Computational Finance Using QuantLib-Python
©2016 IEEE

Jayanth R. Varma
Vineet Virmani

This is the ***accepted version*** of the following paper (Copyright ©2016 IEEE):

- J. R. Varma and V. Virmani, "Computational Finance Using QuantLib-Python," in *Computing in Science & Engineering,* vol. 18, no. 2, pp. 78-88, Mar.-Apr. 2016. doi:10.1109/MCSE.2016.28

The ***published version*** of this paper is available at

- http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7426276

An ***earlier version*** of this paper was brought out as a Working Paper at the Indian Institute of Management, Ahmedabad:

- J. R. Varma and V. Virmani, "Derivatives Pricing using QuantLib: An Introduction", IIMA W.P. No. 2015-03-16, April 2015

Abstract

Given the complexity of over-the-counter derivatives and structured products, almost all of derivatives pricing today is based on numerical methods. While large financial institutions typically have their own team of developers who maintain state-of-the-art financial libraries, till a few years ago none of that sophistication was available for use in teaching and research. For the last decade, there is now a reliable C++ open-source library available called `QuantLib`. This note introduces `QuantLib` for pricing derivatives and documents our experience using its Python extension, `QuantLib-Python`, in our course on Computational Finance at the Indian Institute of Management Ahmedabad. The fact that it is also available in Python has allowed us to harness the power of C++ with the ease of IPython notebooks in the classroom as well as for student's projects.

Keywords

Derivatives pricing, Financial engineering, Open-source computing, Python, QuantLib

Introduction

Financial engineering and algorithmic trading are perhaps two of the most computationally intensive parts of all of finance. A job in either of these areas requires not only a reasonable expertise in finance, mathematics and statistics, but also, and perhaps more importantly today, sophistication in computing.

While algorithmic trading is about finding opportunities in markets that give a temporary statistical edge - so the focus is on speed and data-mining, the field of financial engineering primarily concerns itself with pricing and managing of derivatives (also called structured products) designed to meet specific requirements of large banks and corporations.

The pricing of such products rely on the same principles as those underlying the famous Nobel-prize winning Black-Scholes formula. Even though 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.

While the formula is only applicable for the most simplest of derivatives (plain-vanilla Call and Put options), 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 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.

While theoretically the two approaches are 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. While large banks typically have their own team of ‘quants’ and IT developers hired to implement sophisticated FD and MC engines, many boutique firms have spawned in the last decade who provide such specialized services/software for a fee to other banks and organizations.

Till a few years ago, none of that sophistication was available for use in teaching and research. Neither banks nor the boutique firms share their proprietary software, and if at all they are available, they are either prohibitively expensive or downright useless. For the last few years, however, there is now a reliable open-source library available called `QuantLib`, built in C++ and also available in Python, Ruby, R and Excel among others. This note introduces `QuantLib` for computational finance applications in teaching and research, along with a ‘worked-out’ example. The fact that it is also available (and extendable) in Python allows one to harness the power of C++ with the ease of IPython notebooks.

The Pricing Problem

Market for financial derivatives worldwide today is so large that the size of outstanding positions (at almost 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 pre-specified level (called the 'barrier') before the expiration.

The pricing problem for such derivatives typically constitutes working with a stochastic differential equation (SDE) for the price process (S_t), like 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 that the diffusion coefficient $\sigma(t, S_t)$ for the Brownian motion is deterministic (as taken to be in this note), 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 condition/s coming from the specification of the product being priced.

The Merton argument, on the other hand, is based on showing the existence of 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 to be understood as the expectation corresponding to the risk-neutral measure.

The two approaches may look different and seem to be leading to different solutions, but that is not the case. The Feynman-Kac theorem for parabolic PDEs ensures that they are theoretically consistent.

Numerical methods

For both 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 the Douglas methods.

The state-of-the-art in simulation-based methods constitutes working with Sobol sequences and low discrepancy numbers. It is also common to use variance reduction methods whenever possible, of which antithetic and control variates are the most popular. Often in most practical applications it is clear which of the two approaches is better suited for a job, but in general wherever the curse of dimensionality is high (number of assets more than or equal 4), the MC method is preferred. When pricing products like barrier options, where the value of the underlying has to be monitored regularly (often 'continuously'), FD methods are more popular (as one can easily enforce the location of the barrier to lie on the FD grid).

QuantLib

The QuantLib open-source project was started in the year 2000 at the Italian boutique risk-management firm RiskMap (now called StatPro Italia). QuantLib website (www.quantlib.org) states the aim of the project as "...providing a comprehensive software framework for quantitative finance."

The first QuantLib package was released in December, 2000 under a liberal BSD license. This has allowed the banks and software companies to extend and modify the code without having to release it back. While the original developers, Luigi Ballabio and Ferdinando Ametrano, remain involved in the development and maintenance of the library, the project today has more than 150 contributors, with some of them making substantial contributions. Though financially StatPro continues to support the project (while also using it for consulting and training), growth and quality of QuantLib has been driven mainly by the contributions of the open-source financial software community (see the QuantLib github page for the list of contributors: 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 pre-requisite after July, 2004). It also makes an extensive use of SWIG (Simplified Wrapper and Interface Generator), and bindings exist for a variety of languages including Python, R, Ruby, Excel, C# and others though not all of the are equally well-developed at this stage (Python being one of the most popular and developed).

There is a host of information on the project page for any novice to get started (though it

does assume a working knowledge of both C++ and quantitative finance), starting with the help on installation (quantlib.org/install.shtml) and video examples with IPython notebooks (vimeo.com/channels/qlnotebooks) to a detailed reference manual (quantlib.org/reference/index.html) and a blog (www.implementingquantlib.com) maintained by Ballabio. Nonetheless, despite the available information, we have found that users migrating from Microsoft Excel and the like can get intimidated with the steps involved. In Box A, we provide step-by-step installation and set-up instructions for the latest version of `QuantLib` and `QuantLib-Python` for both Linux (Ubuntu) and Windows operating systems.

The way it is developed, `QuantLib` is completely object-oriented and makes extensive use of design patterns. Even if someone is not developing models, it is a good example to learn the use of design patterns when building a financial library.

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

Structure of `QuantLib`: Important classes

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

- Price of the underlying securities as on date of pricing and their feeds
- Term structure of interest rates, volatility, inflation and default probabilities

- Cash flows (including coupons and dividends) from the instrument
- Stochastic process for the underlying
- Pricing engine (the numerical method used for pricing)

Now if one is pricing a stand-alone product for a 'student project', one could just code the above elements in a single monolithic program to find the price, and it is not altogether necessary to use an object-oriented approach like `QuantLib`.

In a large financial institution or a hedge fund, however, derivatives are part of a larger portfolio and then this monolithic approach quickly becomes inefficient and impractical. The beauty of `QuantLib` is that in its spirit and scope, it is very similar to financial library that one finds in a large bank. It then becomes natural to use `QuantLib` for computational finance even in the classroom, as it helps students come up to speed with the state-of-the-art quickly. 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).

The Instrument class

The Instrument class is designed keeping in mind that as the value of the underlying security changes, so does the value of a instrument. It is then required to maintain 'links' so that when called they would access most current values of the underlying. 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 have changed. In `QuantLib` this is operationalized using the Observer design pattern, where instrument plays the role of the 'observer' and inputs that of 'observables'.

The TermStructure class

It is 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, its job is to:

- Keep track of its 'own' (reference) date and calculate the appropriate future date if reference date is different from the evaluation date (by using number of business days forward)

- Convert dates to times (say, when converting discount factors to zero-yields)

- Check whether a given date/time belongs to the domain covered by the term structure (setting maximum allowed date/range, for example, when fitting volatility term structures)

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

The Payoff and the Exercise class

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

For derivatives with early-exercise feature, a crucial property is the exercise possibilities available. In `QuantLib` this is operationalized using the Exercise base class, which depending on exercise choices available, can be accessed as one of EuropeanExercise

(only at expiration) or BermudanExercise (on a set of discrete dates prior to expiration) or AmericanExercise (any time before expiration) classes.

The StochasticProcess class

All derivatives pricing begins with the assumption of an appropriate stochastic process for the underlying and volatility. For example, the Black-Scholes model begins with a GBM for the stock price, and the Heston model 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 which handles how the process is passed into the pricing engine, of which the EulerDiscretization is the most important one.

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 may need be priced for its own sake, or to derive implied volatilities or for calibrating a stochastic volatility model. And any of these may need to be done via FD or Monte-Carlo methods. In `QuantLib` this is operationalized using the PricingEngine class which is modeled as the Strategy pattern, in which instruments takes an object encapsulating the computation to be performed. This allows the instrument to be priced using any of the various engines available in the library.

For reasons of implementation, most pricing engines do not descend from the PricingEngine class, but 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,

including Analytic, FD and MC methods are available.

Although there are other important classes implementing processes like random number and path generation (RandomSequenceGenerator and Path classes), calibration (the CalibrationHelper and CalibratedModel classes) and models like 'trees' (the Lattice and DiscretizedAsset classes), again, since they aren't discussed in our example we leave the details for these to the relevant chapters of the `QuantLib` reference manual and the Ballabio book.

Example: Pricing Barrier option using FD

Even though barrier option is a simple enough product, and its correct value is available as a closed-form formula, it is perfectly suited to illustrate the use of `QuantLib` and `QuantLib-Python` as it allows one 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 FD method implemented in `QuantLib`. The fact that its value is available in analytically, also allows us to compute the pricing error of the FD scheme.

A barrier option (colloquially referred to simply as 'barriers') is a derivative whose payoff depends on whether the price of the underlying security crosses a pre-specified level before the expiration. It comes in two basic varieties:

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

Barriers are also typically categorized in relation to the current value of the underlying. So if the barrier level of a knock-out option is set to a value below (above) the current value of the asset, it is referred to as a 'Down-and-Out' ('Up-and-Out') option. The 'in' options are defined similarly. In plain-vanilla barriers, the payoff at expiration could be either of a Call or a Put variety. Putting all variants together, then, there exist 8 different combinations of plain-vanilla barrier options.

To mathematically represent the payoff of a barrier option, it is convenient to call m_T and M_T as the minimum and maximum value respectively of the asset between the evaluation (t) and expiration (T) date as:

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

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

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

where $I[m_T > B_d]$ represents an indicator variable which 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 written as:

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

where $I[M_T > B_u]$ 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 an UO Call option V_t is given

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 associated boundary conditions as:

$$\begin{array}{ll} V(T, B_d) = 0 & 0 \leq t \leq T \\ V(T, S_T) = (S_T - K)^+ & S_T > B_d \end{array}$$

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

While discretizing the partial derivatives, whether one does backward or forward difference turns out to be important. The Explicit FD scheme uses backward difference in , and the Implicit FD scheme uses the forward difference in t . Borrowing the notation from Paul Wilmott's standard text (Chapter 77), 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 'backwards' from expiration to evaluation date.

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

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

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

and in the Implicit FD scheme they are discretized as:

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

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

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

$$\frac{\partial V}{\partial t} \approx \frac{V_{i}^{k+1} - V_{i}^{k+1}}{\delta t}$$

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

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

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

Note that $\theta = 1$ gives back the Explicit scheme, and $\theta = 1$ gives back the Implicit scheme. With $\theta = 1/2$, one gets the scheme known as Crank-Nicholson method.

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

Implementation

By now we have all the necessary set-up to talk about 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 is not necessary to know anything about the C++ language at all. The Python programmer works with Python variables and calls Python functions. The `QuantLib-Python` module automatically translates these into the appropriate C++ function and converts the C++ objects into Python objects. To the programmer, it is 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:

1. All the input arguments to the function are converted from Python data types to C++ data types

2. The C++ function is called with the arguments provided as C++ data types

3. The return value from C++ function is converted from C++ data types to Python data types which the Python programmer can use.

In simple cases, SWIG can parse the C++ source code and generate the wrapper code without too much manual intervention. In the case of a complex software like `QuantLib`, it is 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, where the C++ code uses templates, each instance of the template must be wrapped separately, and often the interface file 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 `QuantLib` functions. In most cases, therefore, the programmer does not have to worry about the interface files at all. During the installation process, the predefined interface files would be 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 may not provide access to a `QuantLib` function that is required for the specific task at hand. In our case, the predefined interface files do not provide access to the FD pricing engine for barrier options. To use this engine in Python code, it is necessary to modify the interface file and instruct SWIG to build a wrapper for this engine also. The patch file in Box B shows the lines to be added to the predefined options.i file to achieve this. Rebuilding the wrappers and the `QuantLib`

Python module using this modified file allows the Python programmer to use the FD pricing engine for barrier options in Python code.

In our experience, though, the predefined interface files are quite comprehensive and we have encountered only a couple of instances requiring modification of the interface files and rebuilding the module.

Now we illustrates how `QuantLib` is used in Python. This example used Python 3, but Python 2.7 would work equally well. Box C has the Python code listing.

Line 3 in the listing (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 are defining a Down-and-Out Call option with a barrier at 80 and no rebate. The current market price of the underlying is 100, the strike price of the option is 105, the risk free rate is 5%, the volatility is 20% and the maturity is 12 days.

Lines 10 and 11 define the different grid spacing parameters that we will use for the finite difference method. The number of grid points for the space (asset price) dimension and time dimension range from 5 to 5000.

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 to 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 has to be converted into a QuoteHandle which is a

`QuantLib` class used for market quotes.

Lines 20 to 22 set up the barrier option itself. It may be observed that the last two arguments to this function use the payoff class and the 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 NPV of the option.

Lines 25 to 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 5000 and vary the space grid from 5 to 5000. 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 5000 and vary the time grid from 5 to 5000. 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 also available on our GitHub page at https://github.com/jrvarma/fdbarrier.

Results and extensions

[Figure 1 about here]

The graph produced by the Python code above is shown in Figure 1. This graph can be used to advance several pedagogical purposes. First the red line (dependence of pricing accuracy on 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 time grid) is also a straight line in accordance with theoretical predictions. But the later segment of the curve is almost flat. This is again generally observed in practice: it is usually optimal to have 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 arise at the end is what is the point of doing valuation using FD of an instrument for which there is an analytic pricing formula. The answer is that the FD method will work under alternative assumptions where there is no analytic formula. For example, if change the stochastic process from the GBM or Black-Scholes process to a stochastic volatility or Heston-type process, there is no analytic formula. But the FD method will work equally well even in this case. The results of the GBM case can guide us in choosing the appropriate number of grid points to compute the option value.

Conclusion

This note has introduced `QuantLib` and `QuantLib-Python` for pricing derivative securities in practice. Given the sophistication of pricing models used at financial institutions, reliance on numerical methods is unavoidable. There is currently no other open-source computing library, barring the OpenGamma platform perhaps (see www.opengamma.com/opengamma-_platform) designed for margining and market-risk management applications, which allows capturing real-world pricing issues sitting in the ivory tower as `QuantLib` does. While the set-up cost may be high, the time learning the structure of `QuantLib` is well worth it.

Once the framework of `QuantLib` is clear, one need not even know C++. Thanks to SWIG, one can simply work with `QuantLib-Python`, and a programmer need only know Python well. Even though the predefined interface of `QuantLib` with Python via

SWIG files may not be complete, in our experience it is comprehensive enough that requiring modification of the interface and rebuilding `QuantLib-Python` is quite infrequent.

At the risk of being overly-optimistic, hopefully this note will act as a nudge for some students and practitioners of financial engineering and computational finance to ditch Microsoft Excel and other proprietary software in favour of `QuantLib-Python`.

References

Ballabio, L. (2015). *Implementing QuantLib: A Case Study in C++ for Quantitative Finance*. Leanpub.

Black, F. (1989). How we came up with the option formula. *Journal of Portfolio Management*, 15(2):4–8.

Black, F. and Scholes, M. S. (1973). The Pricing of Options and Corporate Liabilities. *Journal of Political Economy*, 81(3):637–54.

Duffy, D. (2006). *Finite Difference Methods in Financial Engineering: A Partial Differential Equation Approach*. Wiley.

Duffy, D. and Kienitz, J. (2009). *Monte Carlo Frameworks: Building Customisable High-performance C++ Applications*. Wiley.

Firth, N. (2004). Why use QuantLib? Technical report, Oxford University.

Glasserman, P. (2003). *Monte-Carlo Methods in Financial Engineering*. Springer.

Haug, E. G. (2007). *The Complete Guide to Option Pricing Formulas*. McGraw-Hill.

Heston, S. (1993). A closed-form solution for options with stochastic volatility with applications to bond and currency options. *Review of Financial Studies*, 6(2):327–343.

Merton, R. C. (1973). Theory of Rational Option Pricing. *Bell Journal of Economics*, 4(1):141–183.

Shreve, S. (2007). *Stochastic Calculus for Finance - II: Continuous Time Models*. New Age International, India.

Wilmott, P. (2006). *Paul Wilmott on Quantitative Finance - 3 Volume Set*. Wiley.

A Box: Installation Instructions for `QuantLib` and `QuantLib-Python`

For `QuantLib-Python` to work it is necessary to first have `QuantLib` working. After listing out the common prerequisites, we provide step-by-step instructions separately for Ubuntu (12.04 and above) and Windows (7 and above).

Prerequisites

- For installing in Windows, a working C++ environment is required. For the purposes of instructions here (meant for a novice), it is recommended that the user install the free Microsoft Visual Express Desktop 2013 edition which comes with Visual C++ 12 (MSVC12). It is equally easy to install in Windows on Cygwin, or using MinGW, but then a user familiar with those would probably not need this user-guide.

- `QuantLib-Python` requires a working Python environment, and we recommend working with the Anaconda meta-package. 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. It is recommended to install Anaconda with all the default settings, and update it by running `conda update conda` and `conda update anaconda`. Again, it does not matter if one installs Anaconda-2.x (with Python 2.7) or Anaconda-3.x (with Python 3), but it is recommended that the user install 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

which are different for Ubuntu and Windows and is provided below.

- One should download the same version of `QuantLib` and `QuantLib-Python` (http://sourceforge.net/projects/quantlib/files/quantlib/1.6/).

`QuantLib-Python` requires a working `QuantLib`, so the user should follow the instructions in the order as given below.

Installing in Ubuntu

For Ubuntu it is recommended not to install `QuantLib` from the `synaptic` package manager as Ubuntu repositories do not contain the latest version.

- First step is installing the Boost C++ libraries, and in Ubuntu they are available from the repositories (package name `libboost-all-dev`)
- After installing Boost, user should install SWIG, also available from the repositories (package name `swig`)
- Instructions for installing `QuantLib` for Ubuntu are available from the `QuantLib` project page at quantlib.org/install/linux.shtml. Before proceeding further, the user 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 that order):
  - `cd \path\to\QuantLib-SWIG-1.6\Python`
  - `python setup.py wrap`
  - `python setup.py build`

- `python setup.py test`
- `sudo python setup.py install`

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

If user-defined `.i` SWIG files have been added (or existing files have been modified), to ensure that they are available, the last four steps (from wrap to install) must be repeated.

Installing in Windows

If the Windows user is working with a free version of Visual Studio, it should be kept in mind to install the 32-bit versions of both Anaconda and Boost as free Visual Studio lacks the necessary toolkit for building `QuantLib-Python` for 64-bit (there are ways to make the free version work in 64-bit, but they are not guaranteed to be replicated universally). Those with access to the professional Visual Studio environment may 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` should be taken to mean `C:\boost\boost_1_58_0`.

- For Windows, pre-packaged binaries for boost are available (latest version 1.58.0) for specific versions of MSVC (in our case MSVC12) from its sourceforge page. User should download the executable for the 32-bit architecture (http://sourceforge.net/projects/boost/files/boost-_binaries/1.58.0/).
- After installing Boost, user should install SWIG. Pre-packaged binaries for SWIG for Windows are available from the source page (www.swig.org/download.html),

and it is enough to extract the SWIG zip file (latest version `swigwin-3.0.5`) in a convenient folder.

- All the commands below assume that we are working in Visual Studio command prompt (i.e. all the relevant Visual Studio related environment variables have been set). It can be launched from the Start menu (in Windows 7) or from Apps (in Windows 8). Alternatively, one can find how to locate it at https://msdn.microsoft.com/en-_us/library/ms229859(v=vs.110).aspx

- `QuantLib` in Windows can directly be installed in three simple steps after launching the Visual Studio command prompt (mind the gaps):
    - `cd \path\to\QuantLib-1.6`
    - `set myboost=\path\to\boost`
    - `msbuild /p:AdditionalLibPaths="%myboost\lib32-msvc-12.0" /p:Configuration=Release /p:Platform=Win32 QuantLib_vc12.sln`

- The last step above may take a while (on an Intel i5, Windows 7 machine with 4 GB RAM it took almost 45 minutes), so it is advisable to have a couple of hours at hand when sitting down to install `QuantLib`.

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

`PATH=\path\to\Anaconda3;\path\to\Anaconda3\scripts;\path\to\swigwin-3.0.5;\path\to\QuantLib-1.6\lib;%PATH%`

    - `set QL_DIR=\path\to\QuantLib-1.6`
    - `set VS100COMNTOOLS=%VS120COMNTOOLS%`
    - `set INCLUDE=\path\to\boost_1_58_0;%INCLUDE%`
    - `set LIB=\path\to\boost\lib32-msvc-12.0;%LIB%`
    - `echo [build] > setup.cfg && echo compiler=msvc >> setup.cfg`

- The last few steps are identical to that in Ubuntu:
  - `cd \path\to\QuantLib-SWIG-1.6\Python`
  - `python setup.py wrap`
  - `python setup.py build`
  - `python setup.py test`
  - `python setup.py install`

If user-defined `.i` SWIG files have been added (or existing files have been modified), to ensure that they are available, one needs to additionally go through the 'set' commands before running the last four steps (from wrap to install).

B Box: SWIG options.patch file

```
1. --- options.i 2015-01-16 16:22:45.000000000 +0530
2. +++ modified-options.i 2015-02-23 13:30:23.096148000 +0530
3. @@ -1099,6 +1099,63 @@ class MCBarrierEnginePtr : public boost:
4.      }
5.  };
6.
7. +///////////////// added for finite difference barrier valuation /////////
8. +
9. +%{
10. +using QuantLib::FdmSchemeDesc;
11. +%}
12. +
13. +struct FdmSchemeDesc {
14. +  enum FdmSchemeType { HundsdorferType, DouglasType,
15. +                       CraigSneydType, ModifiedCraigSneydType,
16. +                       ImplicitEulerType, ExplicitEulerType };
17. +
18. +  FdmSchemeDesc(FdmSchemeType type, Real theta, Real mu);
19. +
20. +  const FdmSchemeType type;
21. +  const Real theta, mu;
22. +
23. +  // some default scheme descriptions
24. +  static FdmSchemeDesc Douglas();
25. +  static FdmSchemeDesc ImplicitEuler();
26. +  static FdmSchemeDesc ExplicitEuler();
27. +  static FdmSchemeDesc CraigSneyd();
28. +  static FdmSchemeDesc ModifiedCraigSneyd();
29. +  static FdmSchemeDesc Hundsdorfer();
30. +  static FdmSchemeDesc ModifiedHundsdorfer();
31. +};
32. +
33. +%{
34. +using QuantLib::FdBlackScholesBarrierEngine;
35. +typedef boost::shared_ptr<PricingEngine> FdBlackScholesBarrierEnginePtr;
36. +%}
37. +
38. +%rename(FdBlackScholesBarrierEngine) FdBlackScholesBarrierEnginePtr;
39. +class FdBlackScholesBarrierEnginePtr : public boost::shared_ptr<PricingEngine>
    {
40. +  public:
41. +    %extend {
42. +        FdBlackScholesBarrierEnginePtr(const GeneralizedBlackScholesProcessPtr&
     process,
43. +          Size tGrid = 100, Size xGrid = 100, Size dampingSteps = 0,
44. +          const FdmSchemeDesc& schemeDesc = FdmSchemeDesc::Douglas(),
45. +          bool localVol = false,
46. +          Real illegalLocalVolOverwrite = -Null<Real>()) {
47. +            boost::shared_ptr<GeneralizedBlackScholesProcess> bsProcess =
```

```
48. +                 boost::dynamic_pointer_cast<GeneralizedBlackScholesProcess>(
49. +                                                                               process);
50. +            QL_REQUIRE(bsProcess, "Black-Scholes process required");
51. +            return new FdBlackScholesBarrierEnginePtr(
52. +                            new FdBlackScholesBarrierEngine(bsProcess,
53. +                                                tGrid,  xGrid, dampingSteps,
54. +                                                schemeDesc, localVol,
55. +                                                illegalLocalVolOverwrite));
56. +        }
57. +  }
58. +};
59. +
60. +/////////////////////////// end addition ///////////////////////////////
61. +
62. +
63. +
64.  %{
65.  using QuantLib::QuantoEngine;
66.  using QuantLib::ForwardVanillaEngine;
```

C Box: 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(today, divYield, Thirty360())),
17.     YieldTermStructureHandle(FlatForward(today, 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(FdBlackScholesBarrierEngine (
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()
```

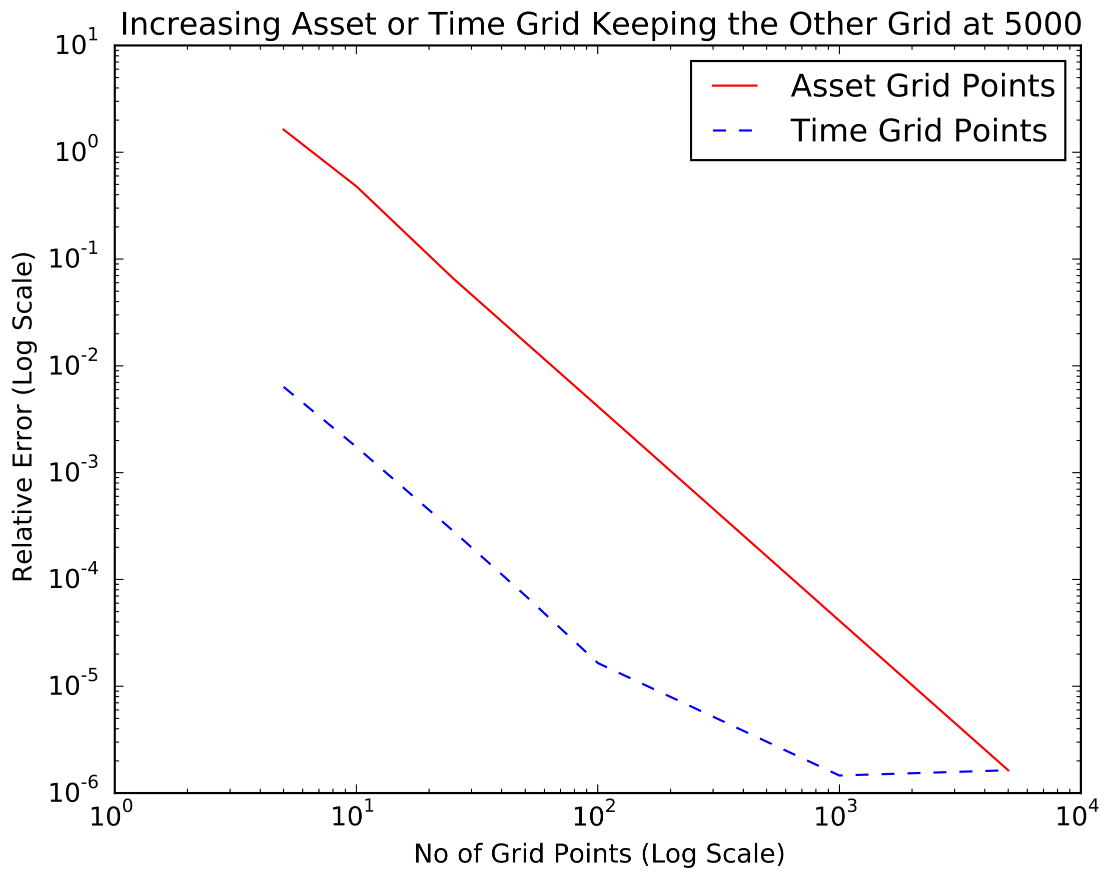

Figure 1: Dependence of Pricing Accuracy on Number of Asset Price Grid Points and Time Grid Points