0 = ynew;
        } while (x0 < x);
        result = ynew;
    }
    return result;
}

static void TestEuler()
{
 double h = 0.001;
 double x0 = 0.0;
 double y0 = 1.0;
 Console.WriteLine("\n Results from Euler's method with h={0}\n",h);
 double result = y0;
 for (int i = 0; i < 11; i++)
 {
   double x = 0.1 * i;
   result = ODE_Euler(f, x0, result, h, x);
   double exact = Math.Exp(Math.Sin(x));
   if (i % 5 == 0)
   Console.WriteLine("x={0:n1},y={1:e12},exact={2:e12}",x,result,
       exact);
   x0 = x;
 }
}

Results from Euler's method with h = 0.001
x = 0.0, y = 1.000000000000e+000, exact = 1.000000000000e+000
x = 0.5, y = 1.614873480863e+000, exact = 1.615146296442e+000
x = 1.0, y = 2.319466352295e+000, exact = 2.319776824716e+000
```

16.3 Runge-Kutta Methods

The Runge-Kutta methods are an important family of implicit and explicit iterative methods for approximating solutions of ordinary differential equations. These techniques were developed around the year 1900 by the German mathematicians C. Runge and M.W. Kutta and offer a more precise alternative to using Euler's method, which can sometimes produce significant truncation errors and is also prone to numerical instabilities.

16.3.1 Second-Order Runge-Kutta Method

The second-order Runge-Kutta method is derived by taking one additional term in the Taylor series expansion of $y(x)$ before truncating the remaining terms. As with the Euler method we begin by seeking an approximate solution to the initial value problem

$$y'(x) = f(x, y(x)) \qquad \text{where} \qquad y(x_0) = y_0$$

However, instead of taking just the first two terms of the Taylor expansion of $y(x)$ about $x = x_0$

$$y(x) \approx y(x_0) + \frac{y'(x_0)}{1!}(x - x_0) + \frac{y''(x_0)}{2!}(x - x_0)^2 + \frac{y^3(x_0)}{3!}(x - x_0)^3 + \cdots$$

we now truncate the series after three terms, and thus retain the following terms

$$y(x) \approx y(x_0) + y'(x_0)(x - x_0) + \frac{y''(x_0)}{2!}(x - x_0)^2$$

which can also be written as

$$y_{n+1} \approx y_n + f(x_n, y_n)h + \frac{1}{2!}f'(x_n, y_n)\,h^2$$

In addition, we do one more thing. The main reason that Euler's method has such a large truncation error per step is that in evolving the solution from x_n to x_{n+1} the method only evaluates derivatives at the beginning of the interval, x_n. The method is therefore very asymmetric with respect to the beginning and the end of the interval. We can construct a more symmetric integration method by making a Euler-like trial step to the midpoint of the interval, and then taking the values of both x and y at the midpoint to make the real step across the interval. To be more exact, we introduce two parameters k_1 and k_2 such that

$$\begin{aligned} k_1 &= f(x_n, y_n)\,h \\ k_2 &= f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1)\,h \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned}$$

where this symmetrization cancels out the first-order error, making the method second-order. Because of the way this method is derived, the Euler method is also known as the first-order Runge-Kutta method. The implementation of this second-order Runge-Kutta method in C# is given below.

```
public delegate double Function(double x, double y);
static double f(double x, double y)
{ return y*Math.Cos(x); }
```

```
public static double ODE_RungeKutta2(Function f, double x0,
                double y0, double h, double x)
{
    double xnew, ynew, k1, k2, result = double.NaN;
    if (x == x0) result = y0;
    else if (x > x0)
    {
        do
        {
            if (h > x - x0) h = x - x0;
            k1 = h * f(x0, y0);
            k2 = h * f(x0 + 0.5 * h, y0 + 0.5 * k1);
            ynew = y0 + k2;
            xnew = x0 + h;
            x0 = xnew;
            y0 = ynew;
        } while (x0 < x);
        result = ynew;
    }
    return result;
}

static void TestRungeKutta2()
{
  double h = 0.001;
  double x0 = 0.0;
  double y0 = 1.0;
  Console.WriteLine("\nResults from the 2nd-order Runge-Kutta method
      with h={0}\n",h);
  double result = y0;
  for (int i = 0; i < 11; i++)
  {
    double x = 0.1 * i;
    result = ODE_RungeKutta2(f, x0, result, h, x);
    double exact = Math.Exp(Math.Sin(x));
    if (i % 5 == 0)
    Console.WriteLine("x={0:n1},y={1:e12},exact={2:e12}",x,result,
        exact);
    x0 = x;
  }
}
```

```
Results from the 2nd-order Runge-Kutta method with h = 0.001
x = 0.0, y = 1.000000000000e+000, exact = 1.000000000000e+000
x = 0.5, y = 1.615146255938e+000, exact = 1.615146296442e+000
x = 1.0, y = 2.319776862722e+000, exact = 2.319776824716e+000
```

16.3.2 Fourth-Order Runge-Kutta Method

Although the second-order Runge-Kutta method provides better precision than Euler's method, it is not used that much in practice. Instead, because of the greater accuracy and precision that can be obtained, the fourth-order Runge-Kutta method is the preferred numerical method of choice for numerically approximating the solutions of first-order differential equations. The derivation concepts are the same as before, except that we now take more terms in the Taylor series expansion of $y(x)$ about $x = x_0$. Since retaining more terms in series expansion only makes the derivation steps longer and more tedious, the fourth-order Runge-Kutta method can be summarized by the following equations.

$$\begin{aligned}
k_1 &= f(x_n, y_n)\,h \\
k_2 &= f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1)\,h \\
k_3 &= f(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2)\,h \\
k_4 &= f(x_n + h, y_n + k_3)\,h \\
y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(h^5)
\end{aligned}$$

Note that the next value y_{n+1} is determined by the present value y_n plus the product of the size of the interval h and an average of four slopes with greater weight being given to the slopes at the midpoint. The implementation of the fourth-order Runge-Kutta method in C# is given below.

```
public delegate double Function(double x, double y);

static double f(double x, double y)
{ return y*Math.Cos(x); }

public static double ODE_RungeKutta4(Function f, double x0,
                double y0, double h, double x)
{
    double xnew, ynew, k1, k2, k3, k4, result = double.NaN;
    if (x == x0) result = y0;
    else if (x > x0)
    {
        do
        {
            if (h > x - x0) h = x - x0;
            k1 = h * f(x0, y0);
            k2 = h * f(x0 + 0.5 * h, y0 + 0.5 * k1);
            k3 = h * f(x0 + 0.5 * h, y0 + 0.5 * k2);
            k4 = h * f(x0 + h, y0 + k3);
            ynew = y0 + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
            xnew = x0 + h;
            x0 = xnew;
            y0 = ynew;
        }while (x0 < x);
        result = ynew;
    }
    return result;
}
```

```
static void TestRungeKutta4()
{
  double h = 0.001; double x0 = 0.0; double y0 = 1.0;
  Console.WriteLine("\n Results from the 4th-order Runge-Kutta method
       with h={0}\n",h);
  double result = y0;
  for (int i = 0; i < 11; i++)
  {
    double x = 0.1 * i;
    result = ODE_RungeKutta4(f, x0, result, h, x);
    double exact = Math.Exp(Math.Sin(x));
    if (i % 5 == 0)
    Console.WriteLine(" x={0:n1},y={1:e12},exact={2:e12}",x,result,
        exact);
    x0 = x;
  }
}
```

```
Results from the 4th-order Runge-Kutta method with h = 0.001
x = 0.0, y = 1.000000000000e+000, exact = 1.000000000000e+000
x = 0.5, y = 1.615146296442e+000, exact = 1.615146296442e+000
x = 1.0, y = 2.319776824716e+000, exact = 2.319776824716e+000
```

16.3.3 Runge-Kutta-Fehlberg Method

The optimum Runge-Kutta method of a particular order is the one whose truncation error is minimum. In order to obtain more accurate and precise results, one could conceivably continue to calculate and extract increasingly higher order formulations of the Runge-Kutta method. The major drawback to this approach, however, is that the number of both equations and terms to keep track of dramatically increases making the process long and tedious which inevitably can also easily lead to calculation errors. In addition, keeping track of truncation errors can quickly become an even more daunting ordeal. Because of these issues, the Runge-Kutta-Fehlberg method incorporates an adaptive procedure where an estimate of the truncation error is calculated at each step and then the step size h is automatically adjusted to keep the calculation error within prescribed limits. At each step, the Runge-Kutta-Fehlberg method works by calculating and comparing two estimates of the solution. If the two results are in close agreement, the calculated approximation is accepted. However, if the results do not agree within the prescribed accuracy, the step size is reduced. If the step sizes agree by more significant digits than is required, the step size is increased.

The Runge-Kutta-Fehlberg method uses a Runge-Kutta method with a local truncation error of order five [69],

$$y_{n+1}^{(5)} = y_n + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6$$

to estimate the local error in a Runge-Kutta method of order four given by

$$y_{n+1}^{(4)} = y_n + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5$$

where the coefficient equations are given by [69]:

$$\begin{aligned}
k_1 &= f(x_n, y_n)\,h \\
k_2 &= f(x_n + \frac{1}{4}h, y_n + \frac{1}{4}k_1)\,h \\
k_3 &= f(x_n + \frac{3}{8}h, y_n + \frac{3}{32}k_1 + \frac{9}{32}k_2)\,h \\
k_4 &= f(x_n + \frac{12}{13}h, y_n + \frac{1932}{2197}k_1 - \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3)\,h \\
k_5 &= f(x_n + h, y_n + \frac{439}{216}k_1 - 8\,k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4)\,h \\
k_6 &= f(x_n + \frac{1}{2}h, y_n - \frac{8}{27}k_1 + 2\,k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5)\,h
\end{aligned}$$

This method has the advantage that only six evaluations of f are required per step. Arbitrary Runge-Kutta methods of orders four and five used together require at least four evaluations of f for the fourth order method and an additional six for the fifth order method, for a total of at least ten functional evaluations.

The optimal step size s is determined by simply multiplying the scaling factor s by the current step size h where the scaling factor s is given by [69]:

$$s = \left(\frac{tolerance}{2 \times error}\right)^{1/4}$$

where *tolerance* is the specified error control tolerance parameter and *error* is the truncation error inherent with this method given by [69]

$$error = \frac{1}{h}\left|y^{(5)}_{n+1} - y^{(4)}_{n+1}\right| = \frac{1}{h}\left|\frac{1}{360}k_1 - \frac{128}{4275}k_3 - \frac{2197}{75240}k_4 + \frac{1}{50}k_5 + \frac{2}{55}k_6\right|$$

The implementation of the Runge-Kutta-Fehlberg method in C# is given below.

```
public delegate double Function(double x, double y);

static double f(double x, double y)
{ return y*Math.Cos(x); }

public static double ODE_RungeKuttaFehlberg(Function f, double x0,
    double y0, double x, double h, double tolerance)
{
    double xnew, ynew, hnew, k1, k2, k3, k4, k5, k6;
    double hmin = 0.0001;
    double hmax = 0.5;
    if (h > hmax) h = hmax;
    if (h < hmin) h = hmin;
    while (x0 < x)
    {
        k1 = h * f(x0,y0);
        k2 = h * f(x0+0.25*h,y0+0.25*k1);
        k3 = h * f(x0+3*h/8,y0 + 3*k1/32 + 9*k2/32);
        k4 = h * f(x0+12*h/13,y0+1932*k1/2197 - 7200*k2/2197 + 7296*
            k3/2197);
        k5 = h * f(x0+h,y0+439*k1/216 - 8*k2 + 3680*k3/513 - 845*k4
            /4104);
        k6 = h * f(x0+0.5*h,y0 - 8*k1/27 + 2*k2 - 3544*k3/2565 +
            1859*k4/4104 - 11*k5/40);
        double error = Math.Abs(k1/360 - 128*k3/4275 - 2197*k4/75240
            + k5/50 + 2*k6/55)/h;
        double s = Math.Pow(0.5*tolerance/error,0.25);
        if (error < tolerance)
        {
            ynew = y0 + 25*k1/216 + 1408*k3/2565 + 2197*k4/4104 -
                0.2*k5;
            xnew = x0 + h;
            x0 = xnew;
            y0 = ynew;
        }
        if (s < 0.1) s = 0.1;
        if (s > 4)   s = 4;
        hnew = h*s;
        h = hnew;
        if (h > hmax) h = hmax;
        if (h < hmin) h = hmin;
        if (h > x - x0) h = x - x0;
    } return y0;
}

static void TestRungeKuttaFehlberg()
{
    double h = 0.2;
    double x0 = 0.0;
    double y0 = 1.0;
    Console.WriteLine("\n Results from the 4th-order Runge-Kutta-
        Fehlberg method with h = {0}\n", h);
    double result = y0;
    for (int i = 0; i < 11; i++)
    {
        double x = 0.1 * i;
        result = ODE_RungeKuttaFehlberg(f, x0, result, x, h, 1e-8);
        double exact = Math.Exp(Math.Sin(x));
        if (i%5==0)
        Console.WriteLine(" x = {0:n1}, y = {1:e12}, exact = {2:e12}"
            , x, result, exact);
        x0 = x;
    }
}
```

```
Results from the 4th-order Runge-Kutta-Fehlberg method with h = 0.2
x = 0.0, y = 1.000000000000e+000, exact = 1.000000000000e+000
x = 0.5, y = 1.615146299180e+000, exact = 1.615146296442e+000
x = 1.0, y = 2.319776827547e+000, exact = 2.319776824716e+000
```

16.4 Coupled Differential Equations

The methods for solving differential equations discussed so far apply only to single first-order differential equations. However, some physical phenomena can only be modeled by using either higher order or coupled differential equations. Fortunately, higher order differential equations can always be transformed into a coupled system of first-order equations by using the clever trick of expanding higher-order derivatives into a series of first-order equations. For example, a spring-mass system can be modeled to include damping by the following second-order differential equation:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2}+\frac{b}{m}\frac{\mathrm{d}x}{\mathrm{d}t}+\omega^2x=0$$

where k is the spring constant, ω is the oscillating frequency and b is the damping coefficient. Since the velocity $v=\frac{\mathrm{d}x}{\mathrm{d}t}$, the equation of motion above can be re-written in terms of two coupled first-order differential equations as follows:

$$\frac{\mathrm{d}v}{\mathrm{d}t}=-\frac{b}{m}v-\omega^2x$$
$$\frac{\mathrm{d}x}{\mathrm{d}t}=v$$

Since higher order differential equations can be simplified down to coupled first-order equations this way, it would be prudent to develop some kind of systematic numerical method for solving coupled first-order differential equations. There are many approaches to solving this kind of problem. One possibility, for example, is to use vector data structures, as described in Chapter 3, to hold and manipulate the contents of the coupled equations. However, that kind of approach can sometimes obscure much of the beautiful mathematics that is present behind the scenes and in the end produce such a compact solution that can often be difficult to follow or understand. Instead, I thought it would be more pedagogically useful to work through an example from scratch so that users can perhaps really understand and appreciate the entire solution process so that in the end, the method can be easily expanded to include the solution of more difficult examples.

Instead of using the well known equations of motion for a damped spring-mass system as given above, whose solutions can be easily found in many physics textbooks, let us explore the equations of motion of something more exotic like a Lorenz oscillator. These equations were first introduced by Edward Lorenz in 1963 who derived them from the simplified equations of convection rolls arising in the equations describing the atmosphere and so these equations also have important implications for climate and weather prediction. The Lorenz oscillator is a 3-dimensional dynamical system that exhibits chaotic flow and is noted for its distinctive characteristic lemniscate shape. The state of such a chaotic dynamical system has been found to evolve over time in a complex, non-repeating pattern. The equations of motion of

Lorenz oscillators are given by [85]:

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

where σ is called the Prandtl number and ρ is called the Rayleigh number. All σ, ρ and β are > 0, and usually $\sigma = 10$, $\beta = 8/3$ and ρ is varied. The system exhibits chaotic behavior for $\rho = 28$ but displays knotted periodic orbits for other values of ρ. Using these values for the constants σ, ρ and β we can write our differential equations in C# as follows:

```
static double dx(double x, double y, double z)
{
  return 10.0 * (y - x);
}

static double dy(double x, double y, double z)
{
  return x * (28.0 - z) - y;
}

static double dz(double x, double y, double z)
{
  return x * y - 8.0 * z / 3.0;
}
```

Next, we are faced with the decision of choosing which numerical method to use for solving the Lorenz first-order differential equations given above. Although the 4th order Runge-Kutta method reportedly gives more accurate results, for the purposes of this example and to keep everything as simple as possible, the Euler method is good enough and is displayed below. The resulting output follows. For illustrative purposes and in order to keep the output size within reasonable limits, I am only displaying part of the actual output. The rest can be seen by running the appropriate sample program that comes with this book.

```
static void TestEulerCoupledODE()
{
  double x = 0.0, y = 5.0, z = 10.0;  //initial conditions
  double dt = 0.001;                  //step size
  int maxnSteps = 100;          //max number of steps

  for (int i = 0; i < maxnSteps; i++)
  {
    double xnew = x + dx(x, y, z) * dt;
    double ynew = y + dy(x, y, z) * dt;
    double znew = z + dz(x, y, z) * dt;
    x = xnew;
    y = ynew;
    z = znew;
    Console.WriteLine("x={0:n1},y={1:e12},z={2:e12}",x,y,z);
  }
}

Results from a Lorenz coupled ODE using Euler's method

 x = 0.1, y = 4.995000000000e+000, z = 9.973333333333e+000
 x = 0.1, y = 4.990906333333e+000, z = 9.946987527778e+000
 x = 0.1, y = 4.987710799090e+000, z = 9.920958573339e+000
 x = 0.2, y = 4.985405377378e+000, z = 9.895242683344e+000
 x = 0.2, y = 4.983982228306e+000, z = 9.869836288041e+000
 x = 0.3, y = 4.983433690899e+000, z = 9.844736028443e+000
 x = 0.3, y = 4.983752281946e+000, z = 9.819938750427e+000
 x = 0.4, y = 4.984930694779e+000, z = 9.795441499071e+000
 x = 0.4, y = 4.986961798001e+000, z = 9.771241513228e+000
 x = 0.5, y = 4.989838634163e+000, z = 9.747336220314e+000
 x = 0.5, y = 4.993554418379e+000, z = 9.723723231326e+000
 x = 0.6, y = 4.998102536914e+000, z = 9.700400336064e+000
 x = 0.6, y = 5.003476545714e+000, z = 9.677365498561e+000
 x = 0.7, y = 5.009670168909e+000, z = 9.654616852711e+000
 x = 0.7, y = 5.016677297277e+000, z = 9.632152698087e+000
 x = 0.7, y = 5.024491986673e+000, z = 9.609971495948e+000
 x = 0.8, y = 5.033108456436e+000, z = 9.588071865420e+000
 x = 0.8, y = 5.042521087762e+000, z = 9.566452579865e+000
 x = 0.9, y = 5.052724422059e+000, z = 9.545112563400e+000
 x = 0.9, y = 5.063713159272e+000, z = 9.524050887597e+000
 x = 1.0, y = 5.075482156201e+000, z = 9.503266768333e+000
 x = 1.0, y = 5.088026424784e+000, z = 9.482759562794e+000
 x = 1.0, y = 5.101341130377e+000, z = 9.462528766633e+000
 x = 1.1, y = 5.115421590013e+000, z = 9.442574011265e+000
 x = 1.1, y = 5.130263270651e+000, z = 9.422895061309e+000
 x = 1.2, y = 5.145861787406e+000, z = 9.403491812163e+000
 x = 1.2, y = 5.162212901779e+000, z = 9.384364287709e+000
 x = 1.2, y = 5.179312519867e+000, z = 9.365512638150e+000
 x = 1.3, y = 5.197156690569e+000, z = 9.346937137963e+000
 x = 1.3, y = 5.215741603785e+000, z = 9.328638183980e+000
 x = 1.4, y = 5.235063588605e+000, z = 9.310616293585e+000
 x = 1.4, y = 5.255119111492e+000, z = 9.292872103013e+000
 x = 1.4, y = 5.275904774463e+000, z = 9.275406365777e+000
 x = 1.5, y = 5.297417313257e+000, z = 9.258219951182e+000
 x = 1.5, y = 5.319653595506e+000, z = 9.241313842957e+000
 x = 1.5, y = 5.342610618895e+000, z = 9.224689137981e+000
 x = 1.6, y = 5.366285509324e+000, z = 9.208347045105e+000
 x = 1.6, y = 5.390675519058e+000, z = 9.192288884071e+000
 x = 1.7, y = 5.415778024881e+000, z = 9.176516084523e+000
 x = 1.7, y = 5.441590526237e+000, z = 9.161030185108e+000
 x = 1.7, y = 5.468110643379e+000, z = 9.145832832661e+000
 x = 1.8, y = 5.495336115502e+000, z = 9.130925781472e+000
 x = 1.8, y = 5.523264798876e+000, z = 9.116310892648e+000
 x = 1.8, y = 5.551894664981e+000, z = 9.101990133536e+000
 x = 1.9, y = 5.581223798625e+000, z = 9.087965577239e+000
 x = 1.9, y = 5.611250396068e+000, z = 9.074239402197e+000
 x = 2.0, y = 5.641972763137e+000, z = 9.060813891852e+000
 x = 2.0, y = 5.673389313330e+000, z = 9.047691434372e+000
 x = 2.0, y = 5.705498565926e+000, z = 9.034874522456e+000
 ...           ...                     ...
```

17

Partial Differential Equations

17.1 Introduction

Partial differential equations, often abbreviated by the acronym PDE, are equations consisting of two or more independent variables and any partial derivatives with respect to those variables. The order of a partial differential equation is the order of the highest derivative involved. Many physical phenomena can be modeled using these types of equations and so PDEs play a very important role in many scientific and engineering applications. Although many PDEs can be solved analytically, most are often so complex that their solutions can only be approximated numerically. However, the numerical treatment of partial differential equations is, by itself, a vast subject easily meriting an entire book of its own. As a result, the intent of this chapter is to provide readers with the briefest possible but hopefully useful introduction and also illustrate how these elementary concepts may be coded in C#. Before doing that, however, let us briefly review some fundamental PDE concepts.

A first-order partial differential equation with n-independent variables has the general form

$$F\left(x_1, x_2, \ldots, x_n, w, \frac{\partial w}{\partial x_1}, \frac{\partial w}{\partial x_2}, \ldots, \frac{\partial w}{\partial x_n}\right) = 0$$

where $w = w(x_1, x_2, \ldots, x_n)$ is the unknown function and $F(\ldots)$ is a given function.

A second-order nonlinear partial differential equation with two independent variables x and y has the general form

$$F\left(x, y, w, \frac{\partial w}{\partial x}, \frac{\partial w}{\partial y}, \frac{\partial^2 w}{\partial x^2}, \frac{\partial^2 w}{\partial x \partial y}, \frac{\partial^2 w}{\partial y^2}\right) = 0$$

where $w = w(x,y)$ is the unknown function and $F(\ldots)$ is a given function.

A second-order semi-linear partial differential equation with two independent variables x and y has the form

$$a(x,y)\frac{\partial^2 w}{\partial x^2} + 2b(x,y)\frac{\partial^2 w}{\partial x \partial y} + c(x,y)\frac{\partial^2 w}{\partial y^2} = F\left(x, y, w, \frac{\partial w}{\partial x}, \frac{\partial w}{\partial x}\right)$$

Given a point (x,y), the semi-linear partial differential equation above is said to be

$$\begin{array}{ll} \text{parabolic} & \text{if } b^2 - ac = 0 \\ \text{hyperbolic} & \text{if } b^2 - ac > 0 \\ \text{elliptic} & \text{if } b^2 - ac < 0 \end{array}$$

at this point.

Due to the complexity of partial differential equations there are many different methods for solving them and a generalized list of the most popular methods include the following:

- The *finite difference method* is perhaps the easiest known technique for numerically solving PDEs and so it is often the first method chosen. The basic idea is to have functions be represented by their values at certain grid points and also have any partial derivatives be approximated through differences in these values. One disadvantage of this method is that it becomes quite complex when solving PDEs on irregular domains. In addition, it is not always easy to follow through and find solutions to the difference equations that result, evaluate their stability or establish their convergence especially for PDEs with variable coefficients or PDEs which are non-linear.

- The *finite element method* is arguably the most popular method for solving various PDEs. Compared to other methods, it has a well established mathematical theory for solving many PDEs even over complex domains. The solution approach is based either on eliminating the differential equation completely, as in the case of steady state problems, or rendering the PDE into an approximating system of ordinary differential equations, which are then numerically integrated using standard techniques such as Euler's method or Runge-Kutta.

- The *boundary element method* is used to solve those PDEs which can be formulated as integral equations. An integral equation is an equation in which an unknown function appears under an integral sign. The boundary element method attempts to use the given boundary conditions to fit boundary values into the integral equation, rather than values throughout the space defined by the PDE. Conceptually, the boundary element method can be thought of as a finite element method over the modeled surface instead of over the modeled physical domain. Hence the boundary element method is often more efficient than other methods in terms of computational resources for problems when the surface-to-volume ratio is small. However, boundary element formulations typically yield fully populated matrices, which can increase both storage requirements and subsequent computation time. In addition, not many problems, such as non-linear problems, can be can be written as integral equations and this limits the applicability of the boundary element method to solve PDEs.

- The *finite volume method* is another technique for solving PDEs by representing and evaluating partial differential equations as algebraic equations. Similar to the finite difference method, values are calculated at discrete places on a meshed geometry. The finite volume refers to the small volume surrounding each node point on a mesh. In the finite volume method, volume integrals in a partial differential equation that contain a divergence term are converted to surface integrals, using the divergence theorem. These terms are then evaluated as fluxes at the surfaces of each finite volume.

- *Spectral methods* take advantage of the underlying periodicity properties of some PDEs by writing the solution as a Fourier series and then substituting this series back into the PDE to get a system of ordinary differential equations in the time-dependent coefficients of the trigonometric terms in the series. Then a time-stepping method is used to solve those ODEs.

- The *meshless* or *meshfree* method is a relatively recent technique that was developed for solving PDEs. This method gets rid of the tedious meshing and re-meshing of the entire modeled domain and directly uses the geometry of the simulated object for calculations. A goal of meshfree methods is to facilitate the simulation of increasingly demanding problems that require the ability to treat large deformations, advanced materials, complex geometry, nonlinear material behavior, discontinuities and singularities.

- The *method of lines* is a technique for solving partial differential equations where all but one variable is discretized. The resulting semi-discrete problem is a set of ordinary differential equations or differential algebraic equations that is then integrated.

- *Domain decomposition* methods solve a boundary value problems by splitting them into smaller boundary value problems on subdomains and iterating to coordinate the solution between the subdomains. The problems on the subdomains are independent, which makes domain decomposition methods suitable for parallel computing.

- *Multigrid methods* in numerical analysis are a group of algorithms for solving differential equations using a hierarchy of discretizations. The idea is similar to extrapolation between coarser and finer grids. The typical application for multigrid is in the numerical solution of elliptic partial differential equations in two or more dimensions.

Because of the huge volume of material comprising each of these techniques, we will further narrow our objectives and instead limit our attention to arguably the most popular numerical method of them all, the finite difference method, and see how that method can be applied to solving parabolic, hyperbolic, and elliptic partial differential equations. Unlike most other numerical methods which eventually can be summarized and sometimes even be generalized into nice compact cohesive units, numerical solutions of partial differential equations using the finite differences method tend to be a rather lengthy and messy process to code thereby making it necessary to customize the implementation of each method to fit a particular problem.

17.2 The Finite Difference Method

The *finite difference method* is a procedure used to obtain approximate numerical solutions of a partial differential equation by discretizing the continuous physical domain (x,y) into a discrete finite difference grid (x_i,y_i) where the xy plane is partitioned into equally spaced grid lines parallel to the coordinate axes and defined by step sizes h and k given by

$$\Delta x = h = \frac{x_n - x_0}{n} \qquad \text{and} \qquad \Delta y = k = \frac{y_m - y_0}{m}$$

where n and m are integers. This means that an arbitrary point (x_i,y_i) can be specified by

$$\begin{aligned} x_i &= x_0 + ih \quad \text{for} \quad i{=}0,1,2,\ldots,n-1 \\ y_j &= y_0 + jk \quad \text{for} \quad j{=}0,1,2,\ldots,m-1 \end{aligned}$$

The lines $x = x_i$ and $y = y_i$ are called grid lines and their intersections are called the mesh points of the grid. To simplify notation, the value of some arbitrary function $f(x,y)$ at some mesh point (x_i,y_j) is sometimes denoted by $f(x_i,y_j) = f(ih,jk) = f_{ij}$. The exact individual partial derivatives in the PDE are then approximated by algebraic finite difference approximations where for each mesh point in the interior of the grid (x_i,y_j), a Taylor series expansion is used in the variable x about x_i to generate the first and second order derivatives. Generally, there are three different ways to do this: A *forward finite difference* is an expression of the form $f(x+h) - f(x)$ where, depending on the application, the spacing h may be variable or held constant. Similarly, a *backward difference* is given by $f(x) - f(x-h)$. Lastly, a *central difference* is given by $f(x+\frac{1}{2}h) - f(x-\frac{1}{2}h)$. Then the finite difference approximations are substituted back into the partial differential equation to obtain an algebraic finite difference equation which can then be solved for the dependent variable. The equations below summarize the results of doing a Taylor series approximation on the first and second order derivatives for each case [69].

$$\begin{aligned} \left(\frac{\partial u}{\partial x}\right)_{i,j} &\approx \frac{u_{i+1,j} - u_{i,j}}{h} + O(h) \qquad \text{(forward difference approximation)} \\ &\approx \frac{u_{i,j} - u_{i-1,j}}{h} + O(h) \quad \text{(backward difference approximation)} \\ &\approx \frac{u_{i+1,j} - u_{i-1,j}}{2h} + O(h^2) \qquad \text{(central difference approximation)} \end{aligned}$$

$$
\begin{aligned}
\left(\frac{\partial u}{\partial y}\right)_{i,j} &\approx \frac{u_{i,j+1}-u_{i,j}}{k} + O(k) \qquad \text{(forward difference approximation)}\\
&\approx \frac{u_{i,j}-u_{i,j-1}}{k} + O(k) \quad \text{(backward difference approximation)}\\
&\approx \frac{u_{i,j+1}-u_{i,j-1}}{2k} + O(k^2) \qquad \text{(central difference approximation)}
\end{aligned}
$$

and

$$
\begin{aligned}
\left(\frac{\partial^2 u}{\partial x^2}\right)_{i,j} &\approx \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{h^2} + O(h^2) \quad \text{(central difference approximation)}\\
\left(\frac{\partial^2 u}{\partial y^2}\right)_{i,j} &\approx \frac{u_{i,j+1}-2u_{i,j}+u_{i,j-1}}{k^2} + O(k^2) \quad \text{(central difference approximation)}
\end{aligned}
$$

17.3 Parabolic Partial Differential Equations

The heat equation is an important second-order linear partial differential equation that describes the distribution of heat or variation in temperature in a given region over time. It is also an excellent example of a parabolic partial differential equation and arises from the theoretical analysis of empirical data obtained from the study of a natural physical phenomena. The heat equation is characterized by a time variable t and by the function $u(x,y,z,t)$ consisting of three spatial variables (x,y,z). In three-dimensional Cartesian coordinates, the heat equation takes the form

$$\frac{\partial u}{\partial t} - \alpha^2\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) = 0 \quad \text{or equivalently,} \quad \frac{\partial u}{\partial t} - \alpha^2\nabla^2 u = 0$$

where ∇^2 is called the Laplacian and α^2 is a constant. For pedagogical purposes, however, we can start by studying the solution methods to a simpler one dimensional equation and later expand the procedure to include more dimensions as needed. For example, a one dimensional heat equation is adequate for modeling how heat propagates over time down a steel rod and can thus be useful in a number of practical engineering applications.

In general, parabolic partial differential equations have the form

$$\frac{\partial u}{\partial t}(x,t) - \alpha^2\frac{\partial^2 u}{\partial x^2}(x,t) = 0 \qquad \text{where} \quad 0 < x < L, \quad t > 0$$

and are subject to some boundary condition like $u(0,t) = u(L,t) = 0$, where $t > 0$ and some initial conditions like $u(x,0) = f(x)$, where $0 \le x \le L$. This equation and its method of solution can be easily extended to include more dimensions as needed.

The first step in the process of numerically solving parabolic partial differential equations using the finite difference method is to create a mesh of grid points along the (x,t) plane. First, select an integer $m > 0$ and define a spatial step size $h = L/m$. Then select a time step size k. The grid points are then given by (x_i, t_j), where $x_i = ih$ for $i = 0, 1, 2, \ldots, m$ and $t_j = jk$ for $j = 0, 1, 2, \ldots$.

The parabolic partial differential equation given above implies that at the interior grid point (x_i, t_j) for each $i = 1, 2, \ldots, m-1$ and $j = 1, 2, \ldots$ we have

$$\frac{\partial u}{\partial t}(x_i, t_j) - \alpha^2 \frac{\partial^2 u}{\partial x^2}(x_i, t_j) = 0$$

At this point we may be tempted to continue by selecting any one of the available three finite difference methods: forward, backward or central. However, Burden and Faires [69] point out that making a careless selection of which finite difference method to use can sometimes lead to undesired unstable or conditionally stable solutions. A better unconditionally stable solution requires a more thorough analysis of all three finite difference methods and, as such, it is beyond the scope of this book. Issacson and Keller [86], for example, provide a more complete theoretical treatment of stability issues in the solution of partial differential equations. As it turns out, using the backward difference method will result in a much more desired implicit solution that is unconditionally stable.

Using the backward difference approximation along with the results for approximating the first and second partial derivative by using a Taylor series expansion as given in section 17.2, we can then write the above general parabolic differential equation as

$$\frac{w_{i,j} - w_{i,j-1}}{k} - \alpha^2 \frac{w_{i+1,j} - 2w_{i,j} + w_{i-1,j}}{h^2} = 0$$

for each $i = 1, 2, \ldots, m-1$ and $j = 1, 2, \ldots$ and where $w_{i,j}$ approximates $u(x_i, t_j)$.

By letting $\lambda = \alpha^2 \frac{k}{h^2}$, we can re-arrange the previous equation for $w_{i,j}$ in the following format:

$$(1 + 2\lambda) w_{i,j} - \lambda w_{i+1,j} - \lambda w_{i-1,j} = w_{i,j-1}$$

for each $i = 1, 2, \ldots, m-1$ and $j = 1, 2, \ldots$. Using the knowledge from the initial conditions that $w_{i,0} = f(x_i)$ for each $i = 1, 2, \ldots, m-1$ and $w_{m,j} = w_{0,j} = 0$ for each $j = 1, 2, \ldots$ this backward difference method leads to the following matrix representation: $Aw^{(j)} = w^{(j-1)}$ for each $i = 1, 2, \ldots$ or more explicitly as:

$$\begin{pmatrix} (1+2\lambda) & -\lambda & 0 & \ldots & 0 & 0 \\ -\lambda & (1+2\lambda) & -\lambda & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \ddots & 0 & 0 \\ 0 & \ldots & -\lambda & (1+2\lambda) & -\lambda & 0 \\ 0 & 0 & \ldots & -\lambda & (1+2\lambda) & -\lambda \\ 0 & 0 & 0 & \ldots & -\lambda & (1+2\lambda) \end{pmatrix} \begin{pmatrix} w_{1,j} \\ w_{2,j} \\ \vdots \\ w_{m-1,j} \end{pmatrix} = \begin{pmatrix} w_{1,j-1} \\ w_{2,j-1} \\ \vdots \\ w_{m-1,j-1} \end{pmatrix}$$

Therefore, we just need to solve a system of linear equations to obtain $w^{(j)}$ from $w^{(j-1)}$. Since $\lambda > 0$, in addition to being tridiagonal, the matrix A is also positive definite and strictly diagonally dominant.

The implementation in C# of the backward difference method for solving the heat equation that follows seeks to approximate the solution to the standard parabolic partial differential equation of the form

$$\frac{\partial u}{\partial t}(x,t) - \alpha^2 \frac{\partial^2 u}{\partial x^2}(x,t) = 0$$

where $0 < x < 1$, $0 < t < T$ and subject to the boundary conditions $u(0,t) = u(1,t) = 0$ and the initial conditions $u(x,0) = f(x) = \sin(\pi x)$. Note that for stopping purposes, a bound T is given for the variable t. The results for a system where $h = 0.1, k = 0.01$, $0 < x < 1$ is given below. For convenience and for comparison with the calculated output values, the actual solution is given by $u(x,t) = e^{-\pi^2 t}\sin(\pi x)$.

```
static double functionX(double x)
{ return Math.Sin(Math.PI * x); }

static void Solving_Heat_PDE()
{
  /*
   *  Solution to the Heat PDE by Backward-Difference Algorithm
   *
   *  Goal: To approximate the solution to the parabolic
   *  PDE subject to the boundary conditions:
   *  u(0,t) = u(l,t) = 0, 0 < t < T = max t,
   *  and the initial conditions u(x,0) = F(x), 0 <= x <= l:
   *  Input: endpoint l; max time T; constant alpha; integers m,N.
   *  Output: approximations W(I,J) to u(x(I),t(J)) for each
   *  I = 1, ..., m-1 and J = 1, ..., N.
   */

   double[] W, L, U, Z, ActualSolution;
   double FT, FX, alpha, H, K, lambda, T, X;
   int N, M, M1, M2, N1, I1, I, J;
   W = new double[25]; L = new double[25];
   U = new double[25]; Z = new double[25];
   ActualSolution = new double[25];
   FX = 1;      //These input parameters values
   FT = 0.5;    //are set by the user and may
   alpha = 1;   //be varied accordingly
   M = 10;
   N = 50;

   //Initialize variables with input parameter values
   M1 = M - 1;
   M2 = M - 2;
   N1 = N - 1;
   H = FX / M;
   K = FT / N;
   lambda = alpha * alpha * K / (H * H);
   for (I = 1; I <= M1; I++) W[I - 1] = functionX(I * H);
   /* Solve the tridiagonal linear system */
   L[0] = 1.0 + 2.0 * lambda;
   U[0] = -lambda / L[0];
   for (I = 2; I <= M2; I++)
   {
     L[I - 1] = 1.0 + 2.0 * lambda + lambda * U[I - 2];
     U[I - 1] = -lambda / L[I - 1];
   }
   L[M1 - 1] = 1.0 + 2.0 * lambda + lambda * U[M2 - 1];
   for (J = 1; J <= N; J++)
   {
     T = J * K;
     Z[0] = W[0] / L[0];
     for (I = 2; I <= M1; I++)
       Z[I - 1] = (W[I - 1] + lambda * Z[I - 2]) / L[I - 1];
     W[M1 - 1] = Z[M1 - 1];
     for (I1 = 1; I1 <= M2; I1++)
     {
       I = M2 - I1 + 1;
       W[I - 1] = Z[I - 1] - U[I - 1] * W[I];
     }
   }
   Console.WriteLine("Solution of a Parabolic (Heat) Partial
       Differential Equation");
   Console.WriteLine("using the backward-difference algorithm and
       user defined parameters\n");

   Console.WriteLine("I\tX(I)\t\tW(X(I)," + FT + ")
        \tActual Solution\n");
   for (I = 1; I <= M1; I++)
   {
     X = I * H;
     ActualSolution[I] = Math.Exp(-Math.PI * Math.PI * 0.5)*
        Math.Sin(Math.PI * X);
     Console.WriteLine(I.ToString() + "\t" +
        X.ToString("0.000000000") + "\t" +
        W[I - 1].ToString("0.000000000") +
        "\t" + ActualSolution[I].ToString("0.000000000"));
   }
   Console.ReadLine();
}
```

```
OUTPUT: Solution of a Parabolic (Heat) Partial Differential Equation
using the backward-difference algorithm and user defined parameters

I       X(I)            W(X(I),0.5)     Actual Solution

1       0.100000000     0.002898017     0.002222414
2       0.200000000     0.005512355     0.004227283
3       0.300000000     0.007587106     0.005818356
4       0.400000000     0.008919178     0.006839888
5       0.500000000     0.009378179     0.007191883
6       0.600000000     0.008919178     0.006839888
7       0.700000000     0.007587106     0.005818356
8       0.800000000     0.005512355     0.004227283
9       0.900000000     0.002898017     0.002222414
```

17.3.1 The Crank-Nicolson Method

The Crank-Nicolson method is another numerically stable finite difference method used for numerically solving parabolic partial differential equations. The Crank-Nicolson method is based on central difference in space, and the trapezoidal rule in time, giving second-order convergence in time. Equivalently, it is the average of the forward and backward Euler methods in time. Using the general form of the parabolic differential equation then the resulting difference equation can be expressed by [69]:

$$\frac{w_{i,j+1}-w_{i,j}}{k}-\frac{\alpha^2}{2}\left[\frac{w_{i+1,j}+2w_{i,j}+w_{i-1,j}}{h^2}-\frac{w_{i+1,j+1}-2w_{i,j+1}+w_{i-1,j+1}}{h^2}\right]=0$$

for each $i=1,2,\ldots,m-1$ and $j=0,1,2,\ldots$ and where $w_{i,j}$ approximates $u(x_i,t_j)$.

This resulting finite difference equation can also be expressed in matrix form as $Aw^{(j+1)}=Bw^{(j)}$ for each $j=0,1,2,\ldots$ where $\lambda=\alpha^2\frac{k}{h^2}$ as before and

$$A=\begin{pmatrix}(1+\lambda) & -\frac{\lambda}{2} & 0 & \ldots & 0 & 0\\ -\frac{\lambda}{2} & (1+\lambda) & -\frac{\lambda}{2} & 0 & \ldots & 0\\ \vdots & \ddots & \ddots & \ddots & 0 & 0\\ 0 & \ldots & -\frac{\lambda}{2} & (1+\lambda) & -\frac{\lambda}{2} & 0\\ 0 & 0 & \ldots & -\frac{\lambda}{2} & (1+\lambda) & -\frac{\lambda}{2}\\ 0 & 0 & 0 & \ldots & -\frac{\lambda}{2} & (1+\lambda)\end{pmatrix}$$

and

$$B=\begin{pmatrix}(1-\lambda) & \frac{\lambda}{2} & 0 & \ldots & 0 & 0\\ \frac{\lambda}{2} & (1-\lambda) & \frac{\lambda}{2} & 0 & \ldots & 0\\ \vdots & \ddots & \ddots & \ddots & 0 & 0\\ 0 & \ldots & \frac{\lambda}{2} & (1-\lambda) & \frac{\lambda}{2} & 0\\ 0 & 0 & \ldots & \frac{\lambda}{2} & (1-\lambda) & \frac{\lambda}{2}\\ 0 & 0 & 0 & \ldots & \frac{\lambda}{2} & (1-\lambda)\end{pmatrix}$$

From these results we see that the matrix A is once again positive definite, strictly diagonally dominant and nonsingular. As a result, we can easily calculate $w^{(j+1)}$ from $w^{(j)}$ for each $j=0,1,2,\ldots$. The implementation in C# of the Crank-Nicolson algorithm below solves the exact same parabolic partial differential equation as in the last example for the heat equation. This was purposely done in order to compare the output results from both methods to see how close they are to each other. A proof that the Crank-Nicolson method is unconditionally stable can be found in Issacson and Keller [86]. For convenience and for comparison with the calculated output values, the actual solution is given by $u(x,t) = e^{-\pi^2 t}\sin(\pi x)$.

```
/*
 *   Solution to the Heat PDE by Crank-Nicolson Algorithm
 *
 *   Goal: To approximate the solution of the parabolic
 *   PDE subject to the boundary conditions
 *   u(0,t) = u(1,t) = 0, 0 < t < T = max t
 *   and the initial conditions u(x,0) = F(x), 0 <= x <= 1:
 *   Input: endpoint 1; maximum time T; constant alpha; integers m,N:
 *   Output: approximations W(I,J) to u(x(I),t(J)) for each
 *   I = 1,..., m-1 and J = 1,..., N.
 */

static void Solving_Heat_PDE_by_CrankNicolson_Algorithm()
{
    double[] V, L, U, ActualSolution, Z;
    double FT, FX, alpha, H, K, lambda, T, X;
    int N, M, M1, M2, I1, I, J;

    V = new double[25]; L = new double[25];
    U = new double[25]; Z = new double[25];
    ActualSolution = new double[25];

    FX = 1;      //These input parameters values
    FT = 0.5;    //are set by the user and may
    alpha = 1;   //be varied accordingly
    M = 10;
    N = 50;

    //Initialize variables with input parameter values
    M1 = M - 1;
    M2 = M - 2;
    H = FX / M;
    K = FT / N;
    lambda = alpha * alpha * K / (H * H);
    V[M - 1] = 0.0;
    for (I = 1; I <= M1; I++) V[I - 1] = functionX(I * H);

    /* Solve the tridiagonal linear system */
    L[0] = 1.0 + lambda;
    U[0] = -lambda / (2.0 * L[0]);
    for (I = 2; I <= M2; I++)
    {
        L[I - 1] = 1.0 + lambda + lambda * U[I - 2] / 2.0;
        U[I - 1] = -lambda / (2.0 * L[I - 1]);
    }
    L[M1 - 1] = 1.0 + lambda + 0.5 * lambda * U[M2 - 1];
    for (J = 1; J <= N; J++)
    {
        T = J * K;
        Z[0] = ((1.0 - lambda) * V[0] + lambda * V[1] / 2.0) / L[0];
        for (I = 2; I <= M1; I++)
            Z[I - 1] = ((1.0 - lambda) * V[I - 1] + 0.5 * lambda *
                (V[I] + V[I - 2] + Z[I - 2])) / L[I - 1];
        V[M1 - 1] = Z[M1 - 1];
        for (I1 = 1; I1 <= M2; I1++)
        {
            I = M2 - I1 + 1;
            V[I - 1] = Z[I - 1] - U[I - 1] * V[I];
        }
    }
    Console.WriteLine("Solution of a Parabolic (Heat) Partial
        Differential Equation");
    Console.WriteLine("using the Crank-Nicolson Method and user
        defined parameters\n");
    Console.WriteLine("I\tX(I)\t\tW(X(I),"
                + FT + ")\tActual Solution\n");
    for (I = 1; I <= M1; I++)
    {
        X = I * H;
        ActualSolution[I] = Math.Exp(-Math.PI * Math.PI * 0.5) *
                Math.Sin(Math.PI * X);
        Console.WriteLine(I.ToString() + "\t" +
                X.ToString("0.000000000") + "\t" +
                V[I - 1].ToString("0.000000000") + "\t" +
                ActualSolution[I].ToString("0.000000000"));
    }
    Console.ReadLine();
```

```
OUTPUT: Solution of a Parabolic (Heat) Partial Differential Equation
using the Crank-Nicolson Method and user defined parameters

I       X(I)            W(X(I),0.5)     Actual Solution

1       0.100000000     0.002305123     0.002222414
2       0.200000000     0.004384605     0.004227283
3       0.300000000     0.006034891     0.005818356
4       0.400000000     0.007094440     0.006839888
5       0.500000000     0.007459536     0.007191883
6       0.600000000     0.007094440     0.006839888
7       0.700000000     0.006034891     0.005818356
8       0.800000000     0.004384605     0.004227283
9       0.900000000     0.002305123     0.002222414
```

17.4 Hyperbolic Partial Differential Equations

The wave equation is an important second-order linear partial differential equation that can be used to describe the propagation of a variety of waves, such as sound waves, light waves and water waves over time. It is also an excellent example of a hyperbolic partial differential equation and arises from the theoretical analysis of empirical data obtained from studying wave phenomena in diverse disciplines such as acoustics, electromagnetics, and fluid dynamics. The wave equation is characterized by the function $u(x,y,z,t)$ consisting of three spatial variables (x,y,z) and a time variable t. In three-dimensional Cartesian coordinates, the wave equation takes the form

$$\frac{\partial^2 u}{\partial t^2} - \alpha^2\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) = 0 \quad \text{or equivalently,} \quad \frac{\partial^2 u}{\partial t^2} - c^2\nabla^2 u = 0$$

where ∇^2 is the Laplacian and α^2 is a fixed constant equal to the propagation speed of the wave. As before, we can start by studying the solution methods to a simpler one dimensional version of the equation and later expand the procedure to include more dimensions as needed.

In general, hyperbolic partial differential equations have the form

$$\frac{\partial^2 u}{\partial t^2}(x,t) - \alpha^2\frac{\partial^2 u}{\partial x^2}(x,t) = 0 \qquad \text{where} \quad 0 < x < L, \quad t > 0$$

and are subject to some boundary condition like $u(0,t) = u(L,t) = 0$, where $t > 0$ and some initial conditions like $u(x,0) = f(x)$, and $\frac{\partial u}{\partial t}(x,0) = g(x)$ for $0 \le x \le L$ where α is a constant. As before, we start by selecting an integer $m > 0$ and a time step size $k > 0$. With $h = L/m$, the mesh points (x_i,t_j) are defined by $x_i = ih$ for $i = 0,1,2,\ldots,m$ and $t_j = jk$ for $j = 0,1,2,\ldots$. At any interior mesh point (x_i,t_j), the wave equation becomes

$$\frac{\partial^2 u}{\partial t^2}(x_i,t_j) - \alpha^2\frac{\partial^2 u}{\partial x^2}(x_i,t_j) = 0$$

Applying the centered difference method along with the results for approximating the first and second partial derivatives using a Taylor series expansion as described in section 17.2, we can then write the general hyperbolic differential equation given above as

$$\frac{w_{i,j+1} - 2w_{i,j} + w_{i,j-1}}{k^2} - \alpha^2\frac{w_{i+1,j} - 2w_{i,j} + w_{i-1,j}}{h^2} = 0$$

for each $i = 1,2,\ldots,m-1$ and $j = 1,2,\ldots$ and where $w_{i,j}$ approximates $u(x_i,t_j)$. If $\lambda = \alpha k/h$, we can write the difference equation as

$$w_{i,j+1} - 2w_{i,j} + w_{i,j-1} - \lambda^2 w_{i+1,j} + 2\lambda^2 w_{i,j} - \lambda^2 w_{i-1,j} = 0$$

and solve for $w_{i,j+1}$, the most advanced time step approximation, to obtain

$$w_{i,j+1} = 2(1-\lambda^2)w_{i,j} + \lambda^2(w_{i+1,j} + w_{i-1,j}) - w_{i,j-1}$$

where $i = 1,2,\ldots,m-1$ and $j = 1,2,\ldots$. The boundary conditions give

$$w_{0,j} = w_{m,j} = 0 \qquad \text{for each} \qquad j = 1,2,\ldots$$

and the initial conditions implies

$$w_{i,0} = f(x_i) \qquad \text{for each} \qquad i = 1,2,\ldots,m-1$$

Expressing this set of equations in matrix form gives

$$\begin{pmatrix} w_{1,j+1} \\ w_{2,j+1} \\ \vdots \\ w_{m-1,j+1} \end{pmatrix} = \begin{pmatrix} 2(1-\lambda^2) & \lambda^2 & 0 & \ldots & 0 & 0 \\ \lambda^2 & 2(1-\lambda^2) & \lambda^2 & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \ddots & 0 & 0 \\ 0 & \ldots & \lambda^2 & 2(1-\lambda^2) & \lambda^2 & 0 \\ 0 & 0 & \ldots & \lambda^2 & 2(1-\lambda^2) & \lambda^2 \\ 0 & 0 & 0 & \ldots & \lambda^2 & 2(1-\lambda^2) \end{pmatrix} \begin{pmatrix} w_{1,j} \\ w_{2,j} \\ \vdots \\ w_{m-1,j} \end{pmatrix}$$

$$- \begin{pmatrix} w_{1,j-1} \\ w_{2,j-1} \\ \vdots \\ w_{m-1,j-1} \end{pmatrix}$$

The equation for $w_{i,j+1}$ and for the boundary conditions imply that the $(j+1)$st time step requires values from the jth and the $(j-1)$st time steps. This produces a minor starting problem because the values for $j=0$ are given by the initial condition $w_{i,0} = f(x_i)$ but values for $j=1$ which are needed to compute $w_{i,2}$ must be obtained from the other initial condition

$$\frac{\partial u}{\partial t}(x,0) = g(x) \qquad \text{for} \qquad 0 \le x \le L$$

One approach is to replace $\frac{\partial u}{\partial t}(x,0)$ by a forward difference approximation

$$\frac{\partial u}{\partial t}(x_i,0) \approx \frac{u(x_i,t_1) - u(x_i,0)}{k} + O(k)$$

Solving for $u(x_i,t_1)$ gives

$$u(x_i,t_1) \approx u(x_i,0) + k\frac{\partial u}{\partial t}(x_i,0) + O(k^2) = u(x_i,0) + kg(x_i) + O(k^2)$$

As a result, $w_{i,1} \approx w_{i,0} + kg(x_i)$ for each $i = 1,2,\ldots,m-1$ gives an approximation only up to an error in the order of $O(k)$. Seeking a better approximation to $u(x_i,0)$ consider expanding $u(x,t_1)$ in a Taylor series about $t=0$

$$u(x_i,t_1) \approx u(x_i,0) + k\frac{\partial u}{\partial t}(x_i,0) + \frac{k^2}{2}\frac{\partial^2 u}{\partial t^2}(x_i,0) + \frac{k^3}{6}\frac{\partial^3 u}{\partial t^3}(x_i,0) + \ldots$$

If f'' exists then

$$\frac{\partial^2 u}{\partial t^2}(x_i,0) = \alpha^2 \frac{\partial^2 u}{\partial x^2}(x_i,0) = \alpha^2 \frac{d^2 f}{dx^2}(x_i) = \alpha^2 f''(x_i)$$

So that

$$u(x_i,t_i) \approx u(x_i,0) + kg(x_i) + \frac{\alpha^2 k^2}{2} f''(x_i) + \frac{k^3}{6}\frac{\partial^3 u}{\partial t^3}(x_i,0) + \dots$$

producing an approximation with an error estimated to be in order of $O(k^3)$. That is,

$$u(x_i,t_i) \approx u(x_i,0) + kg(x_i) + \frac{\alpha^2 k^2}{2} f''(x_i) + O(k^3)$$

Using the finite central difference differentiation formulas from Chapter 11, we can write

$$f''(x_i) \approx \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2} + O(h^2)$$

we now obtain

$$u(x_i,t_i) \approx u(x_i,0) + kg(x_i) + \frac{\alpha^2 k^2}{2h^2}[f(x_{i+1}) - 2f(x_i) + f(x_{i-1})] + \dots$$

By letting $\lambda = (k\alpha/h)$ this last expression can be written as

$$u(x_i,t_i) \approx u(x_i,0) + kg(x_i) + \frac{\lambda^2}{2}[f(x_{i+1}) - 2f(x_i) + f(x_{i-1})] + \dots$$

$$u(x_i,t_i) \approx (1-\lambda^2)f(x_i) + \frac{\lambda^2}{2} f(x_{i+1}) + \frac{\lambda^2}{2} f(x_{i-1}) + kg(x_i) + \dots$$

Thus the difference equation

$$u_{i,1} \approx (1-\lambda^2)f(x_i) + \frac{\lambda^2}{2} f(x_{i+1}) + \frac{\lambda^2}{2} f(x_{i-1}) + kg(x_i) + \dots$$

can be used to find $w_{i,1}$ for each $i = 1,2,\dots,m-1$.

The implementation in C# of the numerical solution of the wave equation that follows seeks to approximate the solution to the standard hyperbolic partial differential equation

$$\frac{\partial^2 u}{\partial t^2}(x,t) - \alpha^2 \frac{\partial^2 u}{\partial x^2}(x,t) = 0$$

where $0 < x < 1$, $0 < t < T$ and subject to the boundary conditions $u(0,t) = u(1,t) = 0$ and the initial conditions $u(x,0) = f(x) = \sin(\pi x)$, where $0 \le x \le 1$ and $\frac{\partial u}{\partial t}(x,0) = 0$, where $0 \le x \le 1$. The finite difference algorithm used in this example with $m = 10, T = 1$, and $N = 20$, which implies that $h = 0.1, k = 0.05$, and $\lambda = 1$. The output of the results immediately follows the listing of the code and is for $w_{i,N}$ for $i = 0,1,2,\dots,10$. For convenience and for comparison with the calculated output values, the actual solution is given by $u(x,t) = \sin(\pi x)\cos(2\pi t)$.

```
/*
 *  Solution to the Wave PDE by finite difference method
 *  Goal: To approximate the solution of the hyperbolic
 *  PDE subject to the boundary conditions
 *  u(0,t) = u(l,t) = 0, 0 < t < T = max t
 *  and the initial conditions
 *  u(x,0) = F(x) and Du(x,0)/Dt = G(x), 0 <= x <= l:
 *  Input:  endpoint l; maximum time T; constant alpha; integers m,N.
 *  Output: approximations W(I,J) to u(x(I),t(J)) for each
 *  I = 0, ..., m and J=0,...,N.
 */

static double fx(double x)
{ return Math.Sin(Math.PI * x); }

static double gx(double x)
{ return 0.0; }

static void Solving_Wave_PDE_byBackward_Diff()
{
  double[,] W;
  double[] ActualSolution;
  double FT, FX, alpha, H, K, lambda, X;
  int M1, M2, N1, N2, I, J;
  W = new double[51, 51];
  ActualSolution = new double[51];

  FX = 1;      //These input parameters values
  FT = 1;      //are set by the user and may
  alpha = 2;  //be varied accordingly
  int M = 10;
  int N = 20;

  //Initialize variables with input parameter values
  M1 = M + 1;
  M2 = M - 1;
  N1 = N + 1;
  N2 = N - 1;
  H = FX / M;
  K = FT / N;
  lambda = alpha * K / H;
  for (J = 2; J <= N1; J++)
  {
    W[0, J - 1] = 0.0;
    W[M1 - 1, J - 1] = 0.0;
  }
  W[0, 0] = fx(0.0);
  W[M1 - 1, 0] = fx(FX);
  for (I = 2; I <= M; I++)
  {
    W[I - 1, 0] = fx(H * (I - 1.0));
    W[I - 1, 1] = (1.0 - lambda * lambda) * fx(H * (I - 1.0))
        + lambda * lambda * (fx(I * H)
        + fx(H * (I - 2.0))) / 2.0 + K * gx(H * (I - 1.0));
  }
  for (J = 2; J <= N; J++)
  {
      for (I = 2; I <= M; I++)
      {
        W[I - 1, J] = 2.0 * (1.0 - lambda * lambda) *
        W[I - 1, J - 1] + lambda * lambda *
        (W[I, J - 1] + W[I - 2, J - 1]) - W[I - 1, J - 2];
      }
  }
  Console.WriteLine("Solution of a Hyperbolic (Wave) Partial
                    Differential Equation");
  Console.WriteLine("using the Finite-Difference Method and
                    user defined parameters\n");
  Console.WriteLine("I\tX(I)\t\tW(X(I),"+FT+")\tActual Solution\n");
  for (I = 1; I <= M1; I++)
  {
    X = (I - 1) * H;
    ActualSolution[I] = Math.Sin(Math.PI * X) *
         Math.Cos(2.0 * Math.PI * 1.0);
    Console.WriteLine(I.ToString() + "\t" +
          X.ToString("0.000000000") + "\t" +
          W[I - 1, N1 - 1].ToString("0.000000000") +
          "\t" + ActualSolution[I].ToString("0.000000000"));
  }
  Console.ReadLine();
}
```

```
OUTPUT:

Solution of a Hyperbolic (Wave) Partial Differential Equation
using the Finite-Difference Method and user defined parameters

I       X(I)            W(X(I),1)       Actual Solution

1       0.000000000     0.000000000     0.000000000
2       0.100000000     0.309016994     0.309016994
3       0.200000000     0.587785252     0.587785252
4       0.300000000     0.809016994     0.809016994
5       0.400000000     0.951056516     0.951056516
6       0.500000000     1.000000000     1.000000000
7       0.600000000     0.951056516     0.951056516
8       0.700000000     0.809016994     0.809016994
9       0.800000000     0.587785252     0.587785252
10      0.900000000     0.309016994     0.309016994
11      1.000000000     0.000000000     0.000000000
```

17.5 Elliptic Partial Differential Equations

Poisson's equation is an important second-order partial differential equation with many practical applications in both science and engineering. It is also an excellent example of an elliptic partial differential equation and arises from the theoretical analysis of empirical data obtained from studying natural physical phenomena, particularly in the fields of electromagnetism, astronomy, and fluid dynamics, because of its ability to describe the behavior of electric, gravitational and fluid potentials. Poisson's equation is characterized by the function $u(x,y,z)$ consisting of three spatial variables (x,y,z). In three-dimensional Cartesian coordinates, Poisson's equation takes the form

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}=f(x,y,z) \quad \text{or equivalently,} \quad \nabla^2 u(x,y,z)=f(x,y,z)$$

where ∇^2 is called the Laplacian. The function $f(x,y,z)$ depends on the context under which the equation is being used. For example, in electrostatics, $f(x,y,z) = \rho(x,y,z)/\varepsilon_0$, where $\rho(x,y,z)$ is the charge distribution and ε_0 is a constant called the permittivity of free space and relates to a material's ability to transmit or "permit" an electric field. In the event that $f(x,y,z)=0$, we have what is known as Laplace's equation. In this section we will use the finite differences method to obtain numerical solutions to the two-dimensional elliptic partial differential equation given by

$$\nabla^2 u(x,y) \equiv \frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=f(x,y,z)$$

on a 2D-grid given by $R=(x,y)$ bounded by $a<x<b$ and $c<y<d$ with boundary conditions specified by $u(x,y)=g(x,y)$. Such solutions cover a wide range of practical problems and the method can be easily extended to 3D grids if so desired.

As usual, the first step is to choose integers n and m and define step sizes $h=(b-a)/n$ and $k=(d-c)/m$. This partitions the interval $[a,b]$ into n equal parts of width h and the interval $[c,d]$ into m equal parts of width k. Each mesh point (x_i,y_j) in the interior of the grid can be specified by

$$\begin{aligned} x_i &= x_0+ih \quad \text{for} \quad i{=}0,1,2,\ldots,n-1 \\ y_j &= y_0+jk \quad \text{for} \quad j{=}0,1,2,\ldots,m-1 \end{aligned}$$

Following the strategy described in section 17.2, we then use the Taylor series in the variable x about x_i and in the variable y about y_i to obtain the centered difference formulas given by

$$\begin{aligned} \left(\frac{\partial^2 u}{\partial x^2}\right)_{i,j} &\approx \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{h^2}+O(h^2) \\ \left(\frac{\partial^2 u}{\partial y^2}\right)_{i,j} &\approx \frac{u_{i,j+1}-2u_{i,j}+u_{i,j-1}}{k^2}+O(k^2) \end{aligned}$$

Using these formulas allows us to express the Poisson equation at the mesh points (x_i,y_j) as

$$\frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{h^2}+\frac{u_{i,j+1}-2u_{i,j}+u_{i,j-1}}{k^2}=f(x_i,y_j)+O(h^2)+O(k^2)$$

for each $i = 1,2,\ldots,n-1$ and $j = 1,2,\ldots,m-1$ and the boundary conditions can be expressed by

$$\begin{array}{llll} u(x_0,y_j) = g(x_0,y_j) & \text{and} \quad u(x_n,y_j) = g(x_n,y_j) & \text{for each} & j = 0,1,2,\ldots,m \\ u(x_i,y_0) = g(x_i,y_0) & \text{and} \quad u(x_i,y_m) = g(x_i,y_m) & \text{for each} & i = 1,2,\ldots,n-1 \end{array}$$

After some simplification, and using $w_{i,j}$ to approximate $u(x_i,y_j)$ the equations above reduce to

$$2\left[\left(\frac{h}{k}\right)^2 + 1\right] w_{i,j} - (w_{i+1,j} + w_{i-1,j}) - \left(\frac{h}{k}\right)^2 (w_{i,j+1} + w_{i,j-1}) = -h^2 f(x_i,y_j)$$

for each $i = 1,2,\ldots,n-1$ and $j = 1,2,\ldots,m-1$ and

$$\begin{array}{llll} w_{0,j} = g(x_0,y_j) & \text{and} \quad w_{n,j} = g(x_n,y_j) & \text{for each} & j = 0,1,2,\ldots,m \\ w_{i,0} = g(x_i,y_0) & \text{and} \quad w_{i,m} = g(x_i,y_m) & \text{for each} & i = 1,2,\ldots,n-1 \end{array}$$

If we use the information from the boundary conditions at all points (x_i,y_j) adjacent to a boundary mesh point, we have an $(n-1)(m-1) \times (n-1)(m-1)$ linear system with the unknowns being the approximations $w_{i,j}$ to $u(x_i,y_j)$ at the interior mesh points. Such linear systems, usually in symmetric-block tridiagonal form, can then be solved by some well known method like Gaussian elimination or Gauss-Seidel iteration.

The implementation in C# of the numerical solution of the Poisson equation that follows is based on the following data and boundary conditions:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = xe^y$$

where $0 < x < 2$, $0 < y < 1$ and subject to the boundary conditions

$$\begin{array}{llll} u(0,y) = 0 & \text{and} \quad u(2,y) = 2e^y & \text{where} & 0 \le y \le 1 \\ u(x,0) = x & \text{and} \quad u(x,1) = xe & \text{where} & 0 \le x \le 2 \end{array}$$

The linear system is solved using the Gauss-Seidel iteration method discussed in Chapter 8 with a the stopping criterion of

$$\left| w_{i,j}^{(l)} - w_{i,j}^{(l-1)} \right| \le 10^{-10}$$

for each $i = 1,2,3,4,5$ and $j = 1,2,3,4$. Under these conditions, the iteration maximum value was set to 100 but after 61 iterations it stopped because by then it had achieved the stopping criterion value of 10^{-10} as stated above. The finite difference algorithm is used in this example with $n = 6, m = 5$. The output of the results immediately follows the listing of the code. For convenience and for comparison with the calculated output values, the actual solution is given by $u(x,t) = xe^y$.

```
static double funcx(double x, double y)
{ return x * Math.Exp(y); }

static double gx(double x, double y)
{ return x * Math.Exp(y); }

static void Solving_Poisson_PDE_byFiniteDiff()
{
    /*  Solution to the Poisson PDE by finite difference method
     *  Goal: To approximate the solution of the Elliptic PDE
     *  DEL(u) = F(x,y), a <= x <= b, c <= y <= d,
     *  subject to the boundary conditions
     *  u(x,y) = G(x,y)
     *  if x = a or x = b for c <= y <= d,
     *  if y = c or y = d for a <= x <= b
     *  Input: endpoints a, b, c, d; integers m, n; tolerance TOL;
     *         maximum number of iterations M
     *  Output: approximations W(I,J) to u(X(I),Y(J)) for each
     *  I = 1,..., n-1 and J=1,..., m-1 or a message that the
     *  maximum number of iterations was exceeded.
     */
    double[,] W, ActualSolution;
    double[] X, Y;
    double TOL, A, B, C, D, H, K, lambda, mu, Z, E;
    int I, J, M, N, NN, M1, M2, N1, N2, L, LL;
    bool OK;
    W = new double[26, 26];
    ActualSolution = new double[26, 26];
    X = new double[26]; Y = new double[26];

    A = 0.0;     //These input parameters values
    B = 2.0;     //are set by the user and may
    C = 0.0;     //be varied accordingly
    D = 1.0;
    TOL = 1.0E-10;
    NN = 100;
    M = 5;
    N = 6;
    //Initialize variables with input parameter values
    M1 = M - 1; M2 = M - 2; N1 = N - 1; N2 = N - 2;
    H = (B - A) / N; K = (D - C) / M;
    // Construct mesh points
    for (I = 0; I <= N; I++) X[I] = A + I * H;
    for (J = 0; J <= M; J++) Y[J] = C + J * K;
    for (I = 1; I <= N1; I++)
    {
        W[I, 0] = gx(X[I], Y[0]);
        W[I, M] = gx(X[I], Y[M]);
    }
    for (J = 0; J <= M; J++)
    {
        W[0, J] = gx(X[0], Y[J]);
        W[N, J] = gx(X[N], Y[J]);
    }
    for (I = 1; I <= N1; I++)
        for (J = 1; J <= M1; J++) W[I, J] = 0.0;
```

```
lambda = H * H / (K * K);
mu = 2.0 * (1.0 + lambda);
L = 1;
OK = false;
/* Z is a new value of W(I,J) to be used in computing
 the norm of the error E used in place of NORM */
/* Perform Gauss-Seidel iterations */
while ((L <= NN) && (!OK))
{
    Z = (-H * H * funcx(X[1], Y[M1]) + gx(A, Y[M1]) +
    lambda * gx(X[1], D) + lambda * W[1, M2] + W[2, M1])/mu;
    E = Math.Abs(W[1, M1] - Z);
    W[1, M1] = Z;
    for (I = 2; I <= N2; I++)
    {
        Z = (-H*H*funcx(X[I],Y[M1])+lambda*gx(X[I],D)
             + W[I-1,M1]+W[I+1,M1]+lambda*W[I, M2])/mu;
        if (Math.Abs(W[I,M1]-Z) > E) E = Math.Abs(W[I M1]-Z);
        W[I, M1] = Z;
    }
    Z = (-H * H * funcx(X[N1], Y[M1]) + gx(B, Y[M1]) +
            lambda*gx(X[N1],D)+W[N2,M1]+lambda*W[N1,M2])/mu;
    if (Math.Abs(W[N1,M1]-Z) > E) E = Math.Abs(W[N1,M1]-Z);
    W[N1, M1] = Z;
    for (LL = 2; LL <= M2; LL++)
    {
        J = M2 - LL + 2;
        Z = (-H*H*funcx(X[1],Y[J])+gx(A,Y[J]) +
            lambda * W[1,J+1]+lambda*W[1,J-1]+W[2,J])/mu;
        if (Math.Abs(W[1,J]-Z) > E) E = Math.Abs(W[1,J]-Z);
        W[1, J] = Z;
        for (I = 2; I <= N2; I++)
        {
            Z = (-H*H*funcx(X[I],Y[J])+W[I-1,J] +
                    lambda*W[I,J+1]+lambda*W[I,J-1]+W[I+1,J])/mu;
            if (Math.Abs(W[I,J]-Z) > E) E = Math.Abs(W[I,J]-Z);
            W[I, J] = Z;
        }
        Z = (-H*H*funcx(X[N1],Y[J])+gx(B,Y[J])+W[N2,J] +
            lambda*W[N1,J+1]+lambda*W[N1,J-1])/mu;
        if (Math.Abs(W[N1,J]-Z) > E) E = Math.Abs(W[N1,J]-Z);
        W[N1, J] = Z;
    }
    Z = (-H*H*funcx(X[1],Y[1])+lambda*gx(X[1],C) +
            gx(A,Y[1])+lambda*W[1,2]+W[2,1])/mu;
    if (Math.Abs(W[1,1]-Z) > E) E = Math.Abs(W[1,1]-Z);
    W[1, 1] = Z;
    for (I = 2; I <= N2; I++)
    {
        Z = (-H*H*funcx(X[I],Y[1])+lambda*gx(X[I],C) +
            W[I+1,1]+W[I-1,1]+lambda*W[I,2])/mu;
        if (Math.Abs(W[I,1]-Z) > E) E = Math.Abs(W[I,1]-Z);
        W[I, 1] = Z;
    }
    Z = (-H*H*funcx(X[N1],Y[1])+lambda*gx(X[N1],C) +
            gx(B,Y[1])+W[N2,1]+lambda*W[N1,2])/mu;
        if (Math.Abs(W[N1,1]-Z) > E) E = Math.Abs(W[N1,1]-Z);
        W[N1, 1] = Z;
        if (E <= TOL)
        {
          Console.WriteLine("Solution of a Elliptic (Poisson) Partial
               Differential Equation");
          Console.WriteLine("using the Finite-Difference Method and
              user defined parameters\n");
          Console.WriteLine("I\tJ\tX(I)\t\t\tY(I)\tW(I,J)\t\
              tActualSolution(I,J)\n");
          for (I = 1; I <= N1; I++)
          {
            for (J = 1; J <= M1; J++)
            {
              ActualSolution[I,J] = X[I] * Math.Exp(Y[J]);
              Console.WriteLine(I + "\t" + J + "\t" + X[I].ToString("
                  0.000000000") + "\t" + Y[J].ToString("0.000000000")
                   + "\t" + W[I, J].ToString("0.000000000") + "\t" +
                  ActualSolution[I, J].ToString("0.000000000"));
            }
            Console.WriteLine("Convergence occurred on iteration
                number " + L + "\n");
          }
          OK = true;
        }
        else
            L++;
    }
   if (L > NN)
   Console.WriteLine("Method fails after iteration number "+NN+"\n");
   Console.ReadLine();
}
```

```
OUTPUT:
Solution of a Elliptic (Poisson) Partial Differential Equation
using the Finite-Difference Method and user defined parameters

I   J   X(I)          Y(I)          W(I,J)        ActualSolution(I,J)

1   1   0.333333333   0.200000000   0.407264617   0.407134253
1   2   0.333333333   0.400000000   0.497483246   0.497274899
1   3   0.333333333   0.600000000   0.607596080   0.607372933
1   4   0.333333333   0.800000000   0.742007065   0.741846976
Convergence occurred on iteration number 61

2   1   0.666666667   0.200000000   0.814523750   0.814268505
2   2   0.666666667   0.400000000   0.994957575   0.994549798
2   3   0.666666667   0.600000000   1.215183178   1.214745867
2   4   0.666666667   0.800000000   1.484008537   1.483693952
Convergence occurred on iteration number 61

3   1   1.000000000   0.200000000   1.221766210   1.221402758
3   2   1.000000000   0.400000000   1.492404725   1.491824698
3   3   1.000000000   0.600000000   1.822742714   1.822118800
3   4   1.000000000   0.800000000   2.225992702   2.225540928
Convergence occurred on iteration number 61
```

```
4   1   1.333333333   0.200000000   1.628963687   1.628537011
4   2   1.333333333   0.400000000   1.989778303   1.989099597
4   3   1.333333333   0.600000000   2.430226561   2.429491734
4   4   1.333333333   0.800000000   2.967928255   2.967387905
Convergence occurred on iteration number 61

5   1   1.666666667   0.200000000   2.036042323   2.035671264
5   2   1.666666667   0.400000000   2.486958389   2.486374496
5   3   1.666666667   0.600000000   3.037505503   3.036864667
5   4   1.666666667   0.800000000   3.709724067   3.709234881
Convergence occurred on iteration number 61
```

Control of approximation errors is central to the calculation of a numerical solution of acceptable accuracy. In the preceding examples, this control of errors can be accomplished in three ways:

1. The number of grid points can be varied. Since in the numerical analysis literature, the grid spacing is often given the symbol h, systematic variation of the grid spacing to investigate accuracy is usually termed h refinement.

2. The order of the approximation can be varied. In general, if the approximation is of order $O(h^p)$, it is termed p-th order. The symbol p is frequently used in the numerical analysis literature, and varying the order p to investigate accuracy is referred to as p refinement.

3. The discretization errors can possibly be estimated and where the error is considered too large, the numerical algorithm can automatically insert grid points. Beyond that, the grid spacing does not have to be uniform. That is, h does not have to remain constant throughout the numerical solution, and the numerical algorithm can automatically vary the spacing to concentrate the grid points where they are needed to achieve acceptable accuracy. This form of refinement is termed usually r refinement.

The preceding numerical solutions were developed using basic finite differences. However, many other approaches to approximating derivatives in PDEs have been developed and used. Among these are finite elements, finite volumes, weighted residuals, collocation, Galerkin and spectral methods. Each of these methods has advantages and disadvantages, often according to the characteristics of the problem of interest starting with the parabolic, hyperbolic and elliptic geometric classifications. Thus, an extensive literature for the numerical solution of PDEs is available, and we have only presented here a few basic concepts and examples.

18

Optimization Methods

18.1 Introduction

Optimization is the mathematical discipline concerned with finding the values of n-variables $(x_1, x_2, x_3, \ldots, x_n)$ that minimize or maximize some objective function, possibly subject to constraints. Optimization problems are made up of three basic ingredients:

1. An objective function which we want to minimize or maximize. Almost all optimization problems have a single objective function. The two interesting exceptions are:
   - No objective function. In some cases, such as the design of integrated circuit layouts, the goal is to find a set of variables that satisfies the constraints of the model. The user does not particularly want to optimize anything so there is no reason to define an objective function. This type of problems is usually called a feasibility problem.
   - Multiple objective functions. Ideally the user would like to optimize a number of different objectives at once. In the panel design problem, for example, it would be nice to minimize weight and maximize strength simultaneously. In practice, however, different objectives may not be compatible and, in addition, the variables that optimize one objective may be far from optimal for the others. Problems with multiple objectives are therefore reformulated as single-objective problems by either forming a weighted combination of the different objectives or else replacing some of the objectives by constraints.
2. A set of unknowns or variables which affect the value of the objective function. Having variables are absolutely essential. If there are no variables, we cannot define the objective function and the problem constraints.
3. A set of constraints that allow the unknowns to take on certain values but exclude others. Constraints are not essential. In fact, the field of unconstrained optimization is a large and important one for which a lot of algorithms and software are available.

In general, optimization problems can be divided into three basic groups:

1. Continuous optimization, in which all the variables are allowed to take values from subintervals of the real line. Here you can also distinguish between two types of continuous optimization:

   - Unconstrained optimization, which is concerned with the practical computational task of finding minima or maxima of functions of one, several or even millions of variables. Here, the appropriate computational method to use depends crucially on the nature of the function being optimized, the nature of the variables, as well as the number of variables.
   - Constrained optimization. Here you can also distinguish between several types of constrained optimization:
     (a) Nonlinearly constrained optimization where the general constrained optimization problem is to minimize a nonlinear function subject to nonlinear constraints.
     (b) Bound-constrained optimization where the parameters that describe physical quantities may be constrained to lie in a given range.
     (c) Quadratic programming where problems involve minimization of a quadratic function subject to linear constraints.
     (d) Linear programming where the goal is to minimize a linear objective function of continuous real variables, subject to linear constraints. The simplex and the interior-point Methods are the two most commonly used approaches to this kind of problem.
     (e) Semidefinite programming which is essentially an ordinary linear program where the constraint is replaced by a semidefinite constraint.
     (f) Stochastic programming where future events are taken into consideration as random variables but with constraints.
     (g) Network programming problems which arise, as the name indicates, in applications that can be represented as the flow of a commodity in a network and can also be linear or non-linear.

2. Discrete optimization, in which you require some or all of the variables to have integer values. Here you can also distinguish between two types of discrete optimization:

   - Integer programming. In many applications, such as the fixed-charge network flow problem and the famous traveling salesman problem, the solution of an optimization problem makes sense only if certain of the unknowns are integers.
   - Stochastic programming. It is often assumed that the data for the given problem are known accurately. However, for many actual problems, the data may not be known accurately for a variety of reasons. For example, it might be due to simple measurement error or perhaps some data represent information about the future and so it simply cannot be known with

certainty. The fundamental idea behind stochastic linear programming is the concept of recourse. Recourse is the ability to take corrective action after a random event has taken place. Therefore, data may be either known with certainty or, as in the case for future events, be stochastic or, in other words, are simply random.

3. Multi-objective optimization, where you would like to simultaneously optimize a number of different objectives. Most realistic optimization problems, particularly those in design, require the simultaneous optimization of more than one objective function. In these and most other cases, it is unlikely that the different objectives would be optimized by the same alternative parameter choices. Hence, some trade-off between the criteria is usually needed to ensure a satisfactory results.

18.2 Gradient Descent Method

The gradient or steepest descent method is an optimization algorithm for finding the local minimum value of multi-dimensional functions $f(x_1,x_2,\ldots,x_n)$. Although this method presumes that the gradient of the target function exists and can be calculated, there are cases where the function and therefore its gradient is not known explicitly. In such situations you can still construct the derivatives numerically from the two-point or three-point formulas and thus obtain an approximate value for the gradient. To find a local minimum of a function $f(x_1,x_2,\ldots,x_n)$ using the gradient descent method, one starts by making an approximate guess as to where the minimum value is located followed up by a series of iterative steps proportional to the negative of the gradient of the function at that point until some tolerance value is obtained. If the iterative steps are taken proportional to the positive value of the gradient, one then approaches a local maximum of that function and the procedure is then known as the gradient ascent method instead.

Gradient descent is based on the observation that if the real-valued function $G(\mathbf{x})$ is defined and differentiable in a neighborhood of a point $\mathbf{x_a}$, then $G(\mathbf{x})$ decreases fastest if one goes from $\mathbf{x_a}$ in the direction of the negative gradient of G at $\mathbf{x_a}$, $-\nabla G(\mathbf{x_a})$. It follows that, if

$$\mathbf{x_b} = \mathbf{x_a} - \alpha \nabla G(\mathbf{x_a})$$

for $\alpha > 0$ for a small enough number, then $G(\mathbf{x_a}) \geq G(\mathbf{x_b})$. With this observation in mind, one starts with a guess $\mathbf{x}_0$ for a local minimum of G, and then considers the sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ such that

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n \nabla G(\mathbf{x}_n),\ n \geq 0$$

We then have

$$G(\mathbf{x}_0) \geq G(\mathbf{x}_1) \geq G(\mathbf{x}_2) \geq \cdots \geq G(\mathbf{x}_n)$$

so hopefully the sequence $(\mathbf{x}_n)$ converges to the desired local minimum. Note that the value of the step size α is allowed to change at every iteration.

The code below shows an implementation of this gradient descent method for the function $f(x,y) = (x-1)^2e^{-y^2} + y(y+2)e^{-2x^2}$ around $x = 0.1$ and $y = -1$. Note that the spacing in the two point formula for the derivative is reduced by a factor of 2 after each iteration, and so is the step size in case of overshooting.

```
// Using the steepest-descent method to search
// for minimum values of a multi-variable function

public static void steepestDescent(double[] x, double alpha, double
    tolerance)
{
    int n = x.Length; //Size of input array
    double h = 1e-6;  //Tolerance factor
    double g0 = g(x); //Initial estimate of result

    //Calculate initial gradient
    double[] fi = new double[n];
    fi = GradG(x, h);

    //Calculate initial norm
    double DelG = 0;
    for (int i = 0; i < n; ++i)
        DelG += fi[i] * fi[i];
    DelG = Math.Sqrt(DelG);

    double b = alpha / DelG;

    //Iterate until value is <= tolerance limit
    while (DelG > tolerance)
    {
        //Calculate next value
        for (int i = 0; i < n; ++i)
            x[i] -= b * fi[i];
        h /= 2;

        fi = GradG(x, h); //Calculate next gradient

        //Calculate next norm
        DelG = 0;
        for (int i = 0; i < n; ++i)
            DelG += fi[i] * fi[i];
        DelG = Math.Sqrt(DelG);

        b = alpha / DelG;

        //Calculate next value
        double g1 = g(x);

        //Adjust parameter
        if (g1 > g0) alpha /= 2;
        else g0 = g1;
    }
}
```

```
// Provides a rough calculation of gradient g(x).
public static double[] GradG(double[] x, double h)
{
    int n = x.Length;
    double[] z = new double[n];
    double[] y = (double[]) x.Clone();
    double g0 = g(x);
    for (int i=0; i<n; ++i)
    {
        y[i] += h;
        z[i] = (g(y)-g0)/h;
    }
    return z;
}

// Method to provide function g(x).
public static double g(double[] x)
{
    return (x[0]-1)*(x[0]-1)*Math.Exp(-x[1]*x[1]) +
            x[1]*(x[1]+2)*Math.Exp(-2*x[0]*x[0]);
}

static void Main(string[] args)
{
    double tolerance = 1e-6;
    double alpha = 0.1;
    double[] x = new double[2];
    x[0] = 0.1; //Initial guesses
    x[1] =  -1; //of location of minimums
    steepestDescent(x, alpha, tolerance);
    Console.WriteLine("Testing steepest descent method\n");
    Console.WriteLine("The minimum is at x[0] = " + x[0] +", x[1] = "
         +x[1]);
    Console.ReadLine();
}
OUTPUT: The minimum is at x[0] = 0.107478502308767
                      and x[1] = -1.22316879147114
```

The program above finds a minimum at $x \approx 0.107355$ and $y \approx -1.223376$. This is a very simple but not a very efficient method. Like many other optimization methods, it simply converges to a local minimum near the starting point without providing any further information on the nature of the local minimum that was found. The search for a global minimum or maximum of a multi-variable function is actually a non-trivial process, especially when the function contains a significant number of minima or maxima, and searching for better and more reliable optimization methods is still very much an area of active research. For example, the bi-conjugate gradient method, the pre-conditioned conjugate gradient method and the nonlinear conjugate gradient method were all derived from the standard gradient method just discussed. In the last few decades, several advanced methods have been introduced for handling function optimization. However, because of the amount of material involved, only the simplex, simulated annealing and the genetic algorithm methods will be briefly discussed in the last sections of this chapter.

18.3 Linear Programming

Linear programming, sometimes also known as linear optimization, is a technique for optimizing a linear objective function $f(x_1, x_2, \dots, x_n)$, subject to linear equality and linear inequality constraints. Informally, linear programming determines the way to achieve the best outcome, such as maximum profit or lowest cost, in a given mathematical model having a list of requirements represented as linear equations. A linear programming problem may be defined as the problem of maximizing or minimizing a linear function subject to linear constraints. The constraints may be equalities or inequalities.

Many practical problems in operations research can be expressed as linear programming problems. Certain special cases of linear programming, such as network flow problems and multi-commodity flow problems are considered important enough to have generated much research on specialized algorithms for their solution. A number of algorithms for other types of optimization problems work by solving linear programming problems as sub-problems. Historically, ideas from linear programming have inspired many of the central concepts of optimization theory, such as duality, decomposition, and the importance of convexity and its generalizations. Likewise, linear programming is heavily used in microeconomics and company management, such as planning, production, transportation, technology and other issues. Although modern management issues are ever-changing, most companies would like to maximize profits or minimize costs with limited resources. Therefore, many practical real-world problems can be expressed as linear programming problems.

The *simplex method* is a systematic algorithm for generating and testing candidate solutions to a real-valued linear function of the form

$$f(x_1, x_2, \dots, x_n) = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n = \sum_{j=1}^{n} c_j x_j$$

which, in addition, is also subject to linear constraints. The linear programming problem is usually expressed in what is said to be the standard form as:

maximize: $\sum_{j=1}^{n} c_j x_j$

subject to: $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$ where $i = 1, 2, \dots, m < n$ and $x_j \geq 0$ where $j = 1, 2, \dots, n$

Here, the coefficients c_j represent the respective weights, or costs, of the variables x_j. The coefficients of the system of equations are represented by a_{ij}, and any constant values in the system of equations are combined on the right-hand side of the inequality in the variables b_j. In the standard form of the problem there are n variables and m constraints, not counting the n non-negativity constraints.

The method uses the geometrical concept of a simplex, which is a polytope of $N+1$ vertices in N dimensions: a line segment in one dimension, a triangle in two dimensions, a tetrahedron in three-dimensional space and so forth. A system of linear inequalities defines a polytope as a feasible region. The simplex algorithm begins by finding a basic feasible solution at a starting vertex and then moves along the edges of the polytope until it reaches the vertex of the optimum solution.

Linear programming problems must be converted into augmented form before being solved by the simplex algorithm. This form introduces non-negative *slack* variables, adding no cost to the solution, to replace inequalities with equalities in the constraints. The problem can then be re-written in the following form:

maximize: $\sum_{j=1}^{n} c_j x_j$

subject to: $\sum_{j=1}^{n} a_{ij} x_j = b_i$ where $(i = 1,2,\ldots,m)$ and $x_j \geq 0$ where $(j = 1,2,\ldots,n)$

or in matrix notation as

maximize: CX

subject to: $AX = B$ and $X \geq 0$

where the matrix A has m rows and $n+m$ columns with the last m columns forming an identity matrix. The vector x is of length $n+m$, and the column b is of length m. Finding a feasible solution in the augmented form corresponds to setting n of the $m+n$ variables (n original and m slack) to 0. We call such a setting of the variables a basic solution. The m variables which are purposely set to 0 are called the non-basic variables. We can then solve for the remaining n constraints, called the basic variables, which will be uniquely determined, while remaining careful not to step out of the feasible region.

Solving this problem then involves finding solutions to the set of equations satisfying the given constraints. Searching for possible solution begins at an arbitrary corner of the solution set. At each iteration, the simplex method selects the variable that will produce the largest change towards the optimal minimum or maximum solution. That variable replaces one of its fellow variables which is restricting it most severely, thus moving the current best found value to a different corner of the solution set and closer to the final solution. In addition, the simplex method can determine if no solution actually exists. This algorithm belongs to a general class of algorithms usually called greedy since it selects the best choice at each iteration without needing information from previous or future iterations.

Once a solution to the linear program has been found, successive improvements are made to the solution. In particular, one of the non-basic variables, with a value of zero, is chosen to be increased so that the value of the cost function, $\sum_{j=1}^{n} c_j x_j$, decreases. That variable is then increased, maintaining the equality of all the equations while keeping the other non-basic variables at zero, until one of the basic nonzero variables is reduced to zero and thus removed from the basis. At this point, a new solution has been determined at a different corner of the solution set. The process is then repeated with a new variable becoming basic as another becomes non-basic. Eventually, one of three things will happen. First, a solution may occur where no non-basic variable will decrease the cost, in which case the current solution is said to be the optimal solution. Second, a non-basic variable might increase to infinity without causing a basic-variable to become zero, resulting in an unbounded solution.

Finally, no solution may actually exist and the simplex method must abort. As is common for research in linear programming, the possibility that the simplex method might return to a previously visited corner will not be considered here.

18.3.1 The Revised Simplex Method

In 1954 Dantzig and Orchard-Hay [87] published an improved version of his original simplex algorithm, called the *revised simplex method*, to provide a more efficient way to solve linear programming problems. The revised simplex method describes linear programs as matrix entities and presents the simplex method as a series of linear algebra computations designed to exploit the fact that in many practical applications the coefficient matrix a_{ij} is very sparse meaning that most of its elements are equal to zero. As before, we start by expressing the linear programming problem in standard form

maximize: $\sum_{j=1}^{n} c_j x_j$

subject to: $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$ where $i = 1, 2, \ldots, m < n$ and $x_j \geq 0$ where $j = 1, 2, \ldots, n$

After introducing the slack variables $x_{n+1}, x_{n+2}, \ldots x_{n+m}$ the original problem can be re-expressed as

maximize: $\sum_{j=1}^{n} c_j x_j$

subject to: $\sum_{j=1}^{n} a_{ij} x_j = b_i$ where $(i = 1, 2, \ldots, m)$ and $x_j \geq 0$ where $(j = 1, 2, \ldots, n)$

or in matrix notation as

maximize: CX

subject to: $AX \leq B$ and $X \geq 0$

The matrix A has m rows and $n+m$ columns with the last m columns forming an identity matrix. The vector X is of length $n+m$ and the column B is of length m. A basic feasible solution X^* partitions X into X_B (m basic variables) and X_N (n non-basic variables). This corresponds to the partition of matrix A into A_B and A_N, and C into C_B and C_N. Each iteration of the revised simplex method can be described as follows [87] [88] [89]:

Step 1: Solve the system $yA_B = C_B$.

Step 2: Choose any column α of A_N such that $y\alpha$ is less than the corresponding component of C_N. If such a column does not exist, then the current solution is optimal.

Step 3: Solve the system $A_B\beta = \alpha$.

Step 4: Find the largest d such that $X_B^* - d\beta \geq 0$. If no such d is found, then the problem is unbounded, otherwise at least one component of $X_B^* - d\beta$ will be equal to zero and the corresponding variable leaves the basis.

Step 5: Set the entering variable to be d. Replace the values of the basic variables X_B^* by $X_B^* - d\beta$. Replace the leaving column of A_B by entering column, and replace the leaving variable by the entering variable.

An implementation in C# of the just described revised simplex algorithm that was originally published in Pascal by Syslo et al. [88] and later translated into Java by Lau [90] is provided below. The procedure parameters can be described as follows. The `maximize` boolean variable controls whether the objective function is to be maximized or minimized. `n` = number of variables, including the slack variables. `m` = number of constraints. `a[i,j]` where $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$ contains the coefficients of the constraints. `a[0,j]` contains the coefficients of the objective function. The right hand side of the constraints are given by `a[i][0]`. The other elements of the matrix A are not required as input. Upon exit, `a[0,0]` contains the optimal value of the objective function and `a[i][0]` contains the optimal value of the basic variable `basicvar[i]` for $i = 1, 2, \ldots, m$. `epsilon` is just the tolerance value below which values can be assumed to essentially be zero. If there is no feasible solution then `basicvar[m+1] > 0` otherwise it is equal to zero. If the problem has no finite solution then `basicvar[m+2] > 0` otherwise `basicvar[m+2] = 0`. `basicvar[i]` is the basic variable in the optimal solution for $i = 1, 2, \ldots, m$.

```
public static void revisedSimplex(bool maximize, int n, int m,
double[,] a, double epsilon, int[] basicvar)
{
  int i, j, k, m2, p, idx = 0;
  double[] objcoeff = new double[n + 1];
  double[] varsum = new double[n + 1];
  double[] optbasicval = new double[m + 3];
  double[] aux = new double[m + 3];
  double[,] work = new double[m + 3, m + 3];
  double part, sum;
  bool infeasible, unbound, abort, outres, iterate;

  if (maximize)
    for (j = 1; j <= n; j++)
      a[0, j] = -a[0, j];
  infeasible = false;
  unbound = false;
  m2 = m + 2;
  p = m + 2;
  outres = true;
  k = m + 1;
  for (j = 1; j <= n; j++)
  {
    objcoeff[j] = a[0, j];
    sum = 0.0;
    for (i = 1; i <= m; i++)
      sum -= a[i, j];
    varsum[j] = sum;
  }
  sum = 0.0;
  for (i = 1; i <= m; i++)
  {
    basicvar[i] = n + i;
    optbasicval[i] = a[i, 0];
    sum -= a[i, 0];
  }
  optbasicval[k] = 0.0;
  optbasicval[m2] = sum;
  for (i = 1; i <= m2; i++)
  {
    for (j = 1; j <= m2; j++)
      work[i, j] = 0.0;
    work[i, i] = 1.0;
  }
  iterate = true;
  do
  {
    if ((optbasicval[m2] >= -epsilon) && outres)
    {
      outres = false;
      p = m + 1;
    }
    part = 0.0;
    for (j = 1; j <= n; j++)
    {
      sum = work[p,m+1] * objcoeff[j] + work[p,m+2] * varsum[j];
      for (i = 1; i <= m; i++)
        sum += work[p, i] * a[i, j];
      if (part > sum)
      {
        part = sum;
        k = j;
      }
    }
    if (part > -epsilon)
    {
      iterate = false;
      if (outres)
        infeasible = true;
      else
        a[0, 0] = -optbasicval[p];
    }
    else
    {
      for (i = 1; i <= p; i++)
      {
        sum = work[i,m+1] * objcoeff[k] + work[i,m+2] * varsum[k];
        for (j = 1; j <= m; j++)
          sum += work[i, j] * a[j, k];
        aux[i] = sum;
      }
      abort = true;
      for (i = 1; i <= m; i++)
        if (aux[i] >= epsilon)
        {
          sum = optbasicval[i] / aux[i];
          if (abort || (sum < part))
          {
            part = sum;
            idx = i;
          }
          abort = false;
        }
      if (abort)
      {
        unbound = true;
        iterate = false;
      }
      else
      {
        basicvar[idx] = k;
        sum = 1.0 / aux[idx];
        for (j = 1; j <= m; j++)
          work[idx, j] *= sum;
        i = ((idx == 1) ? 2 : 1);
        do
        {
          sum = aux[i];
          optbasicval[i] -= part * sum;
          for (j = 1; j <= m; j++)
            work[i, j] -= work[idx, j] * sum;
          i += ((i == idx - 1) ? 2 : 1);
        } while (i <= p);
        optbasicval[idx] = part;
      }
    }
  } while (iterate);
  // return results
  basicvar[m + 1] = (infeasible ? 1 : 0);
  basicvar[m + 2] = (unbound ? 1 : 0);
  for (i = 1; i <= m; i++)
    a[i, 0] = optbasicval[i];
  if (maximize)
  {
    for (j = 1; j <= n; j++)
      a[0, j] = -a[0, j];
    a[0, 0] = -a[0, 0];
  }
}
```

The test problem for the dual simplex method was chosen to be

$$\text{Maximize} : 5x_1 + 5x_2 + 3x_3 = 0$$

Subject to:

$$\begin{array}{rrrrrrrl}
x_1 & +3x_2 & +\ x_3 & +x_4 & & & & =3 \\
-x_1 & & +3x_3 & & +\ x_5 & & & =2 \\
2x_1 & -\ x_2 & +2x_3 & & & +x_6 & & =4 \\
2x_1 & +3x_2 & -\ x_3 & & & & +x_7 & =2 \\
x_1, & x_2, & x_3, & x_4, & x_5, & x_6, & x_7 & \geq 0
\end{array}$$

The corresponding driver program in C# for the test problem above is given below.

```
static void Main(string[] args)
{
  int n = 7;
  int m = 4;
  double eps = 1.0e-5;
  int[] basicvar = new int[m + 3];
  double[,] a = {{0,  5,  5,  3,  0,  0,  0, 0},
                 {3,  1,  3,  1,  1,  0,  0, 0},
                 {2, -1,  0,  3,  0,  1,  0, 0},
                 {4,  2, -1,  2,  0,  0,  1, 0},
                 {2,  2,  3, -1,  0,  0,  0, 1}};

  revisedSimplex(true, n, m, a, eps, basicvar);

  Console.WriteLine("Testing the revised simplex algorithm\n");
  if (basicvar[m + 1] > 0)
    Console.WriteLine("No feasible solution.");
  else
  {
    if (basicvar[m + 2] > 0)
      Console.WriteLine("Objective function is unbound.");
    else
    {
      Console.WriteLine("Optimal solution found. \n\nBasic variable
            Value");
      for (int i = 1; i <= m; i++)
        Console.WriteLine("{0,10}\t {1,-10}", basicvar[i], a[i, 0]);
      Console.WriteLine("\nOptimal value of the objective function =
          {0}", a[0, 0]);
    }
    Console.ReadLine();
  }
}
OUTPUT:
Optimal solution found.
Basic Variable   Value
        3      1.03448275862069
        1      1.10344827586207
        4      0.034482758620631
        2      0.275862068965517
The optimal value of the objective function = 10
```

18.4 Simulated Annealing Method

As its name implies, the Simulated Annealing (SA) algorithm is a global optimization method that exploits an analogy between the way in which a metal cools and freezes into a minimum energy crystalline structure, through a physical process called the annealing, and the search for a good approximation to the global minimum of a given function in a large search space. If you heat a solid past its melting point and then cool it, the final structural properties of the solid depend on the rate of cooling. If the liquid is cooled slowly enough, large crystals will be formed. However, if the liquid is cooled too quickly then the crystal structure will likely have imperfections. This algorithm simulates the cooling process by gradually lowering the *temperature* of the system until it converges to a steady, *frozen* state. The simulated annealing algorithm offers a major advantage over some other global search methods by its ability to avoid becoming trapped at some undesired local minima. This is accomplished by using a random search strategy that not only accepts changes that decrease the objective function f but also some changes that increase it.

The internal details of this algorithm can be conceptually visualized by a geographical landscape such as that of a mountain range, containing two directional parameters: along the North-South and East-West directions. Finding the minimum of a function then becomes equivalent to finding the lowest valley in this terrain. The way that this algorithm approaches this problem is conceptually similar to a super bouncing ball that can bounce over mountains from valley to valley. The algorithm starts at a high *temperature*, where the temperature is just a conceptual parameter that mimics the effect of a fast moving particle in a hot object. This feature allows the ball to bounce over any mountain and have access to any valley. As the temperature of the ball drops, it can no longer bounce as high and so it tends to settle and become trapped in relatively smaller valleys. The mountain range can be physically described by a *cost function* and the two directional parameters can be characterized by probability distributions since they can generate possible valleys or states to explore. We can also define another distribution, called the acceptance distribution, which depends on the difference between cost functions of the present generated valley currently being explored and the last saved lowest cost valley. The acceptance distribution is then used to probabilistically decide whether to stay in a new lower valley or to bounce out of it. Both the generating and acceptance distributions depend on current temperature of the system.

More specifically, the simulated annealing algorithm starts by generating an initial solution, either randomly or heuristically constructed, and by initializing a parameter analogous to the temperature parameter T. Starting from an initial point, the algorithm takes a step and the objective function f is evaluated. The algorithm uses two combined search strategies: random walk and iterative improvement. When minimizing a function, any downhill step is accepted and the process repeats from this new point. However, there's also a probability given by $P = e^{-\Delta/T}$, where Δ is the change in the value of the objective function and T is a control parameter called the temperature, that an uphill step may also be accepted. This feature allows the algorithm to escape from potential local minima. As the optimization process proceeds, the length of the steps decline and the algorithm eventually closes in on the global minimum. However, most large optimization problems have many local minima and this optimization algorithm may also end up trapped in a local minimum. To get out of a local minimum, repeat this calculation process using a higher cost function. Since the algorithm starts with a high temperature, a new state with a larger cost will have a higher probability of being accepted. The simulated annealing algorithm can be summarized by the following general steps.

```
Initialization:
Start algorithm by entering some initial guess
as to what the solution values might be.
Call this the current_solution, temperature
and calculate the current_cost value
Loop
{
   new_state
   calculate new_cost
   if (current_cost - new_cost) <= 0 then
        current_state = new_state
   else
   {
        if exp((current_cost - new_cost)/temperature ) > random(0,1)
        {
           accept
           current_state = new_state
        }
        else
           reject
   }
   decrease the temperature
   if stop_criterion has been met then stop
   otherwise loop around one more time.
}
```

Note that there are two major processes that take place in the simulated annealing algorithm. First, for each temperature, the simulated annealing algorithm runs through a number of cycles predetermined by the programmer. As a cycle runs, the inputs are randomized and only randomizations which produce a better-suited set of inputs are retained. Once the specified number of cycles have been completed, the temperature is lowered and a check is made to determine whether or not the temperature has reached its lowest allowed value. If the temperature's lowest allowed value has not been reached, then another cycle of randomizations will take place. However, if the temperature is lower than the lowest temperature allowed, then the simulated annealing algorithm terminates.

At the core of the simulated annealing algorithm is the randomization of the input values. This randomization is ultimately what causes simulated annealing to alter the input values that the algorithm is seeking to minimize. However, there is no specific method defined by the simulated annealing algorithm for how to randomize the inputs. Instead, the exact nature by which this is done often depends upon the nature of the problem being solved. The randomization process must often be customized and adjusted accordingly for different types of problems.

The simulated annealing example provided below is meant to illustrate how this algorithm may be applied to finding the global minimum of multi-variable functions. The study of simulated annealing algorithms is still a very active and rapidly evolving field. New research results in this area are frequently being published indicating that fresh innovative ideas are routinely being introduced into existing implementations of this algorithm.

```
public delegate double MultiFunction(RVector x);

public static RVector Anneal(MultiFunction f, double[] x,
                             double TMin, int nMaxIterations)
{
  //Set the initial "temperature" value of the system
  double T0 = 1.0;

  //Set the temperature variable equal to the given
  //initial temperature value of the system
  double T = T0;

  //Calculate the initial function value with the
  //given input configuration array
  double f0 = f(new RVector(x));
  double f1 = 0.0;
  double Deltaf = 0.0;
  double[] xCurrentState;
  int j = 0;

  //Loop until the current "temperature" is less than some
  //user supplied minimum threshold temperature.
  do
  {
    j++;
    int i = 0; //Iteration counter
    //Loop until the iteration counter is < than some
    //minimum tolerance value specified by user
    while (i < nMaxIterations)
    {
      i++;
      //Generate a random perturbation of the configuration array
      //This method is given in chapter 10 which covers the topic
      //of random numbers and random distributions
      xCurrentState = RandomPerturbation((double[])x.Clone());

      //and update the target function with this new perturbed value
      //of the configuration array
      f1 = f(new RVector(xCurrentState));

      //Calculate the difference between the new and old function
      //values using the current and the previous perturbed
      //configuration array values.
      Deltaf = f1 - f0;

      //If this difference is < 0 update the previous configuration
      //and function values with the current ones otherwise anneal
      //the current difference in function values with the current
      //temperature, compare the result with a random value and
      //update the previous configuration and function values with
      //the current ones only if the random value is < than the
      //annealed value.
      if (Deltaf < 0)
      {
        x = xCurrentState;
        f0 = f1;
      }
      else
      {
        if (Math.Exp(-Deltaf / T) > rand.NextDouble())
        {
          x = xCurrentState;
          f0 = f1;
        }
      }
    }
    //Decrease the system temperature value
    T *= Math.Pow(0.9, j);
  }
  while (TMin < T);
  //Return the latest configuration value
  return new RVector(x);
}
```

To test the simulated annealing algorithm, I chose the two-dimensional Rosebrock function:

$$f(x,y) = (1-x)^2 + 100(y-x^2)^2$$

which has a known global minimum at $f(1,1) = 0$ and the one-dimensional function

$$f(x) = |x| + \sin(x)$$

which has a known global minimum at $f(0) = 0.$ The implementation of these test functions along with the corresponding output is given below:

```
static void SimulatedAnnealingTest()
{
    //Create input array with initial guesses
    double[] xydata1 = new double[] {1.5, 0.5};
    //Run the simulated annealing algorithm on initial guess
    RVector result1 = Anneal(f1, xydata1, 1e-15, 20000);
    //Print out result and function value
    Console.WriteLine("\nf(x_min,y_min) = f({0},{1}) = {2}\n\n",
                result1[0], result1[1], f1(result1).ToString());

    //Create input array with initial guesses
    double[] xydata2 = new double[] { 8.0 };
    //Run the simulated annealing algorithm on initial guess
    RVector result2 = Anneal(f2, xydata2, 1e-15, 20000);
    //Print out result and function value
    Console.WriteLine("\nf(x_min) = f({0}) = {1}",
                result2[0], f2(result2).ToString());
}
OUTPUT:
Simulated Annealing Test

TEST 1: Rosebrock function f(x,y) = (1-x)^2 + 100(y-x^2)^2
with known global minimums at f(1,1) = 0

Initial guess (x_0,y_0) = (1.5,0.5)
Running simulated annealing algorithm...please wait
Results obtained from simulated annealing algorithm:
```

```
f(x_min,y_min) =
f(0.998616928946903,0.997235306943664) = 1.91290705226745E-06

TEST 2: f(x) = |x| + sin(x)
with known global minimum at f(0) = 0

Initial guess (x_0) = (8)
Running simulated annealing algorithm...please wait
Results obtained from simulated annealing algorithm:

f(x_min) = f(1.75698543884312E-05) = 9.03967113836945E-16
```

18.5 Genetic Algorithms

Genetic algorithms belong to a general class of optimization algorithms called evolutionary algorithms. Evolutionary algorithms consist of a broad field of study whose primary focus is on finding optimization techniques implementing mechanisms inspired by biological evolution such as *inheritance*, *mutation*, *selection*, *crossover*, *reproduction*, *natural selection* and *survival of the fittest*. There are many kinds of optimization, but generally speaking optimization is the process of modifying a system to make some aspect of it work more efficiently or use fewer resources. For instance, a computer program may be optimized so that it executes more rapidly, or is capable of operating with less memory resources. In mathematics, optimization is a very important ongoing research topic that covers many additional subfields. A genetic algorithm (GA) is a technique used in computing to find exact or approximate solutions to optimization and search problems.

A genetic algorithm starts with a random set of solutions, represented by *chromosomes*, called a *population*. Solutions from one population are then taken and used to form a new population. This action is motivated by a hope that the new population will yield better results than the old one. Solutions which are selected to form new populations are called *offspring* and are selected according to their *fitness* which means that the more suitable they are, the more chances they have to *reproduce*. This process is repeated until some condition, such as the population size or improvement of the best solution, is satisfied. The following outline gives a brief summary of the key steps in the genetic algorithm procedure.

1. [Start] Generate a random population of n chromosomes. In other words, start by generating a set of random guesses to suitable solutions for the problem.

2. [Fitness] Evaluate the fitness function $f(x)$ of each chromosome x in the population. In other words, check to see how close your initial guesses are to a solution of the problem.

3. [New population] Create a new population by repeating the following steps until the new population is complete.
    (a) [Selection] Select two parent chromosomes from a population according to their fitness. The better the fitness, the bigger the chances of being selected.
    (b) [Crossover] With a crossover probability cross over the parents to form a new set of offspring, also known as children. If no crossover was performed, the offspring are an exact copy of the parents.
    (c) [Mutation] With a mutation probability mutate new offspring at each position in a chromosome.
    (d) [Accepting] Place the new offspring in a new population.
4. [Replace] Use the newly generated population for a further run of the algorithm.
5. [Test] If the end condition is satisfied, stop, and return the best solution in the current population.
6. [Loop] Otherwise go to step 2 and repeat this process.

The outline of a basic genetic algorithm presented above is actually very general. There are many things that can be implemented differently in various scenarios and problems. For example, the first question to address is how to create chromosomes and what type of encoding to use. The chromosome should in some way contain information about the solution which it represents. The most common way of encoding a chromosome is to use a binary string. Each chromosome has one binary string and each bit in this string can represent some characteristic of the solution. Sometimes the whole string can represent a number. Regardless of the choice that is ultimately made, there are many different ways of encoding a chromosome and how to go about doing this depends mainly on the problem to be solved.

After deciding on what encoding scheme to use, one then needs to decide on how to make a suitable crossover. Crossover selects genes from parent chromosomes and creates a new offspring. The simplest way to accomplish this is to choose some crossover point at random so that everything before this point copies from the first parent to the second parent and then everything after a crossover point copies just from the second parent. However, there are also many other ways to make a crossover. Crossovers can be rather complicated and often depends on the encoding of the chromosome. Selecting a good crossover method for a specific problem can significantly improve the performance time of the genetic algorithm. After a crossover is performed, mutation takes place. This is to help prevent solutions in a given population from falling into a local minimum of the solved problem. Mutation randomly changes some of the new offspring. For binary encoding, for example, we can switch a few randomly chosen bits from 1 to 0 or from 0 to 1. Mutation depends on both the encoding and the crossover.

Genetic algorithms have two critical parameters: crossover probability and mutation probability. Crossover probability indicates how often a crossover will be performed. If there is no crossover, then the offspring are an exact copy of the parents. If there is a crossover, then the offspring are made from parts of the parents' chromosome. If the crossover probability is 100%, then all the resulting offspring are made by the crossover. If it is 0%, then a whole new generation is made from the exact copies of chromosomes from the old population. However, this does not mean that the new generation is the same. A crossover is made with the hope that a new set of chromosomes will contain the good parts of the old chromosomes so that perhaps the new chromosomes will be better. However, it is a good idea to also let at least some part of the population survive to the next generation. Mutation probability indicates how often the parts of the chromosome will be mutated. If there is no mutation, then the offspring are taken after the crossover without any change. If mutation is performed, then part of chromosome is changed. If mutation probability is 100%, then the whole chromosome is changed, if it is 0%, nothing is changed. Mutation is made in order to prevent the genetic algorithm from also falling into local minimums.

Another important parameter is the population size. Population size indicates how many chromosomes are in one generation of the population. If there are too few chromosomes, genetic algorithms have few possibilities to perform crossovers and only a small part of the search space is explored. On the other hand, if there are too many chromosomes, the genetic algorithm slows down. Research has shown that after some limit, which depends mainly on encoding and the problem, it is no longer useful to increase the population size because it does not make solving the problem any faster.

Chromosomes are then selected from the population to be parents to the crossover. The question is how to select these chromosomes. According to Darwin's theory of evolution, the best chromosomes should survive and create new offspring. There are many methods on how to select the best chromosomes. Some examples include the roulette wheel selection, Boltzman selection, tournament selection, rank selection, steady state selection and so on. In the roulette wheel method, for example, the parents are selected according to their fitness. The better the chromosomes are, the more chances they have to be selected. Rank selection first ranks the population and then every chromosome receives a fitness score from this ranking. When creating a new population by crossover and mutation, there is always a chance that we will lose the best chromosome. Elitism is the name of a method, which first copies the best chromosome, or a few of the best chromosomes, to a new population leaving the rest to be done in the more traditional way. Elitism can very rapidly increase performance of GA because it prevents losing the best found solution.

In spite of enjoying a considerable amount of praise and success, it is important to remember that a genetic algorithm does not provide a magic bullet solution to all minimization or maximization problems. In many cases other algorithms are faster and more practical. However, for problems with a large parameter space and where the problem itself can be easily specified, genetic algorithms can be an appropriate method to arrive at a solution which would otherwise seem daunting if not perhaps impossible to find.

Genetic algorithms are still a very rapidly evolving field and new research results in this area are frequently published, indicating that there is always room for improvements in the existing implementations of the algorithm. The simplest case of applying genetic algorithms to optimize functions refers to the study of problems in which one seeks to minimize or maximize a real function by systematically choosing the values of real or integer variables from within an allowed set. The example provided below is meant only to illustrate a basic application of a genetic algorithm to the simplest kind of optimization problem and it is very reasonable to expect that improvements can be made to the code. The test function used is given by:

$$f(x,y) = 15xy(1-x)(1-y)\sin(\pi x)\sin(\pi y)$$

which has a known global maximum at $f(0,0)$. The initial parameters used for starting the genetic algorithm were crossover$=80\%$, mutation$=5\%$, population size $=100$, generation size $=2000$ and chromosome size $=2$.

```
public delegate double GAFunction(double[] values);
private static Random rand = new Random();

//A Genetic Algorithm class
public class GA
{
  public double MutationRate;
  public double CrossoverRate;
  public int ChromosomeLength;
  public int PopulationSize;
  public int GenerationSize;
  public double TotalFitness;
  public bool Elitism;
  private ArrayList CurrentGenerationList;
  private ArrayList NextGenerationList;
  private ArrayList FitnessList;
  static private GAFunction getFitness;
  public GAFunction FitnessFunction
  {
    get { return getFitness; }
    set { getFitness = value; }
  }

  //Constructor with user specified crossover rate,
  //mutation rate, population size, generation size
  //and chromosome length.

  public GA(double XoverRate, double mutRate, int popSize,
            int genSize, int ChromLength)
  {
    Elitism = false;
    MutationRate = mutRate;
    CrossoverRate = XoverRate;
    PopulationSize = popSize;
    GenerationSize = genSize;
    ChromosomeLength = ChromLength;
  }
```

```
//Method which launches the GA into execution mode.

public void LaunchGA()
{
  //Create the arrays to hold the fitness,
  //current and next generation lists
  FitnessList = new ArrayList();
  CurrentGenerationList = new ArrayList(GenerationSize);
  NextGenerationList = new ArrayList(GenerationSize);
  //and initilize the mutation rate.
  Chromosome.ChromosomeMutationRate = MutationRate;

  //Create the initial chromosome population by repeatedly
  //calling the user supplied fitness function
  for (int i = 0; i < PopulationSize; i++)
  {
    Chromosome g = new Chromosome(ChromosomeLength, true);
    CurrentGenerationList.Add(g);
  }
  //Rank the initial chromosome population
  RankPopulation();

  //Loop through the entire generation size creating
  //and evaluating generations of new chromosomes.
  for (int i = 0; i < GenerationSize; i++)
  {
    CreateNextGeneration();
    RankPopulation();
  }
}

//After ranking all the chromosomes by fitness, use a
//"roulette wheel" selection method that allocates a large
//probability of being selected to those chromosomes with the
//highest fitness. That is, preference in the selection process
//is biased towards those chromosomes exhibiting highest fitness.

private int RouletteSelection()
{
  double randomFitness = rand.NextDouble() * TotalFitness;
  int idx = -1;
  int mid;
  int first = 0;
  int last = PopulationSize - 1;
  mid = (last - first) / 2;
  while (idx == -1 && first <= last)
  {
    if (randomFitness < (double)FitnessList[mid])
    { last = mid; }
    else if (randomFitness > (double)FitnessList[mid])
    { first = mid; }
    mid = (first + last) / 2;
    if ((last - first) == 1) idx = last;
  }
  return idx;
}
```

```
// Rank population and then sort it in order of fitness.

private void RankPopulation()
{
  TotalFitness = 0;
  for (int i = 0; i < PopulationSize; i++)
  {
    Chromosome g = ((Chromosome)CurrentGenerationList[i]);
    g.ChromosomeFitness = FitnessFunction(g.ChromosomeGenes);
    TotalFitness += g.ChromosomeFitness;
  }
  CurrentGenerationList.Sort(new ChromosomeComparer());
  double fitness = 0.0;
  FitnessList.Clear();
  for (int i = 0; i < PopulationSize; i++)
  {
    fitness += ((Chromosome)CurrentGenerationList[i]).
        ChromosomeFitness;
    FitnessList.Add((double)fitness);
  }
}

//Create a new generation of chromosomes. There are many
//different ways to do this. The basic idea used here is
//to first check to see if the elitist flag has been set.
//If so, then copy the chromosomes from this generation
//to the next before looping through the entire chromosome
//population spawning and mutating children. Finally, if the
//elitism flag has been set, then copy the best chromosomes
//to the new population.

private void CreateNextGeneration()
{
  NextGenerationList.Clear();
  Chromosome g = null;
  if (Elitism)
    g = (Chromosome)CurrentGenerationList[PopulationSize - 1];
  for (int i = 0; i < PopulationSize; i += 2)
  {
    int pidx1 = RouletteSelection();
    int pidx2 = RouletteSelection();
    Chromosome parent1, parent2, child1, child2;
    parent1 = ((Chromosome)CurrentGenerationList[pidx1]);
    parent2 = ((Chromosome)CurrentGenerationList[pidx2]);

    if (rand.NextDouble() < CrossoverRate)
    { parent1.Crossover(ref parent2, out child1, out child2); }
    else
    {
      child1 = parent1;
      child2 = parent2;
    }
    child1.Mutate();
    child2.Mutate();
    NextGenerationList.Add(child1);
```

```
      NextGenerationList.Add(child2);
    }
    if (Elitism && g != null) NextGenerationList[0] = g;
    CurrentGenerationList.Clear();
    for (int i = 0; i < PopulationSize; i++)
      CurrentGenerationList.Add(NextGenerationList[i]);
  }

  //Extract the best values based on fitness from the current
  //generation. Since the ranking process already sorted the
  //latest current generation list, just pluck out the best
  //values from the current generation list.

  public void GetBestValues(out double[] values, out double fitness)
  {
    Chromosome g=((Chromosome)CurrentGenerationList[PopulationSize
        -1]);
    values = new double[g.ChromosomeLength];
    g.ExtractChromosomeValues(ref values);
    fitness = (double)g.ChromosomeFitness;
  }
}

public class Chromosome
{
  public double[] ChromosomeGenes;
  public int ChromosomeLength;
  public double ChromosomeFitness;
  public static double ChromosomeMutationRate;

  //Chromosome class constructor
  //Actual functionality is to set up an array
  //called ChromosomeGenes and depending on the
  //boolean flag createGenes, it may or may not
  //fill this array with random values from 0 to 1
  //up to some specified ChromosomeLength

  public Chromosome(int length, bool createGenes)
  {
    ChromosomeLength = length;
    ChromosomeGenes = new double[length];
    if (createGenes)
    {
      for (int i = 0; i < ChromosomeLength; i++)
        ChromosomeGenes[i] = rand.NextDouble();
    }
  }

  //Creates two offspring children using a single crossover point.
  //The basic idea is to first pick a random position, create two
  //children and then swap their genes starting from the randomly
  //picked position point.

  public void Crossover(ref Chromosome Chromosome2, out Chromosome
                        child1, out Chromosome child2)
  {
    int position=(int)(rand.NextDouble()*(double)ChromosomeLength);
    child1 = new Chromosome(ChromosomeLength, false);
    child2 = new Chromosome(ChromosomeLength, false);
    for (int i = 0; i < ChromosomeLength; i++)
    {
      if (i < position)
      {
        child1.ChromosomeGenes[i] = ChromosomeGenes[i];
        child2.ChromosomeGenes[i] = Chromosome2.ChromosomeGenes[i];
      }
      else
      {
        child1.ChromosomeGenes[i] = Chromosome2.ChromosomeGenes[i];
        child2.ChromosomeGenes[i] = ChromosomeGenes[i];
      }
    }
  }

  //Mutates the chromosome genes by randomly switching them around
  public void Mutate()
  {
    for (int position = 0; position < ChromosomeLength; position++)
    {
      if (rand.NextDouble() < ChromosomeMutationRate)
        ChromosomeGenes[position] =
       (ChromosomeGenes[position] + rand.NextDouble()) / 2.0;
    }
  }

  //Extracts the chromosome values
  public void ExtractChromosomeValues(ref double[] values)
  {
    for (int i = 0; i < ChromosomeLength; i++)
    values[i] = ChromosomeGenes[i];
  }
}

//Compares two chromosomes by their fitness values
public sealed class ChromosomeComparer : IComparer
{
  public int Compare(object x, object y)
  {
    if (!(x is Chromosome) || !(y is Chromosome))
      throw new ArgumentException("Not of type Chromosome");
    if (((Chromosome)x).ChromosomeFitness >
        ((Chromosome)y).ChromosomeFitness)
      return 1;
    else if (((Chromosome)x).ChromosomeFitness ==
             ((Chromosome)y).ChromosomeFitness)
      return 0;
    else
    return -1;
  }
}
```

```
public static double GenAlgTestFcn(double[] values)
{
 if (values.GetLength(0) != 2)
   throw new Exception("should only have 2 args");
 double x = values[0]; double y = values[1];
 return (15*x*y*(1-x)*(1-y)*Math.Sin(Math.PI*x)*Math.Sin(Math.PI*y));
}

public static void GeneticAlgorithmTest()
{
 GA ga = new GA(0.8, 0.05, 100, 2000, 2);
 ga.FitnessFunction = new GAFunction(GenAlgTestFcn);
 ga.Elitism = true;
 ga.LaunchGA();

 double[] values; double fitness;
 ga.GetBestValues(out values, out fitness);

 Console.WriteLine("Calculated max values are: \nx_max = {0}
                  \ny_max = {1}\n",values[0],values[1]);
 Console.WriteLine("f(x_max,y_max) = f({0},{1}) = {2}", values[0],
                  values[1], fitness);
 Console.WriteLine("\nPress ENTER to terminate program");
 Console.ReadLine();
}
```

```
OUTPUT:
Finding global optimum values to the function:

f(x,y) = 15xy(1-x)(1-y)sin(pi*x)sin(pi*y)

by using a genetic algorithm with initial parameters:

Crossover       =80%
Mutation        =5%
Population size =100
Generations     =2000
Chromosome size =2

Actual max values are: x_max = 0.5 and y_max = 0.5

Calculated max values are:
x_max = 0.500085205730929
y_max = 0.500116775161641

f(x_max,y_max) =
f(0.500085205730929,0.500116775161641) = 0.937499824963427
```

To further illustrate the power of genetic algorithms and how they have to be individually designed and constructed before they can be applied to solve specific problems, consider an example where we want to *evolve* a random string into a particular unknown target string supplied by the user at runtime. Since there are only a finite number of ASCII characters to check, one could conceivably write a program to loop through the entire set of all the available ASCII characters while attempting to individually match them against each character in the target string. As of date, there are a total of 128 ASCII characters, of whom 33 are non-printable and 94 are printable characters. This means that for every character in the target string we have to check at most 94 times to see if it matches one of the printable ASCII characters. As a result, even for a relatively short word like "bunny", which contains just 5 characters, we need to do at most 94×5 or 470 comparison checks. Therefore, the cost of using larger target strings can quickly escalate and slow down the computer time needed to find a correct character-by-character match and ultimately a complete solution to this problem.

Genetic algorithms, however, can significantly cut down the cost and time needed to solve this problem. While the basic ideas behind genetic algorithms remain the same, the gritty details differ considerably from problem to problem and must therefore be worked out individually in every situation. For example, in the string matching problem just introduced above, the genetic algorithm starts by making random guesses as to what the solution is. Then for each guess the genetic algorithm receives back one piece of very useful information in the form of a "fitness" value of the string. Therefore how we define the fitness function to calculate a fitness value for the string is a crucial piece of information in determining how well the genetic algorithm will work. Ultimately we would like to have the fitness function embody the idea of "survival of the fittest" in order to remain within the guidelines of genetic algorithms.

For the case of breeding correct strings, it is natural to define "fitness" to be the percentage of correct letters in each guess. This fitness measure is fine for short phrases, but problematic for long phrases because as we increase the length of the target phrase, the fitness difference between two phrases that differ by one correct letter becomes vanishingly small which could trigger pre-mature termination of the algorithm thereby yielding false results. Another important issue to consider is how the offspring selection is made. There are several ways one could imagine going about doing this. As a first approximation, we could allow only the fittest 50% of the population to reproduce. Unfortunately, this approach has the downside of not permitting much exploration in the DNA search space. After all, unfit individuals may still have some novel genes to contribute to the gene pool, so we don't want to discard genetic diversity too hastily. A much better approach is to use the normalized fitness scores as a sort of roulette wheel as described earlier in this section. This way, every individual has a chance of being selected with a probability equal to that of its fitness just like in nature. With the above scheme, we can now randomly select individuals to reproduce for the next generation. The actual "mating" can also mean several things. However, in the simplest case we perform an operation that is similar to a real-world occurrence called crossover. When two strings are crossed, the result is two new strings, each of which contains part of the genetic material of the parents. First, a random crossover point is chosen. At that point in both strings the genetic material from the left side of one parent is spliced to the material from the right side of the other parent. A second child can be produced by swapping and pairing the other sides. In this way, portions of the genetic material from two fit individuals can be merged so as to potentially produce even more viable offspring. Although some crossovers can also produce less fit individuals, this minor setback usually lasts only a single generation before the law of survival of the fittest tends to again favor survival for only the fittest offspring. Another important consideration is the mutation operator. Once again we do not want to apply the mutation operator to every member of the population, so we randomly select which offspring gets mutated genes. Mutation allows for entirely new genetic material to enter the population, which can yield tremendous rewards if the mutation is favorable. However, it is usually the case that mutations are detrimental and so is makes sense to allow mutation only for a small percentage of the population. These ideas are all taken into consideration in the C# implementation of the final genetic algorithm example below. Here the user is asked to enter a target search word or phrase and then a randomized string is made to *evolve* to the target string using the help of a genetic algorithm.

```
public class strChromosome
{
    public char[] Value;
    public int Fitness;

    //Constructor
    public strChromosome(int size)
    {
        Value = new char[size];
        Fitness = 0;
    }

    //Gets a random character from the available
    //range of ASCII character values
    public void RandomValue(Random rand)
    {
        for (int j = 0; j < Value.Length; j++)
        {
            Value[j] = (char)((126 * rand.NextDouble()) + 32);
        }
    }

    //Calcultes the fitness value by comparing the difference
    //between target and current string values using their
    //ASCII character values one by one.
    public void CalculateFitness(char[] target)
    {
        int fitness = 0;
        for (int j = 0; j < Value.Length; j++)
        {
            fitness += Math.Abs(Value[j] - target[j]);
        }
        Fitness = fitness;
    }
}

public class strGA
{
    public double elitismRate = 0.10;
    public double mutationRate = 0.25;
    public string targetWord;
    public int maxGenerations;
    public int maxPopulationPerGeneration;
    public List<strChromosome> Population;
    public List<strChromosome> NextGeneration;

    public void LaunchGA()
    {
        //Target string to find
        char[] target = targetWord.ToCharArray();
        //Size of string to find
        int targetSize = target.Length;

        //Instantiate result object to hold results
        GASearchResult result = new GASearchResult();

        //Initialize Population and nextGeneration lists
        Population = new
                List<strChromosome>(maxPopulationPerGeneration);
        NextGeneration =
                new List<strChromosome>(maxPopulationPerGeneration);

        //Generate Initial population
        GenerateInitialPopulation(targetSize,
            maxPopulationPerGeneration);

        //Print headings
        Console.WriteLine("\nString Value\tFitness Value\tGeneration
            Number\n");
        //Loop to process each generation
        for (int generationCounter = 0;
             generationCounter < maxGenerations; generationCounter++)
        {
            //Calculate fitness
            CalculateFitness(target);

            //Extract best values so far
            result.Best = GetBest();
            result.GenerationNumber = generationCounter;
            Console.WriteLine("{0}\t\t{1}\t\t{2}",
                new String(result.Best.Value), result.Best.Fitness,
                result.GenerationNumber);

            //Get current best results and check them against
            //target value. If they are equal, then stop
            if (result.Best.Fitness == 0) break;

            //Mate population
            MatePopulation(targetSize, maxPopulationPerGeneration,
                           elitismRate, mutationRate);

            //and swap them
            SwapPopulation();
        }
        Console.WriteLine("\nFINAL BEST RESULTS:");
        Console.WriteLine("{0}\t\t{1}\t\t{2}", new
                String(result.Best.Value), result.Best.Fitness,
                result.GenerationNumber);
    }

    private void GenerateInitialPopulation(int targetSize,
                 int maxPopulationPerGeneration)
    {
        for (int i = 0; i < maxPopulationPerGeneration; i++)
        {
            strChromosome c = new strChromosome(targetSize);
            c.RandomValue(rand);
            Population.Add(c);
        }
    }

    private void CalculateFitness(char[] target)
    {
        foreach (strChromosome c in Population)
        { c.CalculateFitness(target); }
    }

    private strChromosome GetBest()
    {
        strChromosome best = null;
        foreach (strChromosome c in Population)
        {
            if (best == null)
            {
                best = c;
                continue;
            }
            if (c.Fitness < best.Fitness) best = c;
        }
        return best;
    }

    private void MatePopulation(int targetSize,
        int maxPopulationPerGeneration,
        double elitismRate, double mutationRate)
    {
      int elitSize=(int)(maxPopulationPerGeneration*elitismRate);
      Elitism(elitSize);
      for (int i = elitSize; i < maxPopulationPerGeneration; i++)
      {
        NextGeneration.Add(new strChromosome(targetSize));
        int i1=(int)(rand.NextDouble()*maxPopulationPerGeneration);
        int i2=(int)(rand.NextDouble()*maxPopulationPerGeneration);
        int spos =(int)(rand.NextDouble()*targetSize);
        NextGeneration[i].Value =
         (new String(Population[i1].Value).Substring(0,spos)
        + new String(Population[i2].Value).Substring(spos,
         targetSize - spos)).ToCharArray();
        if (rand.NextDouble() < mutationRate)
                Mutate(NextGeneration[i], targetSize, rand);
      }
    }
```

```
    private void Mutate(strChromosome Chromosome, int targetSize,
        Random rand)
    {
        int ipos = (int)(rand.NextDouble() * targetSize);
        int mutantGene = (int)(rand.NextDouble()*126)+32;
        Chromosome.Value[ipos] = (char)mutantGene;
    }

    private void Elitism(int elitSize)
    {
        Population.Sort(new CompareByFitness());
        for (int i = 0; i < elitSize; i++)
        { NextGeneration.Add(Population[i]); }
    }

    private void SwapPopulation()
    {
        Population.Clear();
        foreach (strChromosome c in NextGeneration)
        { Population.Add(c); }
        NextGeneration.Clear();
    }
}

public class CompareByFitness : IComparer<strChromosome>
{
  //Compares two string chromosome objects by their fitness values
  public int Compare(strChromosome obj1,strChromosome obj2)
  {
    return obj1.Fitness.CompareTo(obj2.Fitness);
  }
}

public class GASearchResult
{
    public strChromosome Best;
    public int GenerationNumber;
}

public static void GeneticAlgorithmStringSearchTest()
{
    Console.WriteLine("Testing a genetic algorithm to search
                       a random string for a target value.\n");

    Console.Write("Enter a target string:");
    string targetString = Convert.ToString(Console.ReadLine());

    strGA ga = new strGA();
    ga.targetWord = targetString;
    ga.maxGenerations = 180;
    ga.maxPopulationPerGeneration = 30000;
    ga.LaunchGA();

    Console.WriteLine("\nPress ENTER to terminate program");
    Console.ReadLine();
}
```

```
Testing a genetic algorithm to search a random
string for a target value.

String Value    Fitness Value   Generation Number
evfcz           24              0
evfcz           24              1
atpe-           19              2
bwnpm           16              3
bwnpm           16              4
bznkx           9               5
axnkx           8               6
axnkx           8               7
drpny           7               8
cunnx           2               9
cunnx           2               10
cunnx           2               11
cunnx           2               12
cunnx           2               13
cunnx           2               14
cunnx           2               15
cunnz           2               16
cunnx           2               17
bunnx           1               18
bumny           1               19
bunnx           1               20
bumny           1               21
cunny           1               22
bunny           0               23

FINAL BEST RESULTS:
bunny           0               23
```

References

[1] http://www.ecma-international.org/publication/standards/ecma-334.htm.

[2] http://www.microsoft.com.

[3] http://www.icsharpcode.net.

[4] http://www.dotgnu.org.

[5] http://www.gotmono.com.

[6] http://www.microsoft.com/downloads.

[7] http://msdn.microsoft.com/en-us/library.

[8] http://www.asciitable.com.

[9] http://www.atm.ox.ac.uk/user/iwi/charmap.html.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Reading, Massachusetts, 1994. ISBN: 0-201-63361-2.

[11] http://msdn.microsoft.com/en-us/library/system.collections.aspx.

[12] http://msdn.microsoft.com/en-us/library/system.collections.generic.aspx.

[13] David Goldberg. What every computer scientist should know about floating-point arithmetic. *Computing Surveys*, March 1991.

[14] http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm.

[15] http://community.opensourcedotnet.info/blogs/mathnet_en/archive/2006/08/16/handling-floating-point-numbers.aspx.

[16] IEEE. Standard for binary floating-point arithmetic. *IEEE Standard 754*, 1985.

[17] http://msdn.microsoft.com/en-us/library/system.math.aspx.

[18] John W. Harris and Horst Stocker. *Handbook of Mathematics and Computational Science*. Springer, Berlin, Germany, 1998. ISBN: 0-387-94746-9.

[19] Milton Abramowitz and Irene Stegun. *Handbook of Mathematical Functions: with Formulas, Graphs, and Mathematical Tables*. Dover, Mineola, New York, 1965. ISBN: 0-486-61272-4.

[20] http://support.microsoft.com/kb/196652.

[21] Hans J. Weber and George B. Arfken. *Mathematical Methods for Physicists, 6th Ed.* Academic Press, Burlington, Massachusetts, 2005. ISBN: 978-0120598762.

[22] William Press, Saul Teukolsky, William Vetterling, and Brian Flannery. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, Cambridge, England, 2007. ISBN: 978-0-521-88068-8.

[23] P. Midy and Y. Yakolev. Computing some elementary functions of a complex variable. *Mathematics and Computers in Simulation*, 33:33–49, March 1991.

[24] R. Smith. Collected algorithms from C.A.C.M. ACM, New York, 1962.

[25] G.W. Stewart. A note on complex division. Comput. Sci. Technical Report Series TR-1206, University of Maryland, 1982.

[26] William J. Thompson. *Atlas for Computing Mathematical Functions: An Illustrated Guide for Practitioners With Programs in C and Mathematica.* Wiley-Interscience, New York, New York, 1997. ISBN: 978-0471002604.

[27] Laurentiu Dragan and Stephen M. Watt. Performance analysis of generics for scientific computing. In *Proc. 7th Internatioanl Symposium on Symbolic and Numeric Algorithms in Scientific Computing*, pages 93–100, Timisoara, Romania, September 25-29 2005. SYNASC, IEEE Press.

[28] Trey Nash. *Accelerated C# 2008.* Academic Press, Burlington, Massachusetts, 2007. ISBN: 978-1-59059-873-3.

[29] David Musser. Introspective sorting and selection algorithms software. *Practice and Experience*, 27(3):983–992, 1997.

[30] M. A. Bender, M. Farach-Colton, and M. A. Mosteiro. Insertion sort is o(n log n). *Theory of Computing Systems*, 39(3):391–397, 2006.

[31] Thomas H. Cormen, Charles E.Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (2nd edition).* McGraw-Hill, New York, New York, 2003. ISBN: 978-0-072-97054-8.

[32] Robert Sedgewick. *Bundle of Algorithms in Java, Third Edition, Parts 1-5: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms.* Addison-Wesley, Reading, Massachusetts, 2003. ISBN: 978-0-201-77578-5.

[33] Donald E. Knuth. *The Art of Computer Programming, Volumes 1-3.* Addison-Wesley, Reading, Massachusetts, 1998. ISBN: 978-0-201-48541-7.

[34] Stephen Lacy and Richard Box. A fast, easy sort. *Byte Magazine*, page 315, April 1991.

[35] http://www.cs.vu.nl/˜dick/gnomesort.html.

[36] Charles A. R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[37] Donald L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):3032, 1959.

[38] K. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, New York, New York, 2007. ISBN: 978-0-07-322972-0.

[39] http://msdn.microsoft.com.

[40] http://blogs.msdn.com/bclteam/archive/2008/04/09/working-with-signed-non-decimal-and-bitwise-values-ron-petrusha.aspx.

[41] http://www.hackersdelight.org/.

[42] http://bits.stephan-brumme.com/.

[43] http://graphics.stanford.edu/ seander/bithacks.html.

[44] F. S. Acton. *Numerical Methods That Usually Work*. AMS, Providence, Rhode Island, 1990. ISBN: 978-0883854501.

[45] H. E. Salzer. Lagrangian interpolation at the Chebyshev points, some unnoted advantages. *Comput. J.*, 15:156159, 1972.

[46] W. Werner. Polynomial interpolation: Lagrange versus Newton. *Math. Comput.*, 43:205–217, 1984.

[47] L. B. Winrich. Note on a comparison of evaluation schemes for the interpolating polynomial. *Comput. J.*, 12:154–155, 1969.

[48] H. Anton and C. Rorres. *Elementary Linear Algebra*. John Wiley and Sons, Hoboken, New Jersey, 1994. ISBN: 0-471-58741-9.

[49] James E. Gentle. *Random Number Generation and Monte Carlo Methods*. Springer, Berlin, Germany, 2004. ISBN: 978-0387001784.

[50] D. E. Knuth. *The Art of Computer Programming Vol. 2:Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, 1997. ISBN: 0-201-89684-2 Section 3.2.1: The Linear Congruential Method pp.1026.

[51] http://en.wikipedia.org/wiki/list_of_random_number_generators.

[52] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. Model. Comput. Simul.*, 8:3, 1998.

[53] http://www.math.sci.hiroshima-u.ac.jp/ m-mat/mt/emt.html.

[54] http://en.wikipedia.org/wiki/multiply-with-carry.

[55] http://www.agner.org/random.

[56] http://www.fourmilab.ch/hotbits/.

[57] http://www.lavarnd.org/.

[58] http://www.araneus.fi/products-alea-eng.html.

[59] http://www.randomnumbers.info/content/about.htm.

[60] http://random.irb.hr/.

[61] http://www.random.org/.

[62] http://http://www.ton.scphys.kyoto-u.ac.jp/ hideaki/res/histogram.html.

[63] H. Hideaki and S. Shinomoto. A method for selecting the bin size of a time histogram. *Neural Computation*, 19(6):1503–1527, 2007.

[64] Mario F. Triola. *Elementary Statistics*. Addison-Wesley, Reading, Massachusetts, 2009. ISBN 978-0321500243.

[65] Paul Bratley, Bennett Fox, and Linus Schrage. *A Guide To Simulation*. Springer-Verlag, New York, New York, 1987. ISBN: 0-387-96467-3.

[66] Hisashi Tanizaki. *Computational Methods in Statistics and Econometrics*. Dekker Inc., Monticello, New York, 2004. ISBN: 0-8247-4804-2.

[67] http://en.wikipedia.org/wiki/poisson_distribution.

[68] A.W. Kemp. Efficient generation of logarithmically distributed pseudo-random variables. *Applied Statistics*, 30:249–253, 1981.

[69] Richard L. Burden and J. Douglas Faires. *Numerical Analysis, 8th edition*. Brooks/Cole, Pacific Grove, California, 2004. ISBN = 0-534-39200-8.

[70] Steven Chapra and Raymond Canale. *Numerical Methods for Engineers*. McGraw-Hill, New York, New York, 2005. ISBN = 978-0073101569.

[71] Sylvan Elhay and Jaroslav Kautsky. Algorithm 655: Iqpack, fortran subroutines for the weights of interpolatory quadrature. *ACM Transactions on Mathematical Software*, 13(4):399–415, December 1987.

[72] http://www.alglib.net/integral/gq/.

[73] Malvin H. Kalos and Paula A. Whitlock. *Monte Carlo Methods*. Wiley-VCH, Darmstadt, Germany, 2008. ISBN: 978-3-527-40760-6.

[74] N. Metropolis, A. W. Rosenbluth, N. M. Rosenbluth, A. H. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[75] Ronald N. Bracewell. *The Fourier Transform and Its Applications*. McGraw-Hill, New York, New York, 2008. ISBN: 0-07-303938-1.

[76] Steven W. Smith. *Digital Signal Processing*. Elsevier, Maryland Heights, Missouri, 2003. ISBN: 978-0-7506-7444-7.

[77] David Freedman, Robert Pisani, and Roger Purves. *Statistics*. W. W. Norton, New York, New York, 2007. ISBN 978-0393929720.

[78] Selmer M. Johnson. Generation of permutations by adjacent transposition. *Mathematics of Computation*, 17(83):282–285, July 1963.

[79] Hale F. Perm. *Algorithm 115: C.A.C.M.*, 5(8):434–435, August 1962.

[80] http://www.cs.princeton.edu/introcs/23recursion/johnsontrotter.java.html.

[81] http://theory.cs.uvic.ca/inf/perm/perminfo.html.

[82] James D. McCaffrey. *.NET Automation Recipes*. Academic Press, Burlington, Massachusetts, 2006. ISBN: 1-59059-663-3.

[83] C. Lanczos. A precision approximation of the gamma function. *SIAM Journal on Numerical Analysis*, Ser.B, Vol.1:86–96, 1964.

[84] Philip R. Bevington and D. Keith Robinson. *Data Reduction and Error Analysis for the Physical Sciences*. McGraw-Hill, New York, New York, 2003. ISBN: 0-07-247227-8.

[85] E.N. Lorenz. Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences*, 20:130–141, March 1963.

[86] E. Issacson and H. Keller. *Analysis of Numerical Methods*. Dover Publications, New York, New York, 1994. ISBN: 978-0486680293.

[87] G.B. Dantzig and W. Orchard-Hay. The product form for the inverse in the simplex method. *Mathematical tables and Other Aids to Computation*, 8(46):64–67, April 1954.

[88] M. Syslo, N. Deo, and J.S. Kowalik. *Discrete Optimization Algorithms with Pascal Programs*. Prentice-Hall, Englewood Cliffs, New Jersey, 1983. ISBN: 978-0486453538.

[89] V. Chvatal. *Linear Programming*. W.H. Freeman, New York, New York, 1983. ISBN: 978-0716715870.

[90] Hang T. Lau. *A Java Library of Graph Algorithms and Optimization*. Chapman and Hall/CRC, Boca Raton, Florida, 2007. ISBN: 1-58488-718-4.