Computational Finance Using QuantLib-Python

Jayanth R. Varma and Vineet Virmani | Indian Institute of Management, Ahmedabad

QuantLib is a reliable C++ open source library for pricing derivatives. Its Python extension, QuantLib-Python, lets users harness the power of C++ with the ease of IPython notebooks for use in both classrooms and student projects.

Financial engineering and algorithmic trading are perhaps two of the most computationally intensive parts of finance. A job in either area requires not only a reasonable expertise in finance, mathematics, and statistics, but also—and perhaps more importantly today—sophistication in computing. Because algorithmic trading focuses on finding opportunities in markets that give a temporary statistical edge, speed and data mining are essential. In contrast, financial engineering is primarily concerned with pricing and managing derivatives (or *structured products*) designed to meet specific requirements of large banks and corporations. The pricing of such products relies on the same principles as those underlying the famous Nobel prize-winning Black-Scholes formula.[1] Although the formula is named after the Nobel-winner Myron Scholes and the late Fischer Black, the name of the formula itself was given by Robert Merton, the other Nobel winner responsible for creating the theoretical foundations behind the principle.[2]

The formula is applicable only for the simplest of derivatives (plain-vanilla Call and Put options), but the underlying principle is more general. Starting with a stochastic differential equation (SDE) for the price process of an asset, the Black-Scholes approach leads to a parabolic partial differential equation (PDE) for the price of the derivative. The Merton formulation[3] relies more on probabilistic ideas and leads to the price as a mathematical expectation. These two alternative, but equivalent, approaches form the basis of much of financial engineering applications today.

Although the two approaches are theoretically equivalent, in practical implementation, the PDE approach relies on finite difference (FD) methods and the probabilistic approach relies on Monte Carlo (MC) simulation. Given the complexity of modern structured products, almost all pricing and risk management is based on such numerical methods. Large banks typically have their own team of "quants" and IT developers hired to implement sophisticated FD and MC engines, but many boutique firms have emerged over the past decade to provide such specialized services and software for a fee to other banks and organizations.

Until a few years ago, none of that sophistication was available for use in teaching and research. Neither banks nor boutique firms share their proprietary software, and if they're available at all, they're either prohibitively expensive or downright useless. For the past few years, however, a reliable open source library has been available: *QuantLib*, which is built in C++ and also available for languages such as Python, Ruby, R, and Excel.

Here, we introduce QuantLib for computational finance applications in teaching and research, along with a worked-out example. The fact that QuantLib is also available (and extendable) in Python lets users harness the power of C++ with the ease of IPython notebooks.

The Pricing Problem

The market for financial derivatives worldwide today is so large that the size of outstanding positions—nearly a thousand-trillion US dollars—is many times that of stock markets. A significant part of this market consists of over-the-counter structured products whose complexity varies from a plain vanilla European Call option to a Bermudan cross-currency Swaption.

A popular example is a Barrier option, which we use later to illustrate the use of QuantLib-Python. A Barrier option is a derivative whose payoff depends on whether the price of the underlying security crosses a prespecified level (that is, the *barrier*) before the expiration.

The pricing problem for such derivatives typically entails working with an SDE for the price process (S_t), such as the standard geometric Brownian motion (GBM):

$$dS_t = rSdt + \sigma SdW_t,$$

where W_t is a standard Brownian motion and the interest rate r is assumed to be a constant.

Assuming (as we do here) that the diffusion coefficient $\sigma(t, S_t)$ for the Brownian motion is deterministic, the Black-Scholes argument based on hedging gives the PDE for the price of the derivative $V(t, S_t)$ as:

$$\frac{\partial V}{\partial t} + rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} = rV,$$

with the necessary boundary conditions coming from the specification of the product being priced.

The Merton argument, on the other hand, is based on showing a "risk-neutral" probability measure and results in $V(t, S_t)$ as the following mathematical expectation:

$$V(t, S_t) = e^{-r(T-t)}\tilde{E}[V(T, S_T)],$$

where T represents the maturity/expiration of the product being priced, and $\tilde{E}$ is the expectation corresponding to the risk-neutral measure.

The two approaches might look different and seem to be leading to different solutions, but that's not the case. The Feynman-Kac theorem for parabolic PDEs ensures that they're theoretically consistent.[4]

Numerical Methods

For both of the above approaches, the numerical methods are already well developed in applied mathematics and probability. In particular, for the task at hand, the most popular methods for numerically solving the PDE are explicit and implicit FD methods and their variants, including the Crank-Nicholson and Douglas methods.[5]

The state of the art in simulation-based methods constitutes working with Sobol sequences and low discrepancy numbers. It's also common to use variance reduction methods whenever possible, of which antithetic and control variates are the most popular.[6]

In most practical applications, it's often clear which of the two approaches is better suited for a job; generally, wherever the curse of dimensionality is high (that is, the number of assets is greater than or equal to 4), the MC method is preferred. When pricing products such as Barrier options, in which the value of the underlying security must be monitored regularly (often continuously), FD methods are more popular as users can easily enforce the location of the barrier to lie on the FD grid.

QuantLib

The QuantLib open source project started in 2000 at the Italian boutique risk-management firm RiskMap (now called *StatPro Italia*). The QuantLib website (www.quantlib.org) states that the project's aim is to provide "a comprehensive software framework for quantitative finance."

The first QuantLib package was released in December 2000 under a liberal Berkeley Software Distribution (BSD) license that has allowed banks and software companies to extend and modify the code without having to release it back. Although the original developers, Luigi Ballabio and Ferdinando Ametrano, remain involved in the library's development and maintenance, the project today has more than 150 contributors, some of whom make substantial contributions. Financially, StatPro continues to support the project—while also using it for consulting and training—but QuantLib's growth and quality has been driven mainly by the open source financial software community's contributions (for a list of contributors, see https://github.com/lballabio/quantlib/blob/master/QuantLib/Contributors.txt).

Written in C++, the library comes with more than 500 unit tests using the Boost library (Boost became a prerequisite after July 2004). It also makes an extensive use of Simplified Wrapper and Interface Generator (SWIG), and bindings exist for a variety of languages including Python, R, Ruby, Excel, and C#, although not all are equally well developed at this stage (Python is one of the most popular and developed).

Although it assumes a working knowledge of both C++ and quantitative finance, the project page offers a host of information for novices to get started. It begins with help on installation (www.quantlib.org/install.shtml) and video examples with IPython notebooks (www.vimeo.com/channels/qlnotebooks), then offers a detailed reference manual (www.quantlib.org/reference/index.html) and a blog

Installation Instructions for QuantLib and QuantLib-Python

For QuantLib-Python to work, QuantLib must be working first. Here, following the discussion of common prerequisites, we provide step-by-step instructions for both Ubuntu (12.04 and above) and Windows (7 and above).

Prerequisites

First, for installing in Windows, a working C++ environment is required. Here, the instructions are meant for novices, so we recommend that users install the free Microsoft Visual Express Desktop 2013 edition, which comes with Visual C++ 12 (MSVC12). Although it's equally easy to install in Windows on Cygwin or using MinGW, users familiar with those probably wouldn't need this user guide.

Second, QuantLib-Python requires a working Python environment, and we recommend the Anaconda metapackage. Instructions for both Ubuntu and Windows are available from the Anaconda install page (docs.continuum.io/anaconda/install.html). For our purpose here, Windows users should work with the 32-bit version of Anaconda. We recommend installing Anaconda with all the default settings and then updating it by running `conda update conda` and `conda update anaconda`. Again, it doesn't matter if you install Anaconda-2.x (with Python 2.7) or Anaconda-3.x (with Python 3), but we recommend installing the Python 3 version, as that's where the Python language seems to be headed.

Additional ingredients include Boost C++ libraries and SWIG; instructions for those are different for Ubuntu and Windows, and are provided below. Finally, you should download the same versions of QuantLib and QuantLib-Python, which are available at http://sourceforge.net/projects/quantlib/files/quantlib/1.6.

Installing in Ubuntu

For Ubuntu, we recommended not installing QuantLib from the synaptic package manager, as Ubuntu repositories don't contain the latest version. Instead, do the following:

- Install the Boost C++ libraries; in Ubuntu, they're available from the repositories (in the "libboost-all-dev" package).
- After installing Boost, install SWIG, which is also available from the repositories (in the "swig" package).
- Instructions for installing QuantLib for Ubuntu are available from the QuantLib project page (www.quantlib.org/install/linux.shtml). However, before proceeding, users should ensure that examples given on the QuantLib page are working and not giving any errors.

After installing QuantLib, QuantLib-Python requires running the following steps (in this order):

1. `cd \path\to\QuantLib-SWIG-1.6\Python`
2. `python setup.py wrap`
3. `python setup.py build`
4. `python setup.py test`
5. `sudo python setup.py install`

Advanced users can alternatively install the latest versions of Boost and SWIG directly from source.

Finally, if user-defined .i SWIG files have been added (or you have modified the existing files), repeat the last four steps (from wrap to install) to ensure the files are available,

Installing in Windows

If Windows users are working with a free version of Visual Studio, they should install the 32-bit versions of both Anaconda and Boost because free Visual Studio lacks the necessary toolkit

(www.implementingquantlib.com) maintained by Ballabio. Despite the available information, we have found that users migrating from Microsoft Excel and the like can get intimidated with the steps involved. To address this, the "Installation" sidebar offers step-by-step instructions for the latest versions of QuantLib and QuantLib-Python for both Linux (Ubuntu) and Windows operating systems.

QuantLib is developed to be completely object oriented, and it makes extensive use of design patterns. Even if someone isn't developing models, it offers a good example for learning how to use design patterns when building a financial library.

In terms of financial applications, QuantLib not only includes classes for market conventions and yield curve models, it also comes with low-discrepancy sequences and solvers for PDEs with a large choice of alternative algorithms and exotic payoffs. For modern multicore processors, QuantLib also allows multithreading via OpenMP (though this feature is still under development).

QuantLib Structure: Important Classes

The price of any derivative—be it a plain-vanilla option or a complex structured product—depends on the following inputs:

- the price of the underlying securities, either as a direct input on the date of pricing or picked up automatically from price feeds;
- the term structure of interest rates, volatility, inflation, and default probabilities;
- cash flows (including coupons and dividends) from the instrument;

for building QuantLib-Python for 64-bit versions (you can make the free version work for 64-bit, but it's not guaranteed to be replicated universally). Users with access to the professional Visual Studio environment can choose 64-bit for everything.

Throughout, `\path\to\someplace` represents the directory of `someplace`. For example, if `boost_1_58_0` is installed in `C:\boost`, `\path\to\boost_1_58_0`, it should be taken to mean `C:\boost\boost_1_58_0`.

- For Windows, prepackaged binaries for Boost are available—the latest version is 1.58.0—for specific versions of MSVC (in our case, MSVC12) from its SourceForge page. Users should download the executable for the 32-bit architecture (http://sourceforge.net/projects/boost/files/boost-_binaries/1.58.0/).
- After installing Boost, install SWIG. Prepackaged binaries for SWIG for Windows are available from the source page (www.swig.org/download.html), and it's enough to extract the SWIG zip file (the latest version, swigwin-3.0.5) in a convenient folder.
- All the commands below assume that we're working in a Visual Studio command prompt (that is, that all relevant Visual Studio-related environment variables have been set). The Visual Studio command prompt can be launched from the Start menu (in Windows 7) or from Apps (in Windows 8). You can also find out how to locate it at https://msdn.microsoft.com/en-_us/library/ms229859(v=vs.110).aspx.

After launching Visual Studio command prompt, QuantLib in Windows can be installed directly in three simple steps (minding the gaps):

1. `cd \path\to\QuantLib-1.6`
2. `set myboost=\path\to\boost`
3. `msbuild /p:AdditionalLibPaths="%myboost\lib32-msvc-12.0" /p:Configuration=Release /p:Platform=Win32 QuantLib_vc12.sln`

This last step can take a while (on an Intel i5, Windows 7 machine with 4 Gbytes RAM, it took almost 45 minutes), so it's advisable to have a few hours on hand when sitting down to install QuantLib.

Installing QuantLib-Python in Windows requires some extra settings. After launching the Visual Studio command prompt (`cd \path\to\QuantLib-SWIG-1.6\Python`), set the following (again, minding the gaps):

1. `set PATH=\path\to\Anaconda3;\path\to\Anaconda3\scripts;\path\to\swigwin-3.0.5;\path\to\QuantLib-1.6\lib;%PATH%`
2. `set QL_DIR=\path\to\QuantLib-1.6`
3. `set VS100COMNTOOLS=%VS120COMNTOOLS%`
4. `set INCLUDE=\path\to\boost_1_58_0;%INCLUDE%`
5. `set LIB=\path\to\boost\lib32-msvc-12.0;%LIB%`
6. `echo [build] > setup.cfg && echo compiler=msvc >> setup.cfg`

The last few steps are identical to that in Ubuntu:

1. `cd \path\to\QuantLib-SWIG-1.6\Python`
2. `python setup.py wrap`
3. `python setup.py build`
4. `python setup.py test`
5. `python setup.py install`

If user-defined .i SWIG files have been added (or existing files have been modified), you must go through the "set" commands before running the last four steps (from wrap to install) to ensure that the files are available.

- a stochastic process; and
- a pricing engine (the numerical method used for pricing).

Now, if you're pricing a stand-alone product for a student project, you could simply code the above elements in a single monolithic program to find the price; you don't necessarily need to use an object-oriented approach such as QuantLib. In a large financial institution or a hedge fund, however, derivatives are part of a larger portfolio, and this monolithic approach quickly becomes inefficient and impractical.

The beauty of QuantLib is that, in its spirit and scope, it's very similar to the financial library that you find in large banks.[7] It's thus natural to use QuantLib for computational finance even in the classroom, as it helps students quickly come up to speed with the state of the art. This not only helps make the curriculum more relevant, it also helps attract the best and most interested students to the class (and the institution).

Although many classes are important to implementing processes—such as random number and path generation (RandomSequenceGenerator and Path classes), calibration (the CalibrationHelper and CalibratedModel classes), and models such as trees (the Lattice and DiscretizedAsset classes)—following are the classes we discuss in our example. The other classes are described both in the QuantLib reference manual and Ballabio's book.[8]

The Instrument class. This class is designed to accommodate financial instruments whose values depend on the value of other (underlying) securities. Every instance of the Instrument class must then maintain

It's only in the textbook version of the Black-Scholes that interest rates and volatility are treated as constants. In the messy world of central bank interventions and market volatility, both are time-varying.

"links" so that, when called, it would access the most current values of the underlying security. At the same time, caching is desirable—that is, the value of the instruments should be recalculated only when one or more of the input values has changed. In QuantLib, this is operationalized using the Observer design pattern, in which the instrument plays the role of the observer and inputs are the observables.

The TermStructure class. It's only in the textbook version of the Black-Scholes that interest rates and volatility are treated as constants. In the messy world of central bank interventions and market volatility, both are time-varying. The TermStructure class is responsible for constructing time-varying objects for all such variables. In particular, this class has three jobs:

- keep track of its own (reference) date and calculate the appropriate future date—using the number of business days forward—if its reference date is different from the evaluation date;
- convert dates to times, such as when converting discount factors to zero yields; and
- check whether a given date/time belongs to the domain covered by the term structure, such as setting the maximum allowed date/range when fitting volatility term structures.

With reference dates described in the TermStructure class, specific classes exist to specify yield curve, volatility, inflation, and default probability, along with specific inherited classes to capture known special cases across different assets.

The Payoff and Exercise classes. All derivatives require the specification of a payoff at expiration or early close out. In QuantLib, the payoffs are derived from the Payoff class, whose descendants include PlainVanillaPayoff (which could be set to Call or Put by means of a switch), AssetOrNothingPayoff, and many others. The use of such classes is illustrated later in our example.

For derivatives with an early-exercise feature, a crucial property is *exercise possibilities available*. In QuantLib, this is operationalized using the Exercise base class, which—depending on the exercise choices available—can be accessed as a EuropeanExercise (at expiration only), a BermudanExercise (on a set of discrete dates prior to expiration), or an AmericanExercise (any time before expiration) class.

The StochasticProcess class. All derivatives pricing begins with the assumption of an appropriate stochastic process for the underlying security and volatility. For example, the Black-Scholes model begins with a GBM for the stock price, and the Heston model[9] builds on it by adding a square-root process for instantaneous variance.

In QuantLib, stochastic processes are specified using the StochasticProcess Observer design pattern. The StochasticProcess class descends into a discretization class (the most important being the EulerDiscretization), which handles how the process is passed into the pricing engine.

The PricingEngine class. An instrument can be priced using any of the different pricing engines available in the library and with different objectives in mind. For example, a Call option might need to be priced for its own sake or to derive implied volatilities or for calibrating a stochastic volatility model. Any of these might need to be done via FD or MC methods.

In QuantLib, this pricing is operationalized using the PricingEngine class, which is modeled as the Strategy pattern, in which instruments take an object that encapsulates the computation to be performed. This allows the instrument to be priced using any of the library's various available engines.

For implementation reasons, most pricing engines don't descend from the PricingEngine class but rather from a generic subclass called *GenericEngine*. Currently, the pricing engines are available for Asian, Barrier, Basket, Cap/floor, Cliquet, Forward, Quanto, Swaption, and Vanilla option engines. For most engines, all three procedures (Analytic, FD, and MC) are available.

Example: Pricing Barrier Option Using FD

Although the Barrier option is a simple enough product, and its correct value is available as a closed-form formula, it's perfectly suited to illustrate the use of QuantLib and QuantLib-Python because it allows us to discuss the main challenges faced in pricing, without the complications of an exotic product. In particular, we discuss the pricing of a Down-and-Out Call option using the FD method implemented in QuantLib. The fact that the true value can be computed analytically[10] also allows us to compute the pricing error of the FD scheme.

A Barrier option (colloquially known simply as *barriers*) is a derivative whose payoff depends on whether the price of the underlying security crosses a prespecified level before the expiration. Barriers come in two basic varieties:

- *Knock-in*: the option gives a payoff only if the barrier is breached before expiration.
- *Knock-out*: the option expires if the barrier is breached before expiration.

Barriers are also typically categorized in relation to the current value of the underlying security. So, if the barrier level of a knock-out option is set to a value below (above) the asset's current value, it's referred to as a "Down-and-Out" (or "Up-and-Out") option. The knock-in options are defined similarly. In plain-vanilla barriers, the payoff at expiration could be either a Call or a Put variety. Putting all variants together, then, eight different combinations of plain-vanilla Barrier options exist.

To mathematically represent the payoff of a Barrier option, it's convenient to call m_T and M_T as the minimum and maximum values. respectively, of the asset between the evaluation (t) and expiration (T) date as follows:

$$m_T = \min_{0\le t\le T} S_t$$
$$M_T = \max_{0\le t\le T} S_t .$$

Given a barrier level B_d, the payoff of a Down-and-Out (DO) Call option, for example, is then written as

$$V_T(\text{DO Call}) = (S_T - K)^+ I[m_T > B_d]$$

where $I[m_T > B_d]$ represents an indicator variable that takes the value 1 if the minimum value of the asset before expiration lies above the barrier.

Similarly, for an Up-and-In (UI) Put option with barrier B_u, the payoff is

$$V_T(\text{UI Put}) = (K - S_T)^+ I[M_T > B_u],$$

where $I[m_T > B_d]$ represents an indicator variable, which takes the value 1 if the maximum value of the asset before expiration crosses the barrier B_u. For example, standard Black-Scholes PDE for the value of a UO Call option V_t is

$$\frac{\partial V}{\partial t} + rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} = rV,$$

with the associated boundary conditions as follows:

$$V(t, B_d) = 0 \qquad 0 \le t \le T$$
$$V(T, S_T) = (S_T - K)+ \qquad S_T > B_d .$$

Other Barrier option variants are handled similarly, except that the math (and implementation) is easier for pricing knock-out options. It's then standard to use the fact that the sum of a knock-out and a knock-in option must be the same as the price of an equivalent plain-vanilla option. The price of a knock-in option is then derived as a difference between the price of an equivalent plain-vanilla option and a knock-out option.

Method: Alternative FD Schemes

The FD method begins with discretizing the three partial derivatives in the Black-Scholes PDE, solving and propagating back from the boundary condition at expiration to the evaluation date.

When discretizing the partial derivatives, the choice of backward or forward difference turns out to be an important one. The Explicit FD scheme uses backward difference in t, and the Implicit FD scheme uses the forward difference in t. Borrowing the notation from Paul Wilmott's standard text,[11] the value of the option at each point on the FD grid is written as

$$V_i^k = V(i\delta S, T - k\delta t),$$

where the grid is thought to be made of points in asset values $S = i\delta S$ and times ($t = T - k\delta t$), with time counted backward from expiration to evaluation date.

In particular, in the Explicit FD scheme, the space partial derivatives are discretized as

$$\frac{\partial V}{\partial S} \approx \frac{V_{i+1}^k - V_{i-1}^k}{2\partial S}$$
$$\frac{\partial^2 V}{\partial S^2} \approx \frac{V_{i+1}^k - 2V_i^k + V_{i-1}^k}{\delta S^2}.$$

In the Implicit FD scheme, they're discretized as

$$\frac{\partial V}{\partial S} \approx \frac{V_{i+1}^{k+1} - V_{i-1}^{k+1}}{2\partial S}$$
$$\frac{\partial^2 V}{\partial S^2} \approx \frac{V_{i+1}^{k+1} - 2V_i^{k+1} + V_{i-1}^{k+1}}{\delta S^2}.$$

The time partial derivative is implemented in the same way in both schemes:

$$\frac{\partial V}{\partial t} \approx \frac{V_i^{k+1} - V_i^{k+1}}{\delta t}.$$

SWIG options.patch File

```
--- options.i 2015-01-16 16:22:45.000000000 +0530
+++ modified-options.i 2015-02-23 13:30:23.096148000 +0530
@@ -1099,6 +1099,63 @@ class MCBarrierEnginePtr : public boost:
     }
 };

+///////////////// added for finite difference barrier valuation /////////
+
+%{
+using QuantLib::FdmSchemeDesc;
+%}
+
+struct FdmSchemeDesc {
+  enum FdmSchemeType { HundsdorferType, DouglasType,
+                       CraigSneydType, ModifiedCraigSneydType,
+                       ImplicitEulerType, ExplicitEulerType };
+
+  FdmSchemeDesc(FdmSchemeType type, Real theta, Real mu);
+
+  const FdmSchemeType type;
+  const Real theta, mu;
+
+  // some default scheme descriptions
+  static FdmSchemeDesc Douglas();
+  static FdmSchemeDesc ImplicitEuler();
+  static FdmSchemeDesc ExplicitEuler();
+  static FdmSchemeDesc CraigSneyd();
+  static FdmSchemeDesc ModifiedCraigSneyd();
+  static FdmSchemeDesc Hundsdorfer();
+  static FdmSchemeDesc ModifiedHundsdorfer();
+};
+
```

The Douglas scheme is basically a weighted-average of the Implicit and Explicit FD schemes, and is implemented as follows:

$$\frac{\partial V}{\partial S} \approx \theta\left(\frac{V_{i+1}^{k} - V_{i-1}^{k}}{2\partial S}\right) + (1-\theta)\left(\frac{V_{i+1}^{k+1} - V_{i-1}^{k+1}}{2\partial S}\right)$$

$$\frac{\partial^2 V}{\partial S^2} \approx \theta\left(\frac{V_{i+1}^{k} - 2V_{i}^{k} + V_{i-1}^{k}}{\delta S^2}\right) + (1-\theta)\left(\frac{V_{i+1}^{k+1} - V_{i}^{k+1} + V_{i-1}^{k+1}}{\partial S^2}\right).$$

Note that the Douglas scheme with $\theta = 1$ is the same as the Explicit scheme, while with $\theta = 0$, the Douglas scheme is the same as the Implicit scheme. The Douglas scheme with $\theta = ½$, is known as the *Crank-Nicholson method*.

Given the time step of δt and the space step of δS, the error in both the Explicit and Implicit FD schemes is of the order $O(\delta t,\ \delta S^2)$, so it decreases like δt. Although little difference exists in the computational effort required to implement the Implicit and Crank-Nicholson schemes, the latter is preferred as the error decreases as δt^2.

Implementation

We now have the set up required to describe the implementation of our example in QuantLib-Python.

Once QuantLib and QuantLib-Python have been installed, the single line `from QuantLib import *` at the beginning of a Python script is sufficient to provide access to all the functions in the QuantLib library. It's not necessary to know anything about the C++ language at all. The Python

```
+%{
+using QuantLib::FdBlackScholesBarrierEngine;
+typedef boost::shared_ptr<PricingEngine> FdBlackScholesBarrierEnginePtr;
+%}
+
+%rename(FdBlackScholesBarrierEngine) FdBlackScholesBarrierEnginePtr;
+class FdBlackScholesBarrierEnginePtr : public boost::shared_ptr<PricingEngine> {
+  public:
+    %extend {
+        FdBlackScholesBarrierEnginePtr(const GeneralizedBlackScholesProcessPtr& process,
+          Size tGrid = 100, Size xGrid = 100, Size dampingSteps = 0,
+          const FdmSchemeDesc& schemeDesc = FdmSchemeDesc::Douglas(),
+          bool localVol = false,
+          Real illegalLocalVolOverwrite = -Null<Real>()) {
+            boost::shared_ptr<GeneralizedBlackScholesProcess> bsProcess =
+                 boost::dynamic_pointer_cast<GeneralizedBlackScholesProcess>(
+                                                                     process);
+            QL_REQUIRE(bsProcess, “Black-Scholes process required”);
+            return new FdBlackScholesBarrierEnginePtr(
+                            new FdBlackScholesBarrierEngine(bsProcess,
+                                               tGrid,  xGrid, dampingSteps,
+                                               schemeDesc, localVol,
+                                               illegalLocalVolOverwrite));
+        }
+  }
+};
+
+/////////////////////////// end addition //////////////////////////////
+
+
+
 %{
 using QuantLib::QuantoEngine;
 using QuantLib::ForwardVanillaEngine;
```

programmer works with Python variables and calls Python functions. The QuantLib-Python module automatically translates these into the appropriate C++ functions and converts the C++ objects into Python objects. To the programmer, it's as if the entire QuantLib library had been written in Python instead of C++.

All this magic is accomplished by the SWIG software, which creates wrapper code that converts between Python and C++ data types before and after calling a C++ function. In the simplest situation, the wrapper code does only three things:

- converts all input arguments to the function from Python data types to C++ data types;
- calls the C++ function with the arguments provided as C++ data types; and
- converts the return value from the C++ function from C++ data types to Python data types that the Python programmer can use.

In simple cases, SWIG can parse the C++ source code and generate the wrapper code without much manual intervention. With complex software such as QuantLib, it's necessary to give SWIG considerable guidance on how to build the wrapper. This is done using interface files (which usually have the .i extension). Moreover, when the C++ code uses templates, each instance of the template must be wrapped separately; the interface file often instructs SWIG to wrap only the most common instances of the template.

The QuantLib library comes with a set of interface files that build wrappers for the most important

FD Barrier Python Code

```
1.     #!/usr/bin/env python3
2.     # Requires QuantLib-SWIG with modified options.i
3.     from QuantLib import *
4.     import matplotlib.pyplot as plt
5.     barrier, barrierType, optionType, \
6.         rebate = (80.0, Barrier.DownOut, Option.
           Call, 0.0)
7.     underlying, strike, rf, sigma, maturity, \
8.         divYield = (100, 105, 5e-2, 20e-2, 12, 0.0)
9.     # maturity is in days & must be an integer
10.    Grids = (5, 10, 25, 50, 100, 1000, 5000)
11.    maxG = Grids[-1]
12.    today = Settings.instance().evaluationDate
13.    maturity_date = today + int(maturity)
14.    process = BlackScholesMertonProcess(
15.        QuoteHandle(SimpleQuote(underlying)),
16.        YieldTermStructureHandle(FlatForward(tod
           ay, divYield, Thirty360())),
17.        YieldTermStructureHandle(FlatForward(tod
           ay, rf, Thirty360())),
18.        BlackVolTermStructureHandle(BlackConstantVol(
19.            today, NullCalendar(), sigma, Thirty360())))
20.    option = BarrierOption(barrierType, barrier, rebate,
21.                   PlainVanillaPayoff(optionType, strike),
22.                   EuropeanExercise(maturity_date))
23.    option.setPricingEngine(AnalyticBarrierEngine(process))
24.    trueValue = option.NPV()
25.    uErrors = []
26.    tErrors = []
27.    for Grid in Grids:
28.        option.setPricingEngine(FdBlackScholesBarrierEngine (
29.            process, maxG, Grid))
30.        uErrors.append(abs(option.NPV()/trueValue-1))
31.        option.setPricingEngine(FdBlackScholesBarrierE
           ngine (
32.            process, Grid, maxG))
33.        tErrors.append(abs(option.NPV()/trueValue-1))
34.    plt.loglog(Grids, uErrors, 'r-', Grids, tErrors, 'b--')
35.    plt.xlabel('No of Grid Points (Log Scale)')
36.    plt.ylabel('Relative Error (Log Scale)')
37.    plt.legend(['Asset Grid Points', 'Time Grid Points'])
38.    plt.title('Increasing Asset or Time Grid Keeping the
       Other Grid at ' + str(maxG))
39.    plt.show()
```

QuantLib functions. In most cases, therefore, the programmer doesn't have to worry about the interface files at all. During the installation process, the predefined interface files are used to build the wrapper files and create the QuantLib Python module. The programmer can simply add the line `from QuantLib import *` to the Python code and not worry about C++ and SWIG at all.

In some cases, the predefined interface files might not provide access to a QuantLib function required for a particular task. In our case, the predefined interface files don't offer access to the FD pricing engine for Barrier options. To use this engine in Python code, we must therefore modify the interface file and instruct SWIG to build a wrapper for this engine as well; the sidebar "SWIG options.patch File," shows the lines we must add to the predefined options.i file to achieve this. Rebuilding the wrappers and the QuantLib Python module using this modified file lets Python programmers use the FD pricing engine for Barrier options in Python code. Generally, however, we've found the predefined interface files to be quite comprehensive; we've encountered only a few instances requiring modification of the interface files and module rebuilding.

We now illustrate how QuantLib is used in Python. Our example uses Python 3, but Python 2.7 would work equally well. The sidebar, "FD Barrier Python Code," shows the Python code.

Line 3 in the sidebar (`from QuantLib import *`) has already been discussed; the `import` in line 4 is for plotting. Lines 5 to 9 define the inputs for the Barrier option and are pure Python code—except that they refer to two constants (`Barrier.DownOut` and `Option.Call`) defined in QuantLib. We define a Down-and-Out (DO) Call option with a barrier at 80 and no rebate. The current market price of the underlying security is 100, the strike price of the option is 105, the risk-free rate is 5 percent, the volatility is 20 percent, and the maturity is 12 days.

Lines 10 and 11 define the different grid spacing parameters that we'll use for the FD method. The number of grid points for the space (asset price) dimension and time dimension range from 5 to 5,000. Lines 12 and 13 set the maturity date. `Settings.instance().evaluationDate` is a QuantLib global variable that defines the date on which the evaluation is done. Lines 14–19 set up the stochastic process. This function takes the following arguments: the current market price of the underlying, the dividend yield, the risk-free rate, and the volatility. The last three arguments are converted into instances of the TermStructure class discussed earlier. The first argument must be converted into a QuoteHandle, which is a QuantLib class used for market quotes.

Lines 20–22 set up the Barrier option itself. The last two arguments to this function use the Payoff class and Exercise class discussed earlier. In lines

23 and 24, we find the true (analytic) value of the option by first setting the analytic pricing engine and then computing the option's net present value (NPV). Lines 25–33 perform the FD valuation of the Barrier option for two different sets of the space and time grids. First, we keep the time grid fixed at the maximum value of 5,000 and vary the space grid from 5 to 5,000. The pricing engine is set to the FD engine, the option is valued, and the pricing error (relative to the true analytic value) is calculated and stored in uErrors. Second, we keep the space grid fixed at the maximum value of 5,000 and vary the time grid from 5 to 5,000. The pricing error is calculated and stored in tErrors.

At the end, we plot the results on a log–log scale (line 34); the rest of the code sets up the titles and legend for the plot. The code described in this section, along with the associated IPython notebook and description, are available on our GitHub page (https://github.com/jrvarma/fdbarrier).

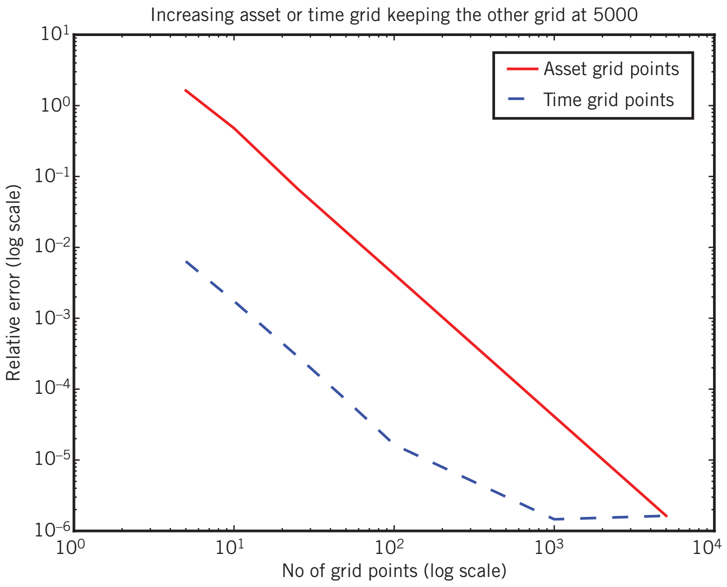

Figure 1. Python code graph. The graph shows how the pricing accuracy of the FD method depends on the number of asset price grid points and the number of time grid points.

Results and Extensions

Figure 1 shows the graph produced by the Python code, which can be used to advance several pedagogical purposes. First, the red line (dependence of pricing accuracy on the asset price grid) conforms to the theoretical prediction of a straight line in a log–log plot. Second, the initial segment of the blue line (dependence of pricing accuracy on the time grid) is also a straight line, in accordance with theoretical predictions. However, the later segment of the curve is almost flat. Again, this is typically observed in practice: it's usually optimal to have a significantly finer grid on the asset price than on the time dimension. Increasing the time grid points without simultaneously increasing asset grid points to a significantly larger value is often futile.

One question that will ultimately arise is, "What's the point of using FD to valuate an instrument that already has an analytic pricing formula?" The answer is that the FD method will work under alternative assumptions, for which no analytic formula exists—such as when we change the stochastic process from the GBM or Black-Scholes process to a stochastic volatility or Heston-type process. The FD method works equally well for a wide variety of stochastic processes. The results of the GBM case can guide us in choosing the appropriate number of grid points to compute the option value.

Given the sophistication of pricing models used at financial institutions, reliance on numerical methods is unavoidable. Aside from perhaps the OpenGamma platform (www.opengamma.com/opengamma-_platform) for margining and market-risk management applications, no other open source computing library currently exists that allows us to capture real-world pricing issues while sitting in the ivory tower. Although setup costs might be high, the time spent learning the structure of QuantLib is well worth it.

At the risk of being overly optimistic, our hope is that this article will nudge some students and practitioners of financial engineering and computational finance to ditch Microsoft Excel and other proprietary software in favor of QuantLib-Python. ■

References

1. F. Black and M.S. Scholes, "The Pricing of Options and Corporate Liabilities," *J. Political Economy*, vol. 81, no. 3, 1973, pp. 637–654.
2. F. Black, "How We Came Up with the Option Formula," *J. Portfolio Management*, vol. 15, no. 2, 1989, pp. 4–8.
3. R.C. Merton, "Theory of Rational Option Pricing," *Bell J. Economics*, vol. 4, no. 1, 1973, pp. 141–183.
4. S. Shreve, *Stochastic Calculus for Finance II: Continuous Time Models*, New Age International, India, 2007.
5. D. Duffy, *Finite Difference Methods in Financial Engineering: A Partial Differential Equation Approach*, Wiley, 2006.

6. P. Glasserman, *Monte-Carlo Methods in Financial Engineering*, Springer, 2003.
7. N. Firth, *Why Use QuantLib?* tech. report, Mathematical Inst., Oxford Univ., 2004.
8. L. Ballabio, *Implementing QuantLib: A Case Study in C++ for Quantitative Finance*, Leanpub, 2015.
9. S. Heston, "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options," *Rev. Financial Studies*, no. 6, vol. 2, 1993, pp. 327–343.
10. E.G. Haug, *The Complete Guide to Option Pricing Formulas*, McGraw-Hill, 2007.
11. P. Wilmott, *Paul Wilmott on Quantitative Finance—3 Volume Set*, Wiley, 2006.

Jayanth R. Varma is a professor of finance and accounting at the Indian Institute of Management, Ahmedabad. His research interests include financial markets and their regulation, mathematical modeling, and computer simulation. Varma received a PhD in management from the Indian Institute of Management, Ahmedabad. Contact him at jrvarma@iimahd.ernet.in.

Vineet Virmani is an assistant professor of finance and accounting at the Indian Institute of Management, Ahmedabad. His research interests include computational finance and risk management. Virmani received a PhD in management from the Indian Institute of Management, Ahmedabad. Contact him at vineetv@iimahd.ernet.in.