The Boost C++ Libraries: Part II

Daniel J. Duffy
Datasim Education BV, e-mail: dduffy@datasim.nl

Abstract
In this article we discuss some of the Boost libraries that were introduced in Part I. The goal of Part I was to categorise and give a global overview of the most important Boost libraries. It was written to give some directions on the usefulness of the libraries in a given context. The goal of the current article is to focus on a number of libraries that in our opinion help promote the flexibility, functionality and performance of C++ applications. We introduce these libraries and we give some examples to illustrate their applicability.

Keywords
Boost, multi-paradigm programming, C++, option pricing, software design

1 Some Views on Software Development
We begin our discussion with remarks on some of the technical problems that confront developers when they wish to implement flexible and maintainable software systems. The discussion is highly simplified and somewhat subjective but it does discuss a number of high-level problems that most developers (and project managers) find challenging. Some of the major problems are:

- *Problem 1*: Modern object-oriented software systems tend to be implemented in a bottom-up manner. The resulting object network consists of tightly-coupled modules that are very difficult to modify and to extend. Furthermore, it is usually very difficult to use a subset of the functionality precisely because each module is directly or indirectly connected to every other module. In graph theory, we can say that the object network has approximately one *connected component*.
- *Problem 2*: C++ — in contrast to Fortran — has relatively few de-facto standard libraries and supporting software tools for scientific, financial and engineering applications. This state of affairs has persisted well into the twenty-first century.
- *Problem 3*: C++ is a *multi-paradigm language* but many developers still tend to create applications based on large and deep class hierarchies that become unstable as more functionality is added to them. Developers seem to be less comfortable with the generic programming model that C++ supports using templates. Anecdotal evidence suggests that fifty percent of quant developers are familiar with C++ but only twelve percent use the containers and algorithms from the Standard Library.
- *Problem 4*: C++ is a systems programming language. Of course, it is used to develop application software but the maintainability costs are relatively high. In particular, the fact that C++ does not support *interfaces* as first-class objects severely restricts our ability to design and implement loosely-coupled software based on system decomposition techniques.

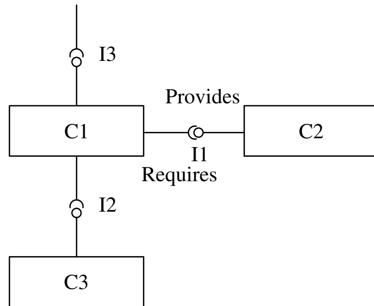
We now address some of these problems and we attempt to resolve them by using C++ in combination with design patterns and appropriate Boost libraries.

2 C++ Status Report: Multi-Paradigm Programming Language
C++ is a popular language and it allows developers to create flexible applications by encapsulating domain entities in classes. Furthermore, we can create complex classes from simpler ones using the *Composition, Aggregation* and *Inheritance* mechanisms. In general, these are client-server relationships between one class (the *client*) that uses the services of one or more other classes (called *server* classes) by calling their member functions. This situation leads to an *Object Connection Architecture* (OCA) because all inter-module connections are from object to object. The major disadvantage is that all modules and classes must be built before the architecture is defined and hence this approach cannot be used to lay out the plan for a software system. This is in contrast to an *Interface Connection Architecture* (ICA) that defines all connections between components of a system using only the interfaces. Interfaces need to specify both *provided* and *required* features. In general, a feature is a computational element of a component, for example a function, port or action. For a detailed introduction to object and interface connection architectures, see Leavens 2000.

The crucial issue is to implement provides and requires interfaces in C++. To this end, we use the Boost Function library to implement provides interfaces and the Boost Signals library to implement requires interfaces. Before we go into the details we describe these interfaces using standard UML *component diagrams*, an example of which is shown in Figure 1. In this case component C1 provides the interface I3 to potential clients and it has the requires interfaces I1 and I2 from server components C2 and C3, respectively. In other words, C1 offers services but it also requires services from other server components.

We now give a simple example of a class that implements the price function for the Black Scholes formula and that shows how to implement provides and requires interfaces. This class also requires the data from another interface that we implement as a Boost function. This latter entity is responsible for producing the actual data that is needed by the pricing formula. We now describe the C++ setup in detail. The main objective is to show how to price a call option by implementing the features that we see in Figure 1. In particular the client class `Pricer3` provides an interface to compute the option price. It communicates with an object that is responsible for creating the data that is used by the pricing formula. The client has no knowledge of

Figure 1: UML Component Diagram.



the precise implementation of the data source object; this decision has been delayed and it for other parts of the software system to implement it:

```
// One version of an implementation of ICA
// A class that offers an interface and
// requires another interface
struct Pricer3
{

  // Embedded notifier
  boost::signal<void (Data& data)> mySignal;

  // Entity that creates data
  boost::function<void (Data& data)> slot;

   Pricer3(boost::function<void (Data& data)> source)
       : mySignal(boost::signal<void (Data& data)> ())
   {
       slot = source;
       mySignal.connect(slot);
   }

   double compute(double S)
   {
       // Define the data and slot
       Data data;

       // Connect to slot and initialise the data
       mySignal(data);

       double tmp = data.sig * sqrt(data.T);

       double d1 = ( log(S/data.K)
              + (data.b+ (data.sig*data.sig)*0.5 ) *
                     data.T )/ tmp;
       double d2 = d1 - tmp;

       return (S * exp((data.b-data.r)*data.T) * N(d1))
              - (data.K * exp(-data.r * data.T)* N(d2));
   }

   ~Pricer3()
   {
       mySignal.disconnect(&slot);
   }
};
```

We now customise this class in different ways. In particular, we can implement the data source as a global C function, a function object, a class' member function or even a lambda function using the features in the new C++ standard. In general, this level of customisability is realised by the appropriate use of the Boost libraries that support higher order functions. For example, we can implement the data source as a simple global function and as a function object which can be customised for various kinds of options, as the following code shows:

```
struct Data
{ // Option data

       double T;
       double K;
       double r;
       double sig;
       double b; // Cost of carry
};

void PlainDataSource(Data& val)
{ // Simple data source; standard stock

       val.T = 0.25;
       val.K = 65.0;
       val.r = 0.08;
       val.sig = 0.3;
       val.b = val.r;

}

enum OptionType {Stock, Index, Future};

struct GeneralisedDataSource
{ // Allows for different kinds of options;
  // this is a function object

   OptionType optType;

   GeneralisedDataSource() : optType(Stock) {}

   GeneralisedDataSource(OptionType optionType) :
       optType(optionType) {}
   void operator () (Data& val)
   {
       val.T = 0.25;
       val.K = 65.0;
       val.r = 0.08;
       val.sig = 0.3;
       val.b = val.r;
```

```
        if (optType == Future)
            val.b = 0.0;

        // more options
    }
};
```

We now use these functions in a test program which we call the *major client* because it is here that we decide to use these functions. The class `Pricer3` knows nothing about these functions and is *policy-free* in this sense:

```
int main()
{ // Major client

    {
        Pricer3 pricer3(PlainDataSource);

        double S = 60.0;
        cout << "Stock, full generalised version: " <<
                    pricer3.compute(S);

        GeneralisedDataSource mySource;

    }

    {
        GeneralisedDataSource mySource;
        Pricer3 pricer3(mySource);

        double S = 60.0;
        cout << "Stock, full generalised version:" <<
            pricer3.compute(S);
    }

    return 0;
}
```

Having completed this self-contained example, we now draw some general conclusions on using C++ in a wider context than just the pure object-oriented one:

- We can create classes and code that are independent of any specific programming paradigm and developer preferences. It is even possible to create code that is a mixture of object-oriented and modular programming techniques.
- Developing applications using the above two-tier approach leads to efficient, reliable and interoperable code. This separation of concerns ensures that each class has a single major responsibility, thus leading to higher maintainability levels.
- The results and ideas in the above example can be used to generalise the object-oriented design patterns that are described in Gamma 1995. In a sense, design patterns based on objects are too restrictive for certain kinds of applications.
- The current methods can be used – in combination with *system partitioning techniques* – to design large systems consisting of components that communicate using provided and requires interfaces (Duffy 2004). We can choose the kind of functions to use based on different quality criteria, for example efficiency. In that case using global functions incurs less overhead than using dynamic polymorphism (by virtual functions) in class hierarchies.

We continue with a discussion of some specific Boost libraries. The emphasis is on giving an overview of the main features in these libraries and some representative examples of use.

3 Statistics Distributions

We give an overview of the univariate statistical distributions and functions in the Math Toolkit. The emphasis is on discussing the functionality in the toolkit, in particular:

- Discrete and continuous distributions, their member functions and defining properties.
- Other non-member functions, for example the probability and cumulative density functions, kurtosis and skewness.
- Some examples to motivate how to use the classes in the toolkit.

All distributions use random variables which are mappings of a probability space into some other space, typically a real number. A discrete probability distribution is one in which the distribution of the random variable is discrete while a continuous probability distribution is one whose cumulative distribution is continuous.
The discrete probability distributions are:

- Bernoulli (a single trial whose outcome is 0 (failure) or 1 (success)).
- Binomial (used to obtain the probability of observing k successes in N trials).
- Negative Binomial (used to obtain the probability of k failures and r successes in k + r trials).
- Hypergeometric (describes the number of events k from a sample n drawn from a total population N without replacement).
- Poisson (expresses the probability of a number of events occurring in a fixed period of time).

The continuous probability distributions are:

- Beta (used in Bayesian statistics applications).
- Cauchy-Lorentz (used in physics, spectroscopy and to solve differential equations).
- Chi Squared (used in statistical tests).
- Exponential (models the time between independent events).
- Extreme Value (models rare events).
- F (The Fisher F-distribution that tests if two samples have the same variance).
- Gamma (and Erlang) (used to model waiting times).
- Laplace (the distribution of differences between two independent variates with identical exponential distributions).
- Logistic (used in logistic regression and feedforward neural network applications).
- Log Normal (used when the logarithm of the random variable is normally distributed).
- Noncentral Beta (a generalisation of the Beta Distribution).
- Noncentral Chi-Squared (a generalisation of the Chi Squared Distribution).

- Noncentral F-distribution (a generalisation of the Fisher F distribution).
- Noncentral T (generalisation of Student's t Distribution).
- Normal (Gaussian) (probably the best known distribution).
- Pareto (compare large and small numbers).
- Rayleigh (combine two orthogonal components having an absolute value).
- Student's t (the 'best' approximate distribution to an unknown distribution).
- Triangular (used when a distribution is only vaguely known, for example in software projects).
- Weibull (used in failure analysis models).
- Uniform (also known as the rectangular distribution and it models a probability distribution with a constant probability).

Each of the above distributions is implemented by a corresponding template class with two template parameters. The first parameter is the underlying data type used by the distribution (the default type if used is `double`) and the second parameter is the so-called *policy*. In general, a policy is a fine-grained compile-time mechanism that we can use to customise the behaviour of a library. It allows us to change error-handling mechanism or calculation precision at both program level and at the client site.

The global functions are:

- cdf (cumulative distribution function).
- cdf complement (this is 1 – cdf).
- hazard (the event rate at time t conditional on survival until time t or later; useful when modelling failure in mechanical systems).
- chf (cumulative hazard function that measures the accumulation of hazard over time).
- kurtosis (a measure of the 'peakedness' of a probability distribution).
- kurtosis_excess (does a distribution have fatter tails than a normal distribution?).
- mean (the expected value).
- median (the value separating the lower and higher halves of a distribution).
- mode (the point at which the probability mass or density function takes its maximum).
- pdf (probability density function).
- range (the length of the smallest interval which contains all the data).
- quantile (points taken at regular intervals from the cdf).
- skewness (a measure of the asymmetry of a probability distribution).
- support (the smallest closed interval/set whose complement has probability zero).
- variance (how far do values differ from the mean).

We discuss a well-known case. The normal (or Gaussian) distribution is one of the most important statistical distributions because of its ability to model many kinds of phenomena in diverse areas such as economics, computational finance, physics and the social sciences. In general, the normal distribution is used to describe variables that tend to cluster around a mean value. We now show how to implement the normal distribution in Boost and we show how to call the member and non-member functions:

```
// Non-member functions
#include <boost/math/distributions.hpp>
#include <boost/math/distributions/normal.hpp>
#include <iostream>
using namespace std;

int main()
{
  // Don't forget to tell compiler which namespace
  using namespace boost::math;

  // Default is 'double'
  normal_distribution<> myNormal(1.0, 10.0);
  cout << "Mean: " << myNormal.mean()
       << ", standard deviation: "
       << myNormal.standard_deviation() << endl;

  // Distributional properties
  double x = 10.25;

  cout << "pdf: " << pdf(myNormal, x) << endl;
  cout << "cdf: " << cdf(myNormal, x) << endl;

  // Choose another data type and now a N(0,1) variate
  normal_distribution<float> myNormal2;
  cout << "Mean: " << myNormal2.mean()
       << ", standard deviation: "
       << myNormal2.standard_deviation() << endl;

  cout << "pdf: " << pdf(myNormal2, x) << endl;
  cout << "cdf: " << cdf(myNormal2, x) << endl;

  // Choose precision
  // Number of values behind the comma
  cout.precision(10);

  // Other properties
  cout << "\n***normal distribution: \n";
  cout << "mean: " << mean(myNormal) << endl;
  cout << "variance: " << variance(myNormal) << endl;
  cout << "median: " << median(myNormal) << endl;
  cout << "mode: " << mode(myNormal) << endl;
  cout << "kurtosis excess: "
       << kurtosis_excess(myNormal) << endl;
  cout << "kurtosis: " << kurtosis(myNormal) << endl;
  cout << "characteristic function: "
       << chf(myNormal, x) << endl;
  cout << "hazard: " << hazard(myNormal, x) << endl;

  return 0;

}
```

We conclude this section with some remarks on using the current functionality in combination with Excel. The primary motive was to call Boost functionality from C#. The Microsoft .NET supports the creation of interoperable applications consisting of both C++ and C# code.

To this end, we use the C++/CLI language to create *wrapper classes* that embed native C++ code and that can be called from C#. An example is the case of the non-central Chi-squared distribution. We need to use its functions by calling them from C#. The corresponding C++/CLI wrapper class is:

```
#include <boost/math/distributions.hpp>

using namespace System;

// Wrapper for the
// boost::math::non_central_chi_squared_distribution
// class. We use the .NET naming conventions
// instead of the original C++ name
public ref class NonCentralChiSquaredDistribution
{
private:
   // The wrapped native class
   boost::math::non_central_chi_squared_distribution<>*
       m_distribution;

public:
   // Default constructor
   NonCentralChiSquaredDistribution();

   // Constructor with lower and upper value
   NonCentralChiSquaredDistribution(double df, double
       lambda);

   // Finaliser (called by garbage collector
   // or destructor)
   !NonCentralChiSquaredDistribution();

   // Destructor (Dispose)
   ~NonCentralChiSquaredDistribution();

   // Get the native object
   boost::math::non_central_chi_squared_distribution<>*
       GetNative();

   double Pdf(double x);
   double Cdf(double x);
};
```

We are primarily interested in the probability density and cumulative probability density functions that we have implemented as follows:

```
double NonCentralChiSquaredDistribution::Pdf(double x)
{
   return boost::math::pdf(*GetNative(), x);
}
double NonCentralChiSquaredDistribution::Cdf(double x)
{
   return boost::math::cdf(*GetNative(), x);
}
```

We see how easy it is to call Boost code from C#. This is a general pattern that can also be used in other situations.

The Math Toolkit is well-documented (see www.boost.org) and it contains full descriptions of the classes, the corresponding global functions and numerous applications.

4 Special Functions

In this section we introduce a number of software modules from the Boost Math Toolkit for the so-called *Special Functions*. Categories are:

- Gamma and beta functions.
- Factorials and binomial coefficients.
- Error function.
- Orthogonal polynomials.
- Bessel functions.
- Elliptic integrals.
- Zeta functions.
- Other functions (exponential integrals and sinus cardinal functions).

These functions have many applications in mathematical physics, statistics and engineering. For example, a number of statistical distributions are defined in terms of gamma and beta functions. Other applications include the solution of differential equations, signal processing and eigenvalue analysis. The library supports many of the functionality that these applications need. Using the library in your applications is easy and an advantage is that the code is portable. You do not have to write your own library suite or use proprietary class libraries.

The Math Tookit has documented these special functions in great detail and for this reason we do not repeat the mathematical formulae here. Instead, we take the example of the *error function* and its variants:

```
#include <boost/math/special_functions/erf.hpp>
#include <cmath>
#include <iostream>
using namespace std;

int main()
{
   using namespace boost::math;

   // Error function
   double z = 2.75;
   cout << "Error function:" << erf(z) << endl;
   z = -3.0;
   cout << "Error function: " << erf(z) << endl;

   // Complement of error function
   z = 2.75;
   cout << "Error function, complement:" << erfc(z) <<
          endl;
   z = -3.0;
   cout << "Error function, complement:" << erfc(z) <<
          endl;
```

```
   // Inverse error function
   double u = 0.5;
   try
{
  // The variable 'u' must be in range (-1,1)
  // CHECK THE ERROR MESSAGE
  cout << "erf inverse: " << erf_inv(u) << endl;
  cout << "erf inverse: " << erf_inv(u - 1.0)
       << endl;

  // The variable 'u' must be in range (0,2)
  // CHECK THE ERROR MESSAGE
  cout << "erfc inverse: " << erfc_inv(u) << endl;
  cout << "erfc inverse: " << erfc_inv(1.99999)
       << endl;
}
   catch(const std::exception& e)
   {
       cout << e.what() << endl;
   }

   cout << "Give a value: "; double val; cin >> val;
       double t; cin >> t;

   cout << 0.5 * (1.0 + erf(val/sqrt(4.0*t)));

   return 0;
}
```

In general, the special functions in Boost are very easy to use and they have many applications in engineering, mathematical physics and computational finance.

5 Matrix and Vectors in uBLAS Library

The Boost uBLAS library supports vector and matrix data structures and basic linear operations on these structures. The syntax closely reflects mathematical notation because operator overloading is used. Furthermore, the library uses *expression templates* to generate efficient code. The library has been influenced by a number of other libraries such as ATLAS, BLAS, Blitz++, POOMA and MTL. The main design goals are:

- Use mathematical notation.
- Efficiency (time and resource management).
- Functionality (provide features that appeal to a wide range of application areas).
- Compatibility (array-like indexing and use of STL allocators for storage allocation).

The two most important data structures represent vectors and matrices. A *vector* is a one-dimensional structure while a *matrix* is a two-dimensional structure. We can define various vector and (especially) matrix *patterns* that describe how their elements are arranged in memory; examples are dense, sparse, banded, triangular, symmetric and Hermitian matrices.

These patterned matrices are needed in many kinds of applications and they can be used directly in code without you having to create them yourself. Furthermore, we can apply primitive operations on vectors and matrices:

- Addition of vectors and matrices.
- Scalar multiplication.
- Computed assignments.
- Transformations.
- Norms of vectors and matrices.
- Inner and outer products.

We can use these operations in code and applications. Finally, we can define subvectors and submatrices as well as ranges and slices of vectors and matrices.

Vectors and matrices are fundamental to scientific and engineering applications and having a well-developed library such as uBLAS with ready-to-use modules will free up developer time. Seeing that matrix algebra consumes much of the effort in an application we expect that the productivity gains are appreciable in general.

uBLAS supports these different kinds of matrix structures and patterns as well as the operations that apply to them, for example:

- *Patterned matrices*, for example symmetric, triangular, banded and Hermitian matrices.
- *Vector and matrices expressions*, for example vector and matrix products and norms.
- *Vector and matrix proxies*, for example creating ranges and slices.

There are many applications of matrix theory. We discuss a particular application to the solution of matrix systems, namely LU decomposition. The specific problem is given a matrix we wish to decompose it into the product of a lower-triangular matrix and an upper-triangular matrix. To this end, we can use the patterned matrices in uBLAS, use the algorithm for LU decomposition and map it to C++:

```
void InitLU(const u::matrix<double>& A,
  u::triangular_matrix<double, u::lower>& L,
  u::triangular_matrix<double, u::upper>& U)
{ // LU decomposition: A -> L*U

   double sum;
   unsigned N = A.size1();

   // Common to make all diagonal elements == 1.0
   for (unsigned k = 0; k < N; ++k)
   {
       L(k,k) = 1.0;
   }

   for (unsigned j = 0; j < N; ++j)  // Loop over columns
   {
       for (unsigned i= 0; i <= j; ++i) // Columns
       {
             sum = 0.0;
             for (unsigned k = 0; k < i; ++k)
```

```
            {
                    sum += L(i,k)*U(k,j);
            }
            U(i,j) = A(i,j) - sum;
    }


    // L
    for (unsigned i = j+1; i < N; ++i) // Rows
    {
            sum = 0.0;
            for (unsigned k = 0; k < j; ++k)
            {
                    sum += L(i,k)*U(k,j);
            }
            L(i,j) = (A(i,j) - sum) / U(j,j);
    }
  }
}
```

In general, uBLAS can be used by code that needs to deploy matrix and vector structures.

6 Conclusion

We have given an overview of some important libraries in Boost. We focused on a number of issues that promote the flexibility and applicability of C++ applications. In particular, the mathematical libraries are useful when creating C++ applications.

Finally, we give some personal views on how Boost can be used to help developers create applications in computational finance. In general, it is more effective to use mature and tested libraries rather than creating your own libraries unless of course it is in your own area of expertise. To this end, a common software design technique is to develop software systems in layers: software in higher layers use the software in lower layers. In the current context quant developers can use the functionality in STL and Boost and integrate it into computational finance. Since there are more than one hundred Boost libraries we first need to determine which libraries are suitable and second how to use these libraries in applications. By examining the categories that we discussed in Part I we see that these objectives are feasible.

Daniel J. Duffy works for Datasim Education BV. His main activities are software design and practice, and the application of modern numerical methods to option pricing applications. He has a PhD in numerical analysis from Trinity College, Dublin.

REFERENCES

Demming, R and Duffy, D.J. 2010. *Introduction to the Boost C++ Libraries Volume I-Foundations*. Datasim Press: Amsterdam.

Demming, R. and Duffy, D.J. 2011. *Introductions to the Boost C++ Libraries Volume II – Advanced Libraries*. Datasim Press: Amsterdam.

Duffy, D.J. 2004. *Domain Architectures Models and Architectures for UML Applications* John Wiley & Sons.

Gamma, E., Helkm, R., Johnson, R. and Llissides, J. 1995. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Boston: Addison-Wesley: Boston.

Leavens, G.T. and Sitaraman, M. 2000. *Foundations of Component-Based Systems* Cambridge University Press.

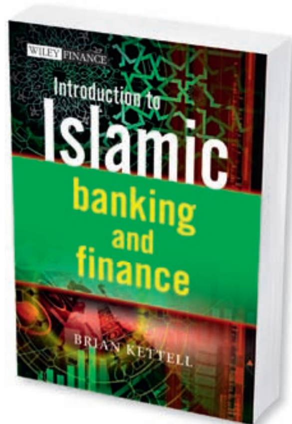
Book Club

Share our passion for great writing – with Wiley's list of titles for independent thinkers ...

Introduction to Islamic Banking and Finance

Brian Kettell

Introduction to Islamic Banking and Finance is a complete guide to the key characteristics of Islamic banking and finance, and explains how they differ from their Western counterparts. It looks at all aspects of Islamic banking, including detailed chapters on its creation through to explanations of Murabaha and Musharaka contracts, Ijara and Istisna'a financing methods, as well as Salam and Takaful insurance. Finally the book takes a look at Sharia'a law and Sharia'a boards, indicating the roles and responsibilities that come with membership.



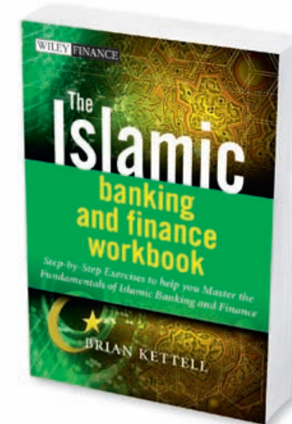
978-0-470-97804-7 • Paperback
192 pages • June 2011
~~£34.99 / €42.00~~ £20.99 / €25.20

The Islamic Banking and Finance Workbook

Step-by-Step Exercises to help you Master the Fundamentals of Islamic Banking and Finance

Brian Kettell

The first practical workbook of its kind that promotes the understanding of Islamic banking and finance, by allowing readers to self-test their knowledge of Islamic finance and banking concepts. The *Workbook* includes a full answer key and brief chapter summaries and learning objectives, making the information that readers attain from *An Introduction to Islamic Banking and Finance* that much more valuable.

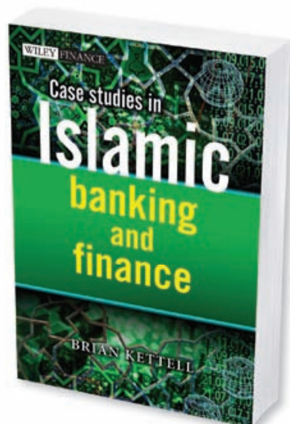


978-0-470-97805-4 • Paperback
152 pages • June 2011
~~£29.99 / €36.00~~ £17.99 / €21.60

Case Studies in Islamic Banking and Finance

Brian Kettell

The first case study based guide to Islamic banking and finance. Based around 13 individual cases, the book stimulates discussion and develops the reader's understanding of Islamic finance by contrasting the existing theoretical knowledge against practical examples. Each chapter concludes with a set of questions designed to test the readers understanding of each case, with suggested solutions at the end of the book.



978-0-470-97801-6 Paperback
192 pages • June 2011
~~£39.99 / €48.00~~ £23.99 / €28.80

2 Volume Set includes…

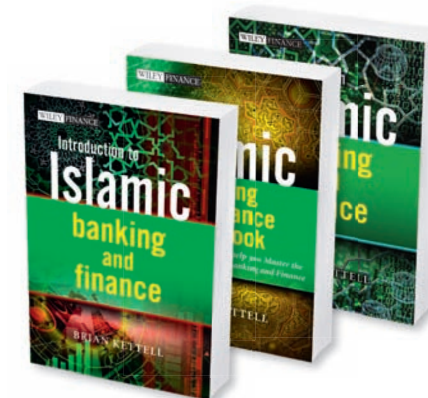
- Introduction to Islamic Banking and Finance
- The Islamic Banking and Finance Workbook

978-1-1199-8995-0
Paperback • 344 pages • June 2011
~~£55.00 / €66.00~~ £33.00 / €39.60

3 Volume Set includes…

- Introduction to Islamic Banking and Finance
- The Islamic Banking and Finance Workbook
- Case Studies in Islamic Banking and Finance

978-1-1199-8996-7 • Paperback • 536 pages
June 2011 • ~~£90.00 / €108.00~~ £54.00 / €64.80



SAVE 40%

When you subscribe to *Wilmott* magazine you will automatically become a member of the *Wilmott* Book Club and you'll be eligible for 40 per cent discount on specially selected books in each issue. The titles will range from finance to narrative non-fiction. For further information, call our Customer Services Department on **+44 (0)1243 843294**, or visit wileyeurope.com/go/wilmott or wiley.com (for North America)