14 Data Visualisation in Excel

14.1 Introduction and Objectives

At some stage in software development we need to have tools to determine the accuracy of numerical methods. These tools help us visualise data and results in some way. To this end, we have chosen to use a small driver that the author developed to display array and matrix data in an Excel sheet. Our goal is to present data in Excel rather than using Excel as an input device (although this is possible by the use of Excel addins whose discussion is outside the scope of this book).

The Excel driver has been used for a number of years. In the current book the code has been upgraded, documented and improved. We have replaced legacy code by new code in C++11. Only one header file needs to be included. It can be used in a number of ways to promote developer productivity:

- Creating charts for presentations in articles and reports.
- Importing data into Excel for processing by Excel addins in C#, C++ or VBA. In this case we are using Excel as a kind of database.
- Quick testing and debugging: having developed a software prototype we can see if it is accurate by displaying its output in Excel.
- Analysing and experimenting with numerical algorithms: in this case we determine the effect of modifying certain parameters by displaying the results in Excel. For example, we may wish to examine what happens when we apply the explicit Euler scheme to the CIR interest-rate model when the Feller condition is not satisfied (Cox and Ross 1976). See exercises 4 and 5 of this chapter for some scenarios.

We shall use Excel driver in later chapters when presenting output from numerical schemes.

14.2 The Structure of Excel-related Objects

We give a global overview of the objects in Excel (the *Excel Object model*) that can be accessed from the Excel driver software. Understanding these objects will help if you decide to examine the corresponding C++ source code. We mention that there is very little documentation for C++ and most of our insights were based on reverse engineering of the code and documentation in VBA and C#. In all cases, the Excel objects use COM (*Component Object Model*) and they define interfaces that can be accessed programmatically. When writing code we note that Microsoft *IntelliSense* is very useful because it allows us to see the members of a given object. We give an introduction to COM in an appendix to this chapter.

We now discuss the most important Excel objects.

1. Application

This object represents the Excel application itself. It is the entry point to the running application and the related user objects. It contains application-wide settings as well as properties that return top-level objects.

2. Workbook

This represents a single workbook in the Excel application. A *workbook* is a member of the *Workbooks* collection, the latter being a property of the Excel application. We note that this property does not include open add-ins which are a special kind of hidden workbook. A workbook has a number of properties:

a) *Workbooks* property: we return a workbook by using a workbook name or index number.
b) *ActiveWorkbook* property: this returns the workbook that is currently active.
c) *ThisWorkbook* property: this is the workbook where the code is running. Normally this is the same as the active workbook.

3. Worksheet

This is a member of the *Worksheets* collection that contains all the Worksheet objects in a workbook. A workbook consists of worksheets. The following properties for returning a Worksheet object are:

a) *Worksheets* property: we return a worksheet by using a worksheet name or index number. The worksheet index number denotes the position of the worksheet in the workbook's tab bar.

b) *ActiveSheet* property: this returns the worksheet that is currently active.

We note that a Worksheet object is also a member of the *Sheets* collection.

4. Range

This object represents a cell, a row, a column or a selection of cells containing one or more contiguous object blocks of a cell or a 3-D range. It could even be a group of cells on multiple sheets. The *Range* object is the one that it is the most used in Excel applications.

5. Chart

This object represents a *chart* in a workbook. It can either be an embedded chart (in a *ChartObject*) or a separate chart sheet.
For more information we refer the reader to the Microsoft online documentation. A comprehensive introduction is given in Bovey, Wallentin, Bullen and Green 2009.

The driver software that we introduce in this chapter is minimalist. It is possible to add new functionality to the driver by consulting the Microsoft online documentation with code examples in C# or VBA. This is enough in our experience to write the corresponding code in C++.
We have created a class `ExcelDriver` and it contains member functions that delegate to the Excel COM-based objects.

14.3 Sanity Check: is the Excel Infrastructure up and Running?

In order to use the Excel driver, a given version of Microsoft Office and Excel must be installed on your system. Furthermore, we define a file called `ExcelImports.cpp` that tells the compiler which version to use. Some typical configurations for *Office 14* and *Office 16* are:

```
#import "C:\Program Files (x86)\Common Files
\Microsoft Shared\office14\mso.dll" \
rename("DocumentProperties", "DocumentPropertiesXL") \
rename("RGB", "RGBXL")

#import "C:\Program Files (x86)\Common Files
\Microsoft Shared\VBA\VBA6\vbe6ext.olb"

#import "C:\Program Files (x86)\Microsoft Office\Office14\EXCEL.EXE" \
rename("DialogBox", "DialogBoxXL") rename("RGB", "RGBXL") \
rename("DocumentProperties", "DocumentPropertiesXL") \
rename("ReplaceText", "ReplaceTextXL") \
rename("CopyFile", "CopyFileXL") no_dual_interfaces

// Excel 2016
#import "C:\Program Files\Microsoft Office\root\VFS\ProgramFilesCommonX86
\Microsoft Shared\OFFICE16\MSO.DLL"
#import "C:\Program Files\Microsoft Office\root\VFS\ProgramFilesCommonX86
\Microsoft Shared\VBA\VBA6\VBE6EXT.OLB"
#import "C:\Program Files\Microsoft Office\root\Office16\EXCEL.EXE"
```

You need to check that your settings are correct before proceeding. Next, we have provided a test program to check if the interface to Excel is working:

```
// ExcelHelloWorld.cpp
//
// (C) Datasim Education BV 2012-2017
//
#include <string>
#include <iostream>
#include "ExcelImports.cpp"

#include "atlsafe.h"// Data types

// Extract a value from a cell
double GetDoubleFromCell(Excel::_WorksheetPtr sheet, CComBSTR cell)
{

    Excel::RangePtr range=sheet->GetRange(CComVariant(cell));
    _variant_t value=range->GetValue2();
    return value;
}

// Put a double value in an Excel cell
void SetCellValue(Excel::_WorksheetPtr sheet, CComBSTR cell, double value)
{
    Excel::RangePtr range=sheet->GetRange(CComVariant(cell));
    range->Value2=value;
}


int main()
{

    Excel::_ApplicationPtr xl;      // Pointer to Excel.


    try
    {
       // Initialize COM Runtime Libraries.
       CoInitialize(NULL);

       // Start excel application.
       xl.CreateInstance(L"Excel.Application");
       xl->Visible = VARIANT_TRUE;
       xl->Workbooks->Add((long) Excel::xlWorksheet);

       // Rename 1 to "Chart Data".
       Excel::_WorkbookPtr workbook = xl->ActiveWorkbook;

// Language-independent
       Excel::_WorksheetPtr sheet = workbook->Worksheets->GetItem(1);
       sheet->Name = "Chart Data";


       double val = 2.0;

       // Display values in Excel
       SetCellValue(sheet, "B9", val);
       SetCellValue(sheet, "B10", val*3);

       // Get the parameters
       double T = GetDoubleFromCell(sheet, "B9");
       double r = GetDoubleFromCell(sheet, "B10");

       // Display values in Excel somwhere else
       SetCellValue(sheet, "C9", T);
       SetCellValue(sheet, "C10", r);
    }
    catch( _com_error & error )
    {
       bstr_t description = error.Description();
       if( !description )
       {
          description = error.ErrorMessage();
       }
       std::cout << std::string(description);
    }

    CoUninitialize();

    return 0;
}
```

You should run this program before proceeding.

14.4 `ExcelDriver` and Matrices

Matrices are very important in numerical analysis and its applications. They are used as input to numerical processes as well as holding the results of computations. Some examples are:

- Two-dimensional grids of computed data, for example two-asset option prices; rows correspond to the first underlying value and the columns correspond to the second underlying value.

- *Look-up tables* that are generated by discretising functions, for example a table of values of the noncentral chi squared distribution. The discrete values of the noncentrality and number of degrees of freedom parameters correspond to row and column values, respectively.
- Using Excel as a database to store two-dimensional and range data. For example, we could create data in C++ and then export it to Excel for further processing in VBA, for example.

The current version of the software supports generic matrix types and we have tested it using Boost uBLAS matrices and a simple user-defined matrix class that is essentially a container for two-dimensional rectangular data. Regarding vectors, we use `std::vector` in the current session of the software driver for convenience. The basic user-defined matrix class is:

```
#ifndef NestedMatrix_HPP
#define NestedMatrix_HPP

template <typename T> class NestedMatrix
{ // Simple class just to show Excel interoperability
private:
    std::vector<std::vector<T>> mat;

    std::size_t nr;
    std::size_t nc;
public:
    NestedMatrix(std::size_t rows, std::size_t cols) : nr(rows), nc(cols)
    {
       mat = std::vector<std::vector<T>>(nr);
       for (std::size_t i = 0; i < nr; ++i)
       {
          mat[i] = std::vector<T>(nc);
       }
    }

    NestedMatrix(const NestedMatrix<T>& m2) : nr(m2.nr), nc(m2.nc)
    {
       mat   = std::vector<std::vector<T>>(nr);

       for (std::size_t i = 0; i < nr; ++i)
       {
          mat[i] = m2.mat[i];
       }
    }

    T& operator ()(std::size_t row, std::size_t col)
    {
       return mat[row][col];
    }

    const T& operator ()(std::size_t row, std::size_t col) const
    {
       return mat[row][col];
    }

    NestedMatrix operator - (const NestedMatrix<T>& m2)
    {

       NestedMatrix<T> diff(nr, nc);

       for (std::size_t i = 0; i < nr; ++i)
       {
          for (std::size_t j = 0; i < nc; ++j)
          {
             diff(i, j) = (*this)(i, j) - m2(i, j);
          }
       }
    }

    std::size_t size1() const
    {
       return nr;
    }

    std::size_t size2() const
    {
       return nc;
    }
};

#endif
```

The class interface has been designed to conform to that of Boost uBLAS. Furthermore, we need two functions to convert in two directions between generic matrix and `std::vector` types:

```
template <typename Matrix, typename Vector>
    Vector CreateVector(const Matrix& mat, long row)
{

    Vector result(mat.size2());
    for (std::size_t i = 0; i < result.size(); ++i)
    {
       result[i] = mat(row, i);
    }

    return result;
}

template <typename Matrix, typename Vector>
    Matrix CreateMatrix(const Vector& vec)
{
    Matrix result(1, vec.size());
    for (std::size_t j = 0; j < result.size2(); ++j)
    {
       result(0,j) = vec[j];
    }
    return result;
}
```

There are two overloaded functions to present matrix data in Excel:

```
// Matrix visualisation
template <typename Matrix>
    void AddMatrix(const Matrix& matrix,
       const std::string& name = std::string("Matrix"),
       long row = 1, long col = 1);

template <typename Matrix>
    void AddMatrix(const Matrix& matrix,
       const std::string& sheetName,
       const std::list<std::string>& rowLabels,
       const std::list<std::string>& columnLabels,
       long row = 1, long col = 1);
```

Both functions print the matrix data in Excel. The second function annotates the rows and columns of the matrix with user-friendly labels. This option can be useful when inspecting output and when we wish to use label values as the variables of interest instead of using numerical indexes.

We now discuss some examples. For completeness, the classes tested are as follows:

```
#include <boost/numeric/ublas/matrix.hpp>
#include "NestedMatrix.hpp"

using NumericMatrix = boost::numeric::ublas::matrix<double>;
//  using NumericMatrix = NestedMatrix<double>;
using Vector = std::vector<double>;
```

You can add your own favourite matrix class to this list and the code will run as long as it conforms to the above matrix interface. The vector class used is hard-coded.

14.4.1 Displaying a Matrix

We first take the simplest case of creating a matrix, setting up the input parameters and then displaying the matrix starting at a given row and column positions in the upper left-hand corner. We construct the matrix as follows:

```
std::size_t N = 10; std::size_t M = 6; // rows and columns
NumericMatrix matrix(N + 1, M + 1);
for (std::size_t i = 0; i < matrix.size1(); ++i)
```

```
{
    for (std::size_t j = 0; j < matrix.size2(); ++j)
    {
       matrix(i, j) = static_cast<double>(i + j);
    }
}
```

We start Excel and display the matrix using the following code:

```
// Start Excel
ExcelDriver& excel = ExcelDriver::Instance();

// Call Excel print function
std::string sheetName("Test Case 101 Matrix");
long row = 4; long col = 2;
excel.AddMatrix<NumericMatrix>(matrix, sheetName, row, col);
```

The output is shown in Figure 14.1.

0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12
7	8	9	10	11	12	13
8	9	10	11	12	13	14
9	10	11	12	13	14	15
10	11	12	13	14	15	16

Figure 14.1 Test Case 101 Matrix

We applied the *Singleton* design pattern (GOF 1995) to create a global instance of the Excel driver in the above code. It is also possible to create an instance using the following:

```
ExcelDriver xl; xl.MakeVisible(true);
```

By default Excel is started in invisible mode.

14.4.2 Displaying a Matrix with Labels

In some cases we may wish to annotate the rows and columns of a matrix before presenting it in Excel. This feature promotes the readability and understandability of the displayed data. In general, the label values tell us more about the semantics of the data. This feature is useful when creating lookup tables, for example.

We take an example. First, we create row and column labels:

```
// Labels for rows and columns of the Excel matrix.
// Only labelled values are printed!!
std::list<std::string> rowLabels{"A","B","C","D","E", "F","K","L"};
std::list<std::string> colLabels{"C1", "C2", "C3", "C4","C5"};

std::string sheetName2("Matrix Labels Case");

ExcelDriver& excel = ExcelDriver::Instance();
excel.MakeVisible(true);           // Default is INVISIBLE!
```

We now present the annotated matrix using some matrix data as in Figure 14.1 using the following code:

```
long rowPos = 4; long colPos = 3;
excel.AddMatrix<NumericMatrix>(matrix, sheetName2,
rowLabels, colLabels, rowPos, colPos);
```

The output is shown in Figure 14.2.

	C1	C2	C3	C4	C5		
A	0	1	2	3	4	5	6
B	1	2	3	4	5	6	7
C	2	3	4	5	6	7	8
D	3	4	5	6	7	8	9
E	4	5	6	7	8	9	10
F	5	6	7	8	9	10	11
K	6	7	8	9	10	11	12
L	7	8	9	10	11	12	13

Figure 14.2 Test Case 101 Matrix with Annotation

We stress that only labelled rows are displayed in the current version of the software. In most cases, we would ensure that the number of labels should be the same as the corresponding matrix dimensions order to view all the data. We thus see that not all rows from Figure 14.1 are visible in Figure 14.2. You need to be aware of this fact.

14.4.3 Lookup Tables, continuous and discrete Functions

A common use case is when we discretise functions of arity one and two. Focusing on the latter case we create two discrete mesh arrays and then we compute the value of a function at these points. These values will be the values in the matrix. As an example, we discretise the bivariate normal probability density function defined in the following way for convenience (the correlation variable is global):

```
double rho = 0.5;
double NormalPdf2d(double x, double y)
{ // Bivariate normal density function

       double fac= 1.0/(2.0 * 3.14159265359 * std::sqrt(1.0 - rho*rho));
       double t = x*x - 2.0*rho*x*y + y*y;
       t /= 2.0 * (1.0 - rho*rho);

       return fac * std::exp(-t);
}
```

In order to discretise this function we use the following utility function:

```
template <typename Matrix>
    Matrix CreateDiscreteFunction2d(const std::vector<double>& x ,
                                    const std::vector<double>& y,
                                    const std::function<double (double x, double y)>& f)
{ // Create a discrete function from a continuous function m = f(x,y)

    std::size_t nr = x.size();
    std::size_t nc = y.size();

    Matrix m(nr,nc);

    for (std::size_t i = 0; i < nr; ++i)
    {
       for (std::size_t j = 0; j < nc; ++j)
       {
          m(i, j) = f(x[i],y[j]);
       }
    }

    return m;
}
```

Next, we define mesh points in the x and y directions by calling a mesh generation function (we show the code of the end of section 14.4.3) as follows:

```
// Using mapping for continuous space to discrete space
std::size_t N = 20; std::size_t M = 10;
```

```
auto x = CreateMesh(N, -4.0, 4.0);
auto y = CreateMesh(M, -4.0, 4.0);

NumericMatrix matrix = DiscreteNormalPdf2d<NumericMatrix>(x, y);
```

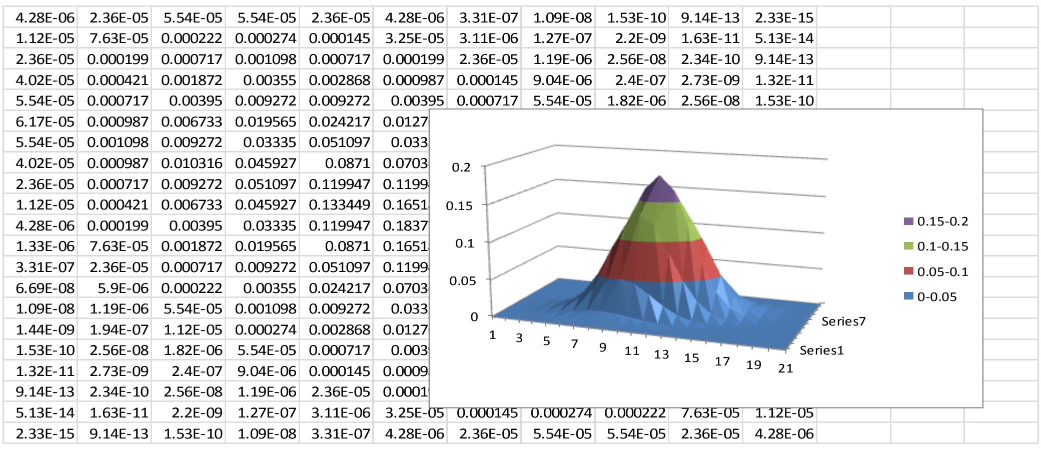
where:

```
template <typename Matrix>
   Matrix DiscreteNormalPdf2d(const std::vector<double>& x,
                              const std::vector<double>& y)
{
      return CreateDiscreteFunction2d<Matrix>(x, y, NormalPdf2d);
}
```

We now present the matrix as follows:

```
//  ExcelDriver& excel = ExcelDriver::Instance();
std::string sheetName("Bivariate Normal pdf");
long row = 1; long col = 1;
excel.AddMatrix<NumericMatrix>(matrix, sheetName, row, col);
```

The output is shown in Figure 14.3.



4.28E-06	2.36E-05	5.54E-05	5.54E-05	2.36E-05	4.28E-06	3.31E-07	1.09E-08	1.53E-10	9.14E-13	2.33E-15
1.12E-05	7.63E-05	0.000222	0.000274	0.000145	3.25E-05	3.11E-06	1.27E-07	2.2E-09	1.63E-11	5.13E-14
2.36E-05	0.000199	0.000717	0.001098	0.000717	0.000199	2.36E-05	1.19E-06	2.56E-08	2.34E-10	9.14E-13
4.02E-05	0.000421	0.001872	0.00355	0.002868	0.000987	0.000145	9.04E-06	2.4E-07	2.73E-09	1.32E-11
5.54E-05	0.000717	0.00395	0.009272	0.009272	0.00395	0.000717	5.54E-05	1.82E-06	2.56E-08	1.53E-10
6.17E-05	0.000987	0.006733	0.019565	0.024217	0.0127					
5.54E-05	0.001098	0.009272	0.03335	0.051097	0.033					
4.02E-05	0.000987	0.010316	0.045927	0.0871	0.0703					
2.36E-05	0.000717	0.009272	0.051097	0.119947	0.1199					
1.12E-05	0.000421	0.006733	0.045927	0.133449	0.1651					
4.28E-06	0.000199	0.00395	0.03335	0.119947	0.1837					
1.33E-06	7.63E-05	0.001872	0.019565	0.0871	0.1651					
3.31E-07	2.36E-05	0.000717	0.009272	0.051097	0.1199					
6.69E-08	5.9E-06	0.000222	0.00355	0.024217	0.0703					
1.09E-08	1.19E-06	5.54E-05	0.001098	0.009272	0.033					
1.44E-09	1.94E-07	1.12E-05	0.000274	0.002868	0.0127					
1.53E-10	2.56E-08	1.82E-06	5.54E-05	0.000717	0.003					
1.32E-11	2.73E-09	2.4E-07	9.04E-06	0.000145	0.0009					
9.14E-13	2.34E-10	2.56E-08	1.19E-06	2.36E-05	0.0001					
5.13E-14	1.63E-11	2.2E-09	1.27E-07	3.11E-06	3.25E-05	0.000145	0.000274	0.000222	7.63E-05	1.12E-05
2.33E-15	9.14E-13	1.53E-10	1.09E-08	3.31E-07	4.28E-06	2.36E-05	5.54E-05	5.54E-05	2.36E-05	4.28E-06

Figure 14.3 Bivariate Normal pdf

This code can be generalised to work with arbitrary functions of arity one and two. See exercise 1 below. In the interest of completeness, we show the basic code that creates mesh points on an interval:

```
std::vector<double> CreateMesh(std::size_t n, double a, double b)
{ // Create a mesh of size n+1 on closed interval [a,b]

      std::vector<double> x(n + 1);
      x[0] = a; x[x.size()-1] = b;

      double h = (b - a) / static_cast<double>(n);
      for (std::size_t j = 1; j < x.size() - 1; ++j)
      {
         x[j] = x[j - 1] + h;
      }

      return x;
}
```

14.5 `ExcelDriver` and Vectors

We see vectors as special cases of matrices in this context; a vector is a matrix with one row. The driver code converts a vector to a matrix and then calls the corresponding function that displays matrices. The corresponding interface is:

```
// Vector visualisation as numbers
template <typename Matrix>
   void AddVector(const std::vector<double>& vec,
                  const std::string& sheetName = std::string("Vector"),
                  long row = 1, long col = 1);

template <typename Matrix>
   void AddVector(const std::vector<double>& vec,
                  const std::string& sheetName,
                  const std::string& rowLabel,
                  const std::list<std::string>& columnLabels,
                  long row = 1, long col = 1);
```

This code is a special case of the discussion in section 14.4 and for this reason we do not discuss it in detail. For completeness, we give an example:

```
std::string rowLabel("row1");
std::list<std::string> colLabels{"C1", "C2", "C3", "C4","C5"};

Vector myVector(colLabels.size());
for (std::size_t i = 0; i < myVector.size(); ++i)
{

    myVector[i] = static_cast<double>(i);
}


long row = 4; long col = 3;
excel.AddVector<NumericMatrix>(myVector, sheetName, rowLabel, colLabels, row, col);
```

A possibly more interesting use case is when we wish to display one or more vectors as line graphs as a way to visualise them. This is the subject of the next section.

14.5.1 Single and Multiple Curves

We have two functions to display vectors. The first function displays a list of vectors while the second function displays a single vector. The interfaces are:

```
// Create chart with a number of functions. The arguments are:
// x:             std::vector<double> with input values
// labels:        labels for output values
// vectorList:    list of vectors with output values.
// chartTitle:    title of chart
// xTitle:        label of x axis
// yTitle:        label of y axis
void CreateChart(const std::vector<double> & x,
                 const std::list<std::string> & labels,
                 const std::list<std::vector<double>> & vectorList,
                 const std::string& chartTitle,
                 const std::string& xTitle = "X",
                 const std::string& yTitle = "Y");

// Create chart with a number of functions. The arguments are:
// x:             std::vector<double> with input values
// y:             std::vector<double> with output values.
// chartTitle:    title of chart
// xTitle:        label of x axis
// yTitle:        label of y axis

void CreateChart(const std::vector<double> & x,
                 const std::vector<double>& y,
                 const std::string& chartTitle,
                 const std::string& xTitle = "X",
                 const std::string& yTitle = "Y");
```

Using these functions is similar to the approach taken in section 14.4. We initially consider the case of the function $f(x) = x^2$. Note that it is convenient to define it as a stored lambda function `fun`:

```
long N = 40;

// Create abscissa x array
double A = 0.0; double B = 10.0;
auto x = CreateMesh(N, A,B);

auto fun = [](double x) { return x*x; };
auto vec1 = CreateDiscreteFunction< std::vector<double>>(x, fun);
```

```
ExcelDriver xl; xl.MakeVisible(true);
xl.CreateChart(x, vec1, "Test 101 curve);
```

The output is shown in Figure 14.4.

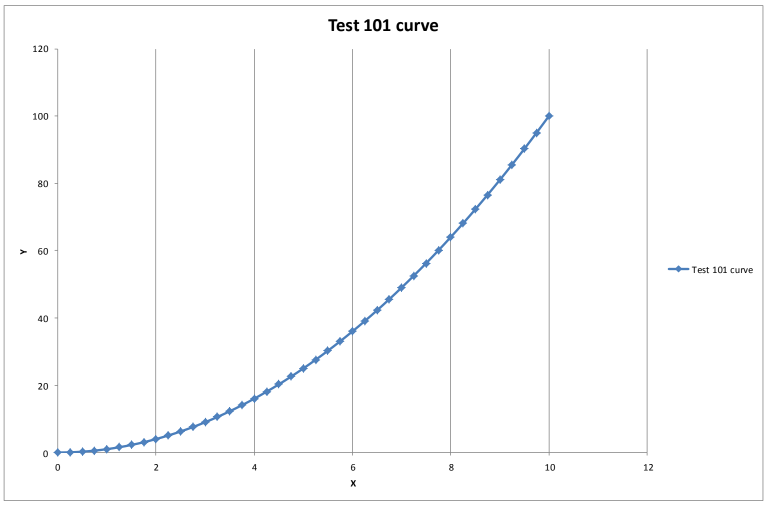


Figure 14.4 Single Curve

In the case of displaying multiple curves in an Excel sheet, we first define a number of functions and we then convert each of them to a vector (discrete function). Finally, we append each vector to a list and we call the appropriate driver function:

```
long N = 40;

// Create abscissa x array
double A = 0.0; double B = 3.0; // Interval
auto x = CreateMesh(N, A, B);

auto fun = [](double x) { return std::log(x+ 0.01); };
auto fun2 = [](double x) { return x*x; };
auto fun3= [](double x) { return x*x*x; };
auto fun4 = [](double x) { return std::exp(x); };
auto fun5 = [](double x) { return x; };

auto vec1 = CreateDiscreteFunction< std::vector<double>>(x, fun);
auto vec2 = CreateDiscreteFunction< std::vector<double>>(x, fun2);
auto vec3 = CreateDiscreteFunction< std::vector<double>>(x, fun3);
auto vec4 = CreateDiscreteFunction< std::vector<double>>(x, fun4);
auto vec5 = CreateDiscreteFunction< std::vector<double>>(x, fun5);
// Now Excel output in one sheet for comparison purposes

// Names of each vector
std::list<std::string> labels
{ "log(x+0.01)", "x^2", "x^3", "exp(x)","x" };

// The list of Y values using uniform initialisation
std::list<std::vector<double> > curves{ vec1, vec2, vec3, vec4, vec5 };

std::cout << "Data has been created\n";


ExcelDriver xl; xl.MakeVisible(true);
xl.CreateChart(x, labels, curves, "Comparing Functions", "x", "y");
```

The output is shown in Figure 14.5.
For completeness, we show the code to create vectors and matrices of values from a function:

```
template <typename Vector>
   Vector CreateDiscreteFunction(const std::vector<double>& x,
                                 const std::function<double(double)>& f)
{ // Create a discrete function from a continuous function y = f(x)
```

```
    Vector y(x.size());

    for (std::size_t j = 0; j < y.size(); ++j)
    {
       y[j] = f(x[j]);
    }

    return y;
}

template <typename Matrix>
    Matrix CreateDiscreteFunction2d(const std::vector<double>& x ,
                                     const std::vector<double>& y,
                                     const std::function<double (double x, double y)>& f)
{ // Create a discrete function from a continuous function m = f(x,y)

    std::size_t nr = x.size();
    std::size_t nc = y.size();

    Matrix m(nr,nc);

    for (std::size_t i = 0; i < nr; ++i)
    {
       for (std::size_t j = 0; j < nc; ++j)
       {
          m(i, j) = f(x[i],y[j]);
       }
    }

    return m;
}
```

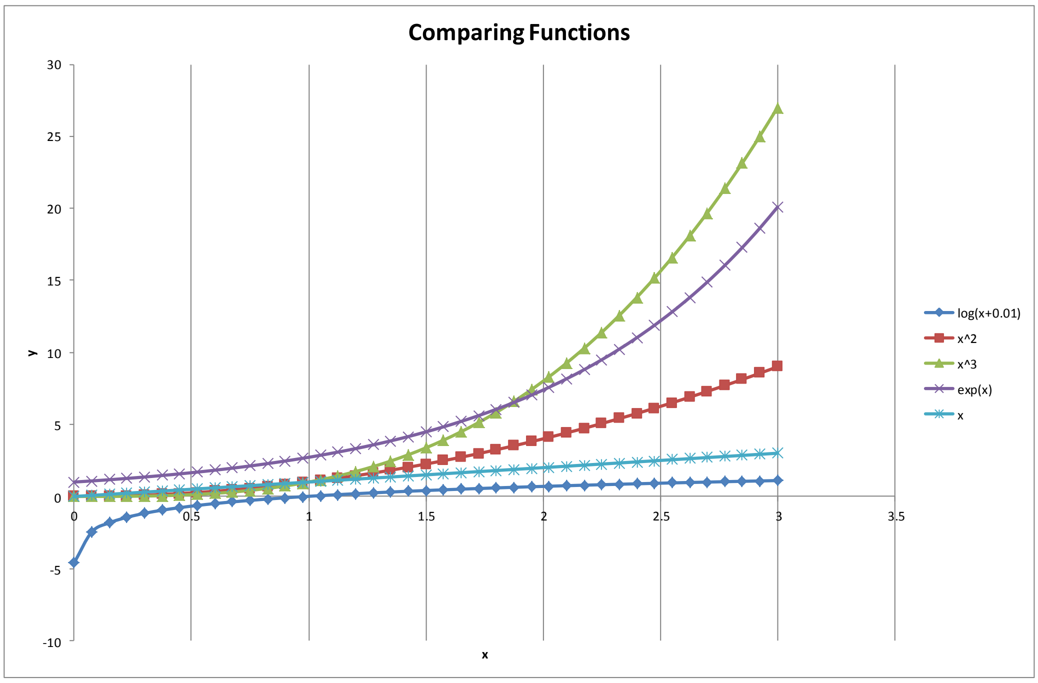


Figure 14.5 Multiple Curves

14.6 Path Generation for Stochastic Differential Equations (SDEs)

The generic examples in the preceding sections were chosen to show how the Excel driver works without becoming embroiled in software design decisions or mathematically advanced discussions. In this section we examine more complex examples that are related to computational finance. To this end, we simulate the paths of quantities that are defined by SDEs. In most cases analytical solutions do not exist and we must resort to numerical approximations, in this case the finite difference method.

We now discuss some schemes to calculate the path of the following nonlinear autonomous SDE:

$$dX(t) = \mu(X(t))dt + \sigma(X(t))dW(t) \quad 0 < t \le T$$
$$X(0) = A. \tag{14.1}$$

Some finite difference schemes are:

- *Explicit Euler*:

$$X_{n+1} = X_n + \mu_n \Delta t + \sigma_n \Delta W_n$$

$$\text{where} \tag{14.2}$$

$$\mu_n = \mu(X_n)\sigma_n = \sigma(X_n).$$

- *Semi-implicit Euler*:

$$X_{n+1} = X_n + [\alpha\mu_{n+1} + (1-\alpha)\mu_n]\Delta t + \sigma_n \Delta W_n$$

$$\text{with special cases} \tag{14.3}$$

$$\alpha = \tfrac{1}{2} \text{ (Trapezoidal)}, \alpha = 1 \text{ (Backward Euler)}.$$

- *Heun*:

$$X_{n+1} = X_n + \tfrac{1}{2}[F_1 + F_2]\Delta t + \tfrac{1}{2}[G_1 + G_2]\Delta W_n$$

where

$$F(x) \equiv \mu(x) - \tfrac{1}{2}\sigma^{'}(x)\sigma(x) \text{ where } \sigma^{'}(x) \equiv \tfrac{d\sigma}{dx} \tag{14.4}$$
$$F_1 = F(X_n), \quad G_1 = \sigma(X_n)$$
$$F_2 = F(X_n + F_1\Delta t + G_1\Delta W_n)$$
$$G_2 = \sigma(X_n + F_1\Delta t + G_1\Delta W_n).$$

- *Milstein*:

$$X_{n+1} = X_n + \mu_n\Delta t + \sigma_n\Delta W_n + \tfrac{1}{2}[\sigma^{'}\sigma]_n((\Delta W_n)^2 - \Delta t). \tag{14.5}$$

- *Derivative-free*:

$$X_{n+1} = X_n + F_1\Delta t + G_1\Delta W_n + [G_2 - G_1]\Delta t^{-1/2}\tfrac{(\Delta W_n)^2 - \Delta t}{2}$$

$$\text{where} \tag{14.6}$$

$$F_1 = \mu(X_n), \quad G_1 = \sigma(X_n)\ G_2 = \sigma(X_n + G_1\Delta t^{1/2}).$$

- *First-order Runge Kutta* with Ito coefficient (FRKI):

$$X_{n+1} = X_n + F_1\Delta t + G_2\Delta W_n + [G_2 - G_1]\Delta t^{1/2}$$

$$\text{where} \tag{14.7}$$

$$F_1 = \mu(X_n), \quad G_1 = \sigma(X_n)\ G_2 = \sigma(X_n + \tfrac{G_1(\Delta W_n - \Delta t^{1/2})}{2}).$$

In this section we extend the C++ code of section 6.8 in chapter 6 that models SDEs. In particular we simulate the paths of SDEs using the above finite difference methods. We design part of the system context diagram of Figure 7.1 (and related topics in chapter 9).

The main systems are:
- S1: The option (derivative) input data.
- S2: Modelling one-factor SDEs.
- S3: Modelling finite difference schemes.
- S4: Presenting SDE paths in Excel.

- S5: A Builder/Factory system to configure and initialise the systems S1 to S4.

We show how we have designed this system using modern C++ language features based on a multiparadigm approach. It is part of a larger system to compute one-factor option prices using the Monte Carlo method in chapter 31. We give a preview of upcoming design and we are able to display stochastic paths in a user-friendly manner.

14.6.1 The main Classes

The design of the current problem is shown in Figure 14.6 and it is a special case of the context diagram 7.1 and the more general diagrams in chapter 9. Furthermore, we have added a new data member to the SDE class that represents the upper limit of the time interval in which the SDE is defined:

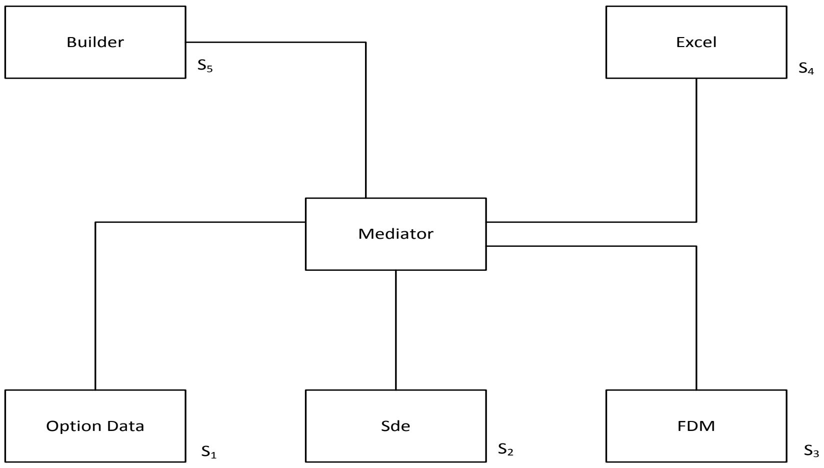


Figure 14.6 Context Diagram for Path Evolver

```
// Functions of arity 2 (two input arguments)
template <typename T>
    using FunctionType = std::function<T (const T& arg1, const T& arg2)>;


// Interface to simulate any SDE with drift/diffusion
// Use a tuple to aggregate two functions into what is similar to an
// interface in C# or Java.
template <typename T>
    using ISde = std::tuple<FunctionType<T>, FunctionType<T>>;

template <typename T = double> class Sde
{
private:
    FunctionType<T> dr_;
    FunctionType<T> diff_;

public:
    T ic; // Initial condition
    T B;  // SDE on interval [0,B]
public:
    Sde() = delete;
    Sde(const FunctionType<T>& drift, const FunctionType<T>& diffusion,
       const T& initialcCondition, const T& expiration)

       : dr_(drift), diff_(diffusion), ic(initialcCondition), B(expiration) {}

    Sde(const Sde<T>& sde2, const T& initialcCondition, const T& expiration)

       : dr_(sde2.dr_), diff_(sde2.diff_), ic(initialcCondition), B(expiration) {}

    Sde(const ISde<T>& functions, const T& initialcCondition, const T& expiration)

       : dr_(std::get<0>(functions)), diff_(std::get<1>(functions)),
             ic(initialcCondition), B(expiration) {}

    T drift(const T& S, const T& t) const { return dr_(S, t); }
    T diffusion(const T& S, const T& t) const { return diff_(S, t); }
};
```

We now discuss the design of the classes that model the finite difference schemes to approximate the solution of an SDE. We wish to write as little code as possible and the options open to us are:

- C1: Using dynamic/subtype polymorphism and class hierarchies (as in Duffy and Kienitz 2009).
- C2: Using *Curiously Recurring Template Pattern* (CRTP) and static polymorphism with class hierarchies.
- C3: Creating a single class with universal function wrappers as data members.

In this chapter we have chosen for option C2 for reasons of efficiency (all functions are known at compile-time) and maintainability (all data and functions are encapsulated in a set of related classes). In general, the FDM class needs to be acquainted with the SDE class. The FDM class models one-step time-marching schemes. Adhering to the *Single Responsibility Principle* (SRP) we say that such schemes advance the approximate solutions from a given time level n to time level $n+1$.

The base class is:

```
template <typename Derived, typename T> class Fdm
{ // Using CRTP pattern

//private:
protected:
    std::normal_distribution<T> dist;
    std::default_random_engine eng;
    std::shared_ptr<Sde<T>> sde;
public:
    Fdm(const std::shared_ptr<Sde<T>>& oneFactorProcess,
         const::normal_distribution<T>& normalDist,
         const::default_random_engine& engine)
    : sde(oneFactorProcess), dist(normalDist), eng(engine){}

    // Compute x(t_n + dt) in terms of x(t_n)
    T advance(T xn, double tn, double dt)
    {
       return static_cast<Derived*> (this)->advance(xn, tn, dt);
    }

    T generateRN()
    {
       return dist(eng);
    }
};
```

Notice that we have used the C++ `<random>` library's functions to generate uniform variates (this can be made more generic but not just yet). Specific finite difference schemes implement the function `advance()` in their own way, for example explicit Euler and Heun methods:

```
template <typename T>
    class FdmEuler : public Fdm<FdmEuler<T>, T>
{ // Using CRTP pattern

    private:

    public:
       FdmEuler(const std::shared_ptr<Sde<T>>& oneFactorProcess,
                const std::normal_distribution<T>& normalDist,
                const std::default_random_engine& engine)
             : Fdm(oneFactorProcess, normalDist, engine)
       {}

       // Compite x(t_n + dt) in terms of x(t_n)
       T advance(T xn, T tn, T dt)
       {
          T normalVar = generateRN();
          return xn + sde->drift(xn, tn)*dt
                      +sde->diffusion(xn, tn) * std::sqrt(dt) * normalVar;
       }
};
```

```
template <typename T>
    class FdmHeun : public Fdm<FdmHeun<T>, T>
{ // Using CRTP pattern

    private:

    public:
       FdmHeun(const std::shared_ptr<Sde<T>>& oneFactorProcess,
                const std::normal_distribution<T>& normalDist,
                const std::default_random_engine& engine)
             : Fdm(oneFactorProcess, normalDist, engine)
       {}

       // Compute x(t_n + dt) in terms of x(t_n)
       T advance(T xn, T tn, T dt)
       {
          T a = sde->drift(xn, tn);
          T b = sde->diffusion(xn, tn);
          T normalVar = generateRN();
          T suppValue = xn + a * dt + b * std::sqrt(dt) * normalVar;

          return xn + 0.5 * (sde->drift(suppValue, tn) + a) * dt
                     + 0.5 * (sde->diffusion(suppValue, tn) + b)
                     * std::sqrt(dt) * normalVar;
       }
    };
```

We have built a factory class that creates an SDE based on user choice:

```
namespace SdeFactory
{
    // Specific SDEs
    template <typename T>
       std::shared_ptr<Sde<T>> GbmSde(const OptionData& data, T S0)
    {
       T r = data.r; T sig = data.sig; T B = data.T;
       auto drift = [=](T t, T S) { return r * S; };
       auto diffusion = [=](T t, T S) { return sig * S; };

       ISde<T> Functions = std::make_tuple(drift, diffusion);

       // Create Sde
       return std::shared_ptr<Sde<T>>(new Sde<T>(Functions, S0, B));
    }

    template <typename T>
             std::shared_ptr<Sde<T>> CevSde(const OptionData& data, T S0)
    {
       T r = data.r; T sig = data.sig; T B = data.T;
       T beta = 0.5; // Hard-coded
       auto drift = [=](T t, T S) { return r * std::pow(S, beta); };
       auto diffusion = [=](T t, T S) { return sig * S; };

       ISde<T> Functions = std::make_tuple(drift, diffusion);

       // Create Sde
       return std::shared_ptr<Sde<T>>(new Sde<T>(Functions, S0, B));
    }

    template <typename T>
       std::shared_ptr<Sde<T>> CIRSde(T alpha, T b, T sigma, T S0, T B)
    {

       auto drift = [=](T t, T X) { return alpha*(b-X); };
       auto diffusion = [=](T t, T X) { return sigma * std::sqrt(X); };
       ISde<T> Functions = std::make_tuple(drift, diffusion);

       // Create Sde
       return std::shared_ptr<Sde<T>>(new Sde<T>(Functions, S0, B));
    }


    template <typename T>
       std::shared_ptr<Sde<T>> ChooseSde(const OptionData& data, T S0)
    { // Simple factory method
```

```
        std::cout << "1. GBM, 2. CEV, 3. CIR: ";
        int choice; std::cin >> choice;

        if (1 == choice)
        {
           return GbmSde<T>(data, S0);
        }
        if (2 == choice)
        {
           return CevSde<T>(data, S0);
        }

        if (3 == choice)
        {
           // Feller condition => 2 a b >= sigma^2
           double alpha = 0.01;
           double b = 7.0;
           double sigma = 0.9;
           double B = 1.0; // Upper limit of interval
           double X0 = 0.4;

           return CIRSde<T>(alpha, b, sigma, X0, B);

        }
        else
        {
           return GbmSde<T>(data, S0);

        }
     }

  }
```

We note that we have not created factories for the FDM classes in this version in order to avoid cognitive overload. In any case, it would have a structure similar to that in the above SDE case. Having created the SDE and FDM classes we now design the *mediator* class in Figure 14.6 that connects them and constructs the simulated path.

The code is:

```
  // Creating a path of an SDE approximated by FDM
  template <typename Derived, typename T>
     std::vector<T> Path(const Sde<T>& sde, Fdm<typename Derived, T>& fdm, long NT)
  {

     std::vector<T> result(NT + 1);

     result[0] = sde.ic;

     double dt = sde.B / static_cast<T>(NT);
     double tn = dt;

     for (std::size_t n = 1; n < result.size(); ++n)
     {
        result[n] = fdm.advance(result[n - 1], tn, dt);
        tn += dt;
     }

     return result;
  }
```

Finally, we encapsulate all relevant model constants in a convenient struct called `OptionData`:

```
  #include <algorithm> // for max()
  #include <boost/parameter.hpp>

  namespace OptionParams
  {
     BOOST_PARAMETER_KEYWORD(Tag, strike)
     BOOST_PARAMETER_KEYWORD(Tag, expiration)
     BOOST_PARAMETER_KEYWORD(Tag, interestRate)
     BOOST_PARAMETER_KEYWORD(Tag, volatility)
     BOOST_PARAMETER_KEYWORD(Tag, dividend)
```

```
    BOOST_PARAMETER_KEYWORD(Tag, optionType)
}


// Encapsulate all data in one place
struct OptionData
{ // Option data + behaviour

    double K;
    double T;
    double r;
    double sig;

    // Extra data
    double D;        // dividend

    int type;        // 1 == call, -1 == put

    explicit constexpr OptionData(double strike, double expiration,
                                  double interestRate,
                                  double volatility, double dividend, int PC)
       : K(strike), T(expiration), r(interestRate),
             sig(volatility), D(dividend), type(PC)
    {}

    template <typename ArgPack> OptionData(const ArgPack& args)
    {
       K = args[OptionParams::strike];
       T = args[OptionParams::expiration];
       r = args[OptionParams::interestRate];
       sig = args[OptionParams::volatility];
       D = args[OptionParams::dividend];
       type = args[OptionParams::optionType];

       std::cout << "K " << K << ", T " << T << ",r " << r << std::endl;
       std::cout << "vol " << sig << ", div " << D << ",type "
             << type << std::endl;
    }

};
```

We can initialise the elements of the struct `OptionData` by calling its constructor (position of args is important), by uniform initialisation or by using *named variables* as supported in the Boost *Parameter* library. In the latter case the relative order of the input arguments is immaterial as the following example shows:

```
//OptionData opt2 { 100.0, 1.0, 0.06, 0.2, 0.03, 1 };

// Init via named Boost variables
OptionData myOption((OptionParams::strike = 65.0,
                    OptionParams::expiration = 1.0,
                    OptionParams::volatility = 0.001, OptionParams::dividend = 0.0,
                    OptionParams::optionType = -1, OptionParams::interestRate = 0.1));
```

14.6.2 Testing the Design and Presentation in Excel

We now discuss the configuration and initialisation of the classes in Figure 14.6 The steps are easy to follow:

```
// A. Create basic input data
//OptionData(double strike, double expiration, double interestRate,
//  double volatility, double dividend, int PC)
OptionData myOption { 100.0, 10.0, 0.1, 0.9, 0.03, 1 };
double S0 = 20.0;

// B. Get the SDE
// Using factories
std::shared_ptr<Sde<double>> sde = SdeFactory::ChooseSde<double>(myOption, S0);

// C. Set up the FDM classes; 1 to N relationship between SDE and FDM
std::default_random_engine eng;

std::normal_distribution<double> nor(0.0, 1.0);
std::random_device rd;
eng.seed(rd());

FdmEuler<double> fdm(sde, nor, eng);
```

```
FdmHeun<double> fdm2(sde, nor, eng);


// D. Joining up SDE and FDM in Mediator class
long NT = 200;
auto vec = Path(*sde, fdm, NT);
auto vec2 = Path(*sde, fdm2, NT);


double A = 0.0; double B = myOption.T;
auto x = CreateMesh(NT+1, A, B);

// E. 'Report' system; display the garphs in Excel
ExcelDriver xl; xl.MakeVisible(true);

// E_1: Display each curve in its own sheet
xl.CreateChart(x, vec, "Euler method");
xl.CreateChart(x, vec2, "Heun method");

// Names of each vector
std::list<std::string> labels{ "Euler", "Heun"};

// The list of Y values
std::list<std::vector<double> > curves{ vec, vec2};

// E_2: Display all curves in one sheet
xl.CreateChart(x, labels, curves, "Comparing FDM ", "t", "S");
```

Typical output for the explicit Euler method is shown in Figure 14.7. In this case we simulate the short rate of the CIR model (square root diffusion process) and we see (and it is well-known) that the Euler method produces negative values which is incorrect. We discuss this issue in exercise 4.

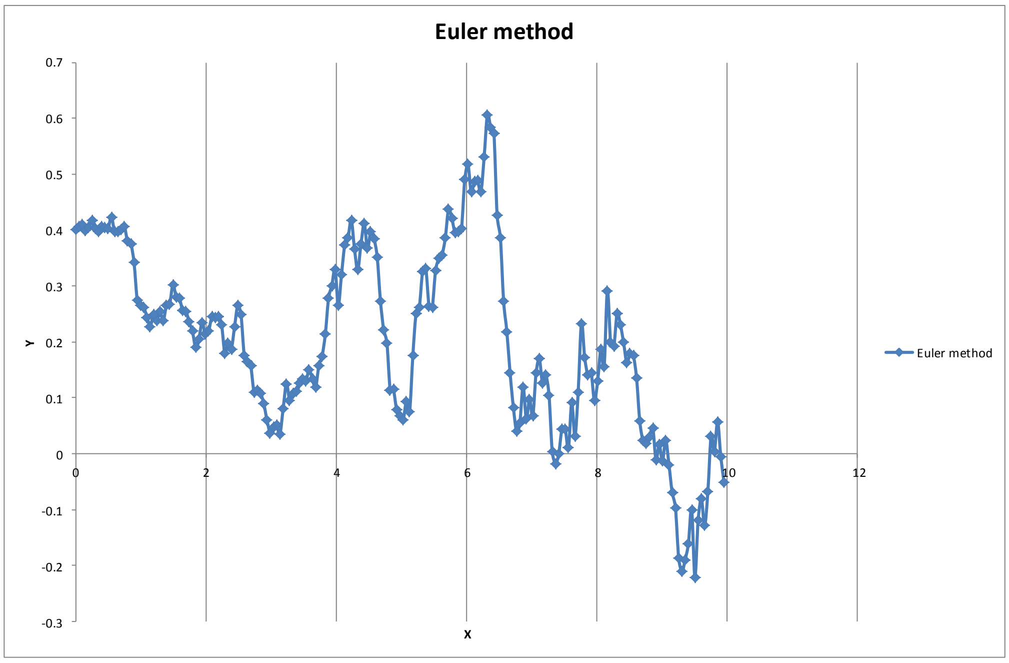


Figure 14.7 CIR path with Euler Method

Finally, we show the code that implements the mediator component from Figure 14.6. In this version it is a free function but in larger and more complex applications it could be modelled as a *Whole-Part object* (POSA 1996) to promote maintainability:

```
//  Creating a path of an SDE approximated by FDM
template <typename Derived, typename T>
    std::vector<T> Path(const Sde<T>& sde,
Fdm<typename Derived, T>& fdm, long NT)
```

```
{

    std::vector<T> result(NT + 1);

    result[0] = sde.ic;

    double dt = sde.B / static_cast<T>(NT);
    double tn = dt;

    for (std::size_t n = 1; n < result.size(); ++n)
    {
       result[n] = fdm.advance(result[n - 1], tn, dt);
       tn += dt;
    }

    return result;

}
```

14.7 Summary and Conclusions

In this chapter we introduced a software framework to display numeric data in Excel. We focused on visualising vectors, matrices and curves in the Excel spreadsheet program. We motivated the advantages and we gave a number of examples.

14.8 Exercises and Projects

1. (Extensions to the Excel Driver)

There are many ways to add new functionality to the Excel Driver class. In this exercise we focus on generalising the code in section 14.4. We wish to display functions of arity one and two. We hide low-level details such as the creation of mesh arrays and other supporting code. The desired function have the following interfaces:

```
using Function1DType = std::function<double (double x)>;
using Function2DType = std::function<double(double x, vdouble y)>;

void printDiscreteFunctionValues(const Function1DType& f,
double A, double B,long nSteps,
                                 const std::string& title,
                                 const std::string& horizontal,
                                 const std::string& vertical,
                                 const std::string& legend);

void printDiscreteFunctionValues(const Function1DType& f,
                                 const std::vector<double>& mesh,
                                 const std::string& title,
                                 const std::string& horizontal,
                                 const std::string& vertical,
                                 const std::string& legend);

void printDiscreteFunctionValues(const Function2DType& f,
double A, double B,long xSteps,
double C, double D,long ySteps,
                                 const std::string& title,
                                 const std::string& horizontal,
                                 const std::string& vertical,
                                 const std::string& legend);

void printDiscreteFunctionValues(const Function2DType& f,
                                 const std::vector<double>& mesh,
                                 const std::vector<double>& mesh2,
                                 const std::string& title,
                                 const std::string& horizontal,
                                 const std::string& vertical,
                                 const std::string& legend);
```

Answer the following questions:

a) Write the code for the above functions based on existing code in the driver.

b) Test functions of arity one on Black Scholes call option prices and the corresponding greeks:

$$\Delta_C = \frac{\partial C}{\partial S} = e^{(b-r)T} N(d_1)$$

$$\Gamma_C \equiv \frac{\partial^2 C}{\partial S^2} = \frac{\partial \Delta_C}{\partial S} = \frac{n(d_1) e^{(b-r)T}}{S\sigma\sqrt{T}}$$

$$Vega_C \equiv \frac{\partial C}{\partial \sigma} = S\sqrt{T} e^{(b-r)T} n(d_1)$$

$$\Theta_C \equiv -\frac{\partial C}{\partial T} = -\frac{S\sigma e^{(b-r)T} n(d_1)}{2\sqrt{T}} - (b-r) S e^{(b-r)T} N(d_1) - rKe^{-rT} N(d_2).$$

c) Test functions of arity two by displaying the probability density function of the *bivariate t-distribution* on a given interval (Nadarajah and Kotz 2005):

$$f(x,y;\nu,\rho) = \frac{1}{2\pi\sqrt{1-\rho^2}} \left\{ 1 + \frac{x^2 - 2\rho xy + y^2}{\nu(1-\rho^2)} \right\}^{-(\nu+2)/2}.$$

We shall discuss this distribution in more detail in chapter 16 when we discuss ways to compute the (cumulative) probability integral:

$$P(x,y;\nu,\rho) = \int_{-\infty}^{x} \int_{-\infty}^{y} f(x,y;\nu,\rho) dx dy.$$

2. (Finite Difference Schemes for the one-dimensional Heat Equation)
An interesting and useful application of the Excel driver is in displaying the values of the approximate solutions (using FDM in this case) to the heat equation as we discussed in chapter 13. In particular, I have found it useful for the following cases:

- C1: Having written FD code, we wish to see if it is producing correct results. To this end, displaying the results in an easy format is a boon.
- C2: We would like to compare the accuracy of the ADE family of schemes with a baseline case such as the Crank-Nicolson method or an analytical solution of a PDE.
- C3: Display the accuracy of an approximate method as a function of some independent variable. An example is plotting option price in the binomial method as a function of the number of steps. The same exercise is applicable to the Finite Difference Method.

Answer the following questions:

a) Use the two ADE schemes that we introduced in chapter 13 to approximate the solution of the heat equation (see equation 13.14 In chapter 13).
b) Compare the solutions in part a) with the exact solution and the solution produced by the Crank-Nicolson method.
c) Display the four results from parts a) and b) as four curves in a single Excel sheet.
d) Experiment with various initial conditions in equation 13.14 and repeat steps a), b) and c).

3. (Cubic Spline Overshoots, Part II)
In chapter 13 we gave a detailed introduction to cubic splines and some of their applications. In particular, exercise 5 in chapter 13 was concerned with a discussion using cubic splines and possible overshoots than can occur with yield curve data. Answer the following questions:

a) Use the same data as in exercise 13.5 and display the interpolated cubic spline values in Excel.
b) Compare the results with those produced by linear interpolation (see equation 13.32).
c) Display the first and second derivatives of the cubic spline from part a) based on the formulae (13.38) and (13.37), respectivily.

4. (The Cox-Ingersoll-Ross Model (CIR))

We examine the Cox-Ingersoll-Ross interest model (also known as *square-root diffusion*) defined the SDE with constant coefficients (Glasserman 2004):

$$dr = a(b-r)dt + \sigma\sqrt{r}dW.$$

This is a model of the short rate and it is generally referred to as the CIR model. In this case the short rate $r(t)$ is pulled towards b at a speed controlled by a. Furthermore, if $r(0) > 0$ then $r(t)$ will never be negative and $r(t)$ remains strictly positive for all t if the following *Feller condition* is satisfied:

$$ab > \frac{1}{2}\sigma^2.$$

The objective of this exercise is to introduce this model into the framework in Figure 14.6 and gain insight into its analytical and numerical properties.

Answer the following questions:

a) Create a class for the CIR SDE and integrate it into the framework.
b) The CIR SDE does not have an explicit solution. We then resort to numerical methods. Apply the explicit Euler and Heun methods for various time-step sizes and expirations.
c) Investigate the finite difference schemes when the Feller condition is not satisfied.
d) Experiment with the finite difference schemes and determine positivity for small and large values of drift, volatility and expiration parameters.

We remark that the *transition density* for the CIR process is known and it can be represented in terms of a noncentral chi-squared distribution (Glasserman 2004 page 122).

5. (SDEs with Analytical Solutions)
In some cases (we might get lucky) it is possible to find analytical solutions to SDEs (Kloeden and Platen 1995, pages 118-126). Some examples of SDEs and their explicit solutions are:

$$\begin{aligned}&dX = \tfrac{1}{2}a^2X dt + aX dW\\&X(t) = X_0 \exp(aW(t)).\end{aligned}$$

$$\begin{aligned}&dX = aX dt + bX dW\\&X(t) = X_0 \exp((a - \tfrac{1}{2}b^2)t + bW(t)).\end{aligned}$$

$$\begin{aligned}&dX = \tfrac{1}{2}X dt + X dW\\&X(t) = X_0 \exp(W(t)).\end{aligned}$$

Here $W(t)$ is a Wiener process.

Answer the following questions:

a) Integrate these equations into the framework and decide how to determine the difference between the explicit solution and the solution defined by a given finite difference scheme.
b) Experiment with the schemes by varying the number of time steps. Is there some way to determine what the order of convergence of a given scheme is?
c) We focus on the explicit Euler method. Produce a multi-curve graph in the style of Figure 14.5 in which each curve is a path produced by the Euler method with a given number of time steps, for example step sizes NT = 10, 20, 50, 100, 200, 300. Do you see monotonic convergence to the 'exact path' as NT increases?

6. (Log-log and Semi-log Plots)
In some cases we may wish to transform data in some way before presenting it in Excel. In general, the *linear plot* between the horizontal x and vertical y axes can be transformed in two main ways:

- *Log-log plot*: uses a logarithmic scale for both the horizontal and vertical axes. This transformation is suitable for *monomials* (relationships of the form $y = ax^k$). These become straight lines in log space as the following steps show:

$$\begin{cases} y = ax^k \\ \log y = k \log x + \log a \\ X = \log x, Y = \log y \\ \\ Y = mX + b. \end{cases}$$

We thus see that $m = k$ is the slope (gradient) of the straight line while $b = \log a$ is the intercept on the $\log y$ axis. We thus can see that this graph can recognise relationships and estimate parameters, in this case the parameters a and k.

- *Semi-log plot*: In this case only one of the variables is log transformed. We distinguish between *log-lin* in which the y axis is transformed and *lin-log* in which the x axis is transformed. Equations of the form $y = \lambda a^{\gamma x}$ become straight lines when plotted semi-logarithmically as the following steps show:

$$\log y = \log(\lambda a^{\gamma x}) = \log \lambda + \log(a^{\gamma x}) = \log \lambda + (\gamma \log a)x.$$

In general, any log base can be used but most applications use base $10, 2$ or e.

Answer the following questions:

a) Determine how to introduce log transformations into the current Excel driver framework.
b) As test 101 case, we take the example $y = x^2/2$. Generate x and y array values on the interval [1,10]. Then perform a log-log transform on the arrays and plot the new arrays in Excel. Can you reproduce or 'see' the original parameters from the slope and intercept of the straight line?

We shall see more examples in later chapters, for example when determining the order of convergence of finite difference schemes.

7. (Extending the Framework in Figure 14.6, Brainstorming)
The solution to this exercise is discussed in chapter 31 but we it does no harm to address the problem as early as possible in the book. The focus is in extending the design in Figure 14.6 to create a small one-factor Monte Carlo option pricer.

Answer the following questions:

a) Consider the initial case of pricing one-factor call and put option prices. Which new systems and classes need to be added to the framework? A requirement is that the framework computes option price and the standard error.
b) The mediator class needs to be extended because in its current form it computes a single path only. Using the Monte Carlo method entails that we create a large number of paths and that we use well-known averaging and discounting algorithms.
c) Test the code from parts a) and b) for call and put options and compare exact and approximate solutions with each other.
d) Extend the framework so that it can be configured to simultaneously price several options for the same path information.

14.9 Appendix: COM Architecture Overview COM Architecture Overview

In this section we give a global overview of the *Component Object Model* (COM). You may skip section 14.9, 14.10 and 14.11 without loss of continuity although you should read sections 14.12 and 14.13. The two key entities in COM are *interfaces* and *components*. In this section we explain what they are from a conceptual viewpoint and we then discuss how COM realises them. An *interface* is a pure specification of intended behaviour. In other words, it consists of a set of abstract methods. Each method is thus devoid of implementation and an interface may not contain non-abstract methods nor may it contain member data. This description is consistent with how C# and Java support interfaces. C++ does not support interfaces as such but they can be emulated by classes having no member data and consisting solely of pure virtual functions. COM has a keyword called `interface` that is defined as follows:

```
#define interface struct
```

which is defined in the file `objbase.h` header included in the Microsoft Win32 *Software Development Kit* (SDK). We take an initial example in C++ that emulates COM functionality. To this end, we define two interfaces that contain methods for presenting data in different ways. We then use multiple inheritance to create a class that implements both interfaces. The complete code is:

```
#include <iostream>
#include <objbase.h>               // Define interface.


// Abstract interfaces
interface IDraw
{ // Present information

   virtual void __stdcall display() = 0;
   virtual void __stdcall print() = 0;
};

interface ISave
{ // Save to permanent storage

   virtual void __stdcall saveXML() = 0;
   virtual void __stdcall saveDB() = 0;
};

// Interface implementation
class CA : public IDraw, public ISave
{
public:

   // Implement interface IDraw.
   virtual void __stdcall display() {std::cout << "CA::display" << std::endl;}
   virtual void __stdcall print() {std::cout << "CA::print" << std::endl;}

   // Implement interface ISave.
   virtual void __stdcall saveXML() {std::cout << "CA::saveXML" << std::endl;}
   virtual void __stdcall saveDB() {std::cout << "CA::saveDB" << std::endl;}
};
```

We first note that this code is still C++ and not COM code. Second, this code contains the Microsoft-specific extension to the compiler called `__stdcall`. This is a somewhat technical issue and it is concerned with method arguments, how they are pushed onto the stack and who (whether it is the *caller* or *callee*) removes them from the stack. The general options are:

- *cdecl*: arguments are pushed onto the stack in reverse order, that is from right to left and the caller (calling function) cleans up the stack. This is the C language convention.
- *pascal*: arguments are pushed onto the stack from left to right and the callee cleans up the stack. This was the calling convention in the Win16 operating system.
- *stdcall*: this is the standard and most common calling convention. It is a variation of the pascal calling convention in that the callee cleans the stack but the parameters are pushed onto the stack from right to left. It is the convention used by the application programming interfaces in the Win32 operating system.

A simple test program based on the above code is:

```
// Client
int main()
{
   CA* pA = new CA ;

   // Get an IDraw pointer.
   IDraw* pd = pA ;

   std::cout << "Client: Use the IDraw interface.";
   pd->display() ;
```

```
    pd->print() ;

    // Get an ISave pointer.
    ISave* py = pA;

    std::cout << "Client: Use the ISave interface.";
    py->saveXML() ;
    py->saveDB() ;

    delete pA ;

    return 0 ;
}
```

In general, each method in an interface can have input, output and input/output parameters as well as a return type. In the above simple example the methods had no input parameters and they all had a `void` return type.

14.9.1 COM Interfaces and COM Objects

We now discuss two important concepts in COM, namely COM interfaces and COM objects. COM interfaces are special in a number of ways. We need to discuss a number of issues which will help us achieve a good understanding of how to define, discover and use these interfaces and objects.

Each system that supports COM must include an implementation of the COM library. The library contains groups of functions that provide basic services to objects and to their clients. For example, it contains functionality for clients to start an object's server. The COM library's services are accessed through ordinary function calls.
An interface is essentially a contract between an object that implements the interface and clients of the object. In order to make the contract work between the object and client we must address the following issues:

- A1: a way to explicitly identify an interface.
- A2: how to describe (define) the methods in an interface.
- A3: how to actually implement an interface.

Regarding activity A1, there are two ways to identify an interface. The first way is to identify the interface by a string name which humans find easy to read while the second way uses a 16 byte (128 bit) *Globally Unique Identifier* (GUID) to identify the interface in software. GUIDs are guaranteed to be unique in space and time and the chance is extremely small that two interfaces (or other entities in COM) will be assigned the same value. In the case of interfaces the unique identifier is called an *interface identifier* (IID). The next challenge is to have some global clearinghouse that keeps track of all names (including interface identifiers). This clearinghouse is called the *Registry* and it is the shared system database for the Windows operating system. It contains information about system hardware, software, configuration and users.
A COM component maintains a number in its state called the *reference count*. This count is incremented when a client gets an interface and it is decremented when the client is finished using the interface. Each interface must support the `IUnknown` interface which has three methods:

- `QueryInterface`: the client can call this method to discover whether a component supports a particular interface. A pointer to the interface will be returned if the component supports the interface.
- `AddRef`: increments the reference count on the interface.
- `Release`: releases a reference to the interface.

In other words, the developer must implement `IUnknown` and its reference counting mechanism is similar to how both C++ 11 and how the Boost Smart Pointer library automatically manage the lifetime of heap-based objects. The precise specification for `IUnkown` is:

```
interface IUnknown
{
    virtual HRESULT QueryInterface( [in] IID& iid, [out] void** ppv) =0;
    virtual ULONG AddRef(void) =0;
    virtual ULONG Release(void) =0;
};
```

We see that `QueryInterface` has two arguments; the first input parameter `iid` is the identifier for an interface that we wish to determine whether it is supported and the second argument `ppv` that is set to `NULL` before the call to `QueryInterface`. If the call succeeds we get a pointer to `ppv` and we can then call its methods. Notice that we can specify parameters as input or output parameters.

14.9.2 **`HRESULT` and other Data Types**

Components use the 32-bit field `HRESULT` to communicate with clients. Its structure is shown in Figure 14.8. For example, it can signal success or failure when a method is called. Most COM functions return this integer as return type.

The possible return types are:

- `E_ABORT`: the operation was aborted because of an unspecified error.
- `E_ACCESSDENIED`: a general access-denied error.
- `E_FAIL`: an unspecified failure has occurred.
- `E_HANDLE`: an invalid handle was used.
- `E_INVALIDARG`: one or more arguments are invalid.
- `E_NOINTERFACE`: the `QueryInterface` method of the interface `IUnknown` did not recognise the requested interface. The interface is not supported.
- `E_NOTIMPL`: the method is not implemented.
- `E_OUTOFMEMORY`: the method failed to allocate memory.
- `E_PENDING`: the data necessary to complete the operation is not yet available.
- `E_POINTER`: an invalid pointer was used.
- `E_UNEXPECTED`: a catastrophic failure occurred.
- `S_FALSE`: the method succeeded and returned the boolean value `FALSE`.
- `S_OK`: the method succeeded. If a Boolean return value is expected, the returned value is `TRUE`.

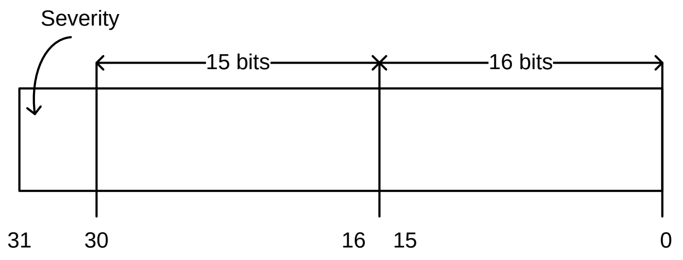


Figure14.8 Format of `HRESULT`

We can check the return type in code by calling the `SUCCEEDED` and `FAILED` macros as the following examples show:

```
HRESULT hr = SetStrings ();
if (FAILED (hr))
    return hr;

HRESULT hr2 = PropBag->Read();
if (SUCCEEDED(hr2))
{
    // code
}
```

It is advisable to use these macros when testing a method's return value instead of comparing the result `hr` against the raw values; however, it is allowed to use these values as return types in code, for example:

```
HRESULT FinalConstruct()
{
    return S_OK;
}
```

14.9.3 **Interface Definition Language (IDL)**

COM supports a tool that allows us to define interfaces in a standard way. It is called *Interface Definition Language* (IDL) and it is an extension of the IDL in Microsoft's *Remote Procedure Call* (RPC) which is in turn based on the IDL in the Open Software Foundation's *Distributed Computing Environment* (OSF DCE). IDL describes a COM object's interfaces in an unambiguous manner. It allows us to associate the methods of an interface with its IID. We can also specify the details of the interface in a form that can be machine-processed to produce marshalling code when we create *dynamic link libraries* (DLLs).

```
// Interface IDraw
[
   object,                                      // 1.
   uuid(D6FCCE1F-8A2D-4303-A974-09AEC422EFFE),  // 2.
   helpstring("My drawing interface"),          // 3.
   pointer_default(unique)                      // 4.
]

interface IDraw : IUnknown
{
   HRESULT draw();
};
```

14.9.4 Class Identifiers (CLSID)

A COM object is an instance of some class. Each class is assigned a GUID called a *class identifier* (CLSID). A client can pass a CLSID to the COM library to create an instance of the class. However, this is not mandatory. The primary use of a CLSID is to identify a specific piece of code for the COM library to load and execute for instances of the class corresponding to the CLSID.

14.10 An Example

We give a simple example to show how to define both COM interfaces and COM objects that use these interfaces. We focus on the language issues that we have discussed. In this case we define interfaces and a class that implements one of them. The interface definitions are:

```
// Interfaces
interface IDraw : IUnknown
{
      virtual void __stdcall display() = 0 ;
};

interface ISave : IUnknown
{
      virtual void __stdcall save() = 0 ;
};
```

The corresponding IDs are:

```
// IIDs
//
// {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
static const IID IID_IDraw =
      {0x32bb8320, 0xb41b, 0x11cf,
      {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};

// {32bb8321-b41b-11cf-a6bb-0080c7b2d682}
static const IID IID_ISave =
      {0x32bb8321, 0xb41b, 0x11cf,
      {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
```

We now create class called `MyClass` that implements `IDraw` that we defined in section 14.9; notice that it must also implement the methods of `IUnknown`:

```
//
// Component
//
class MyClass : public IDraw
{

public:
   // IUnknown implementation
   virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv) ;

   virtual ULONG __stdcall AddRef() ;
   virtual ULONG __stdcall Release() ;
```

```
    // Interface IDraw implementation
    virtual void __stdcall display() { cout << "display" << endl ;}

    // Constructor
    MyClass() : m_cRef(0) {}

    // Destructor
    ~MyClass() { std::cout << "Bye, bye\n";}

private:
    long m_cRef;
};
```

The implementation of this component is:

```
HRESULT __stdcall MyClass::QueryInterface(const IID& iid, void** ppv)
{
    if (iid == IID_IUnknown)
    {
       std::cout << "MyClass QI: Return pointer to IUnknown.\n";
       *ppv = static_cast<IDraw*>(this) ;
    }
    else if (iid == IID_IDraw)
    {
       std::cout << "MyClass QI: Return pointer to IDraw.\n";
       *ppv = static_cast<IDraw*>(this) ;
    }
    else
    {
       std::cout <<"MyClass QI: Interface not supported.\n";
       *ppv = NULL ;
       return E_NOINTERFACE;
    }
    reinterpret_cast<IUnknown*>(*ppv)->AddRef() ;
    return S_OK ;
}

ULONG __stdcall MyClass::AddRef()
{
    // Use InterlockedIncrement to ensure thread-safeness
    return InterlockedIncrement(&m_cRef) ;
}

ULONG __stdcall MyClass::Release()
{

    // Use InterlockedDecrement to ensure thread-safeness
    if (InterlockedDecrement(&m_cRef) == 0)
    {
       delete this ;
       return 0 ;
    }
    return m_cRef ;
}

//
// Creation function
//
IUnknown* CreateInstance()
{ // Factory method pattern in the form of a function

    IUnknown* pI = static_cast<IDraw*>(new MyClass) ;
    pI->AddRef() ;
    return pI ;
}
```

A test program to show how to use the component is:

```
int main()
{
    HRESULT hr ;

    IUnknown* pIUnknown = CreateInstance() ;

    std::cout << "Client: Get interface IDraw.";
```

```
      IDraw* pIDraw = NULL ;
      hr = pIUnknown->QueryInterface(IID_IDraw, (void**)&pIDraw) ;

      if (SUCCEEDED(hr))
      {
         std::cout <<"Client: Succeeded getting IDraw.";
         pIDraw->display() ;          // Use interface IDraw.
         pIDraw->Release() ;
      }


      std::cout <<
         "Client: Get interface ISave which MyClass does _not_ support.\n";

      ISave* pISave = NULL ;
      hr = pIUnknown->QueryInterface(IID_ISave, (void**)&pISave) ;
      if (SUCCEEDED(hr))
      {
         std::cout << "Client: Succeeded getting ISave.";
         pISave->save() ;             // Use interface ISave.
         pISave->Release() ;
      }


      std::cout << "Client: Release IUnknown interface.";
      pIUnknown->Release() ;

      return 0;
   }
```

The output from this code is:

```
Client: Get interface IDraw.MyClass QI: Return pointer to IDraw.
Client: Succeeded getting IDraw.display
Client: Get interface ISave which MyClass does _not_ support.
MyClass QI: Interface not supported.
Client: Release IUnknown interface.Bye, bye
```

14.11 Virtual Function Tables

We now discuss how interfaces work in COM and we discuss some of the consequences of the related design choices that have been made in COM. In particular, COM interfaces are implemented using pure abstract base classes, that is classes consisting solely of pure virtual member functions and having no member data. An important point to note is that such classes define the specific memory structure that COM requires for an interface. In fact, we are defining the layout of a block of memory.
Let us take an example of an interface that models functions to price financial derivatives and calculate their sensitivities (such as delta and gamma):

```
interface IOption
{ // Functions related to option pricing

   virtual void __stdcall price(double* myPrice) = 0;
   virtual void __stdcall delta(double* myDelta) = 0;
   virtual void __stdcall gamma(double* myGamma) = 0;
};
```

As is usual with C++, we create a class that implements these pure virtual member functions (for convenience, we do not yet implement the full Black Scholes formula):

```
class Option : public IOption
{ // All values are simulated, used for motivational purposes

public:
   virtual void __stdcall price(double* myPrice) { *myPrice = 1.0;}
   virtual void __stdcall delta(double* myDelta) { *myDelta = 0.5;}
   virtual void __stdcall gamma(double* myGamma) { *myGamma = 0.0;}
};
```

A simple usage is given by:

```
// Option class
IOption* opt = new Option;
double value;
```

```
  opt->price(&value);
  std::cout << value << std::endl;
  delete opt;
```

All implementations (for example, class `Option`) are blocks of memory having the same basic layout as shown in Figure 14.9. The abstract class defines the memory structure and it performs no memory allocation, this being done by derived classes of the abstract class. The *virtual function table* is an array of pointers that point to the implementations of the pure virtual functions. The first entry in the *virtual table* (vtabl) contains the address of the function `price` as it is implemented in derived classes. The second entry in the table is the address of the function `delta` and so on. In general, the layout for a COM interface is the same as the memory layout that the C++ compiler generates for an abstract base class. Thus, abstract classes can be used to define COM interfaces.

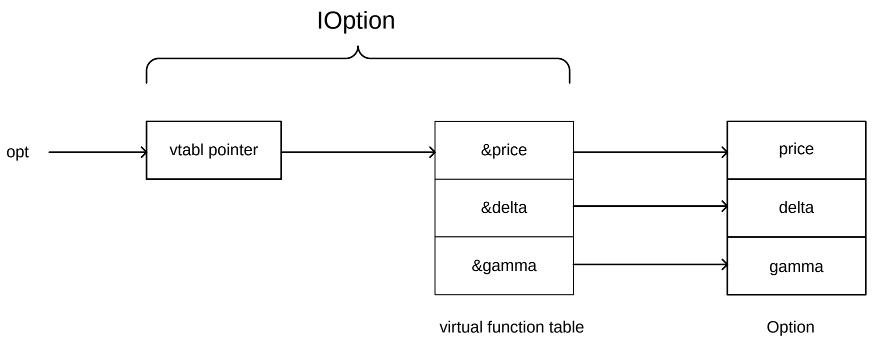


Figure 14.9 Memory layout

We now discuss how memory layout works in the case of the class `CA` from section 14.9. that implements multiple interfaces. We note that these interfaces are derived from `IUnknown`. The layout is shown in Figure 14.10.

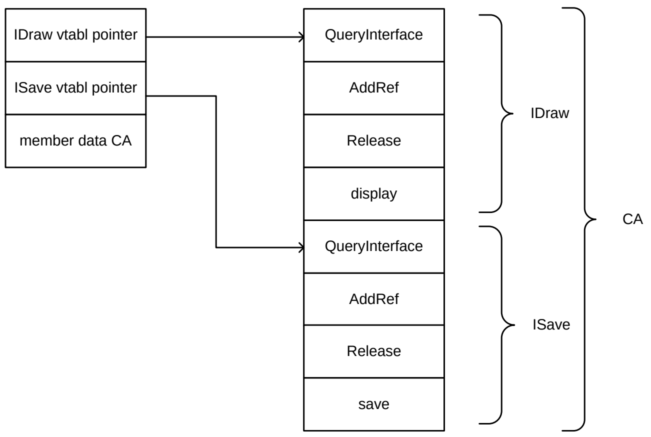


Figure 14.10 Memory layout of a class with multiple inheritance

One final remark; once a client gets an interface pointer from a component then the only thing connecting the client and component is the binary layout of the interface. In this sense the client is in fact asking for a chunk of memory when it requests an interface to a component.

14.12 Differences between COM and Object-Oriented Paradigm

We conclude this chapter with a general discussion on the differences between the COM way of thinking and the C++ object-oriented model. First, COM relies solely on *interface inheritance* which means that classes can only be derived from interfaces. This is in contrast to C++ *implementation inheritance* in which a derived class can inherit functionality and data from one or more base classes. This latter approach can and does lead to maintenance problems, sometimes called the *fragile base class problem* by which we mean that changes in the base class can have knock-on (and potentially dangerous) effects on the integrity of derived classes.

Implementation inheritance corresponds to the ISA semantic relationship between structural entities (such as classes) while the term interface inheritance is somewhat of a misnomer because neither functionality nor data is inherited. It is more accurate to say that a class *implements an interface*.

Finally, we give some more differences:

a) A class is not a component. It is possible to implement a single COM component using several C++ classes and it is even possible to implement a COM component without using any C++ classes at all.
b) It is not always necessary to inherit interfaces. It is possible to implement an interface in a class and then use this class in client code.
c) COM uses multiple inheritance to allow components to support multiple interfaces.
d) Interfaces in COM never change. If you wish to modify the methods in an interface or even remove or add a method then you should not change the interface but instead create a *new version* of the interface and leave the old one alone.
e) COM supports polymorphism because of the ability to switch between multiple interfaces. This feature improves application reusability.

14.13 Initialising the COM Library

Finally, in order to initialise the COM library we must call `CoInitialize` before any of the functions in COM can be used. When we no longer need the COM library we must call `CoUninitialize`.
The COM library is initialised only once per process. In-process components do not need to initialise the COM library. These components do not change the fundamental structure of a Windows program.
Calling `CoInitialize` multiple times in a process is allowed as long as each `CoInitialize` has a corresponding `CoUninitialize`.