Fast and flexible libor model pricing: two-stage Monte Carlo and on-the-fly payoff processing

M. Auer & S. Biffl
Institute of Software Technology and Interactive Systems,
Vienna University of Technology, Austria

Abstract

The Libor market model is the standard interest rate model. Yet its application relies on Monte Carlo simulation, which is slow, especially in a flexible, product-independent model setup.

This paper proposes an alternative software design of the Monte Carlo simulation to achieve fast and flexible pricing. The design is fast because it separates the computation of forward rate paths from that of payoffs to avoid redundant calculations; it is flexible because it adapts to new types of payoff functions at run-time via on-the-fly compilation.

The approach is arbitrarily accurate—it supports a high discretization resolution and even full factors without affecting the response time. It also features a small-footprint software design that is cheap to maintain and product-independent.
Keywords: Libor market model, Monte Carlo simulation, derivative pricing, user interface responsiveness.

1 Introduction

The Libor market model [1, 7] is the standard interest rate model. Its application requires Monte Carlo simulation, and thus some sort of software implementation. Ideally, this software should provide fast, accurate results, at low setup and maintenance costs, and independent of product types. Yet these are conflicting goals.

Consider a conventional software implementation of the Libor model, like a Euler scheme applied to the logarithm of the forward rates. It provides arbitrarily accurate results, and it is straightforward and cheap to set up and to maintain, but it will have too slow a response time for traders who need quotes fast.

WIT Transactions on Information and Communication Technologies, Vol 41, © 2008 WIT Press
www.witpress.com, ISSN 1743-3517 (on-line)
doi:10.2495/CF080031

One way to speed it up is to sacrifice some accuracy. If, for example, a higher standard error is acceptable, the number of Monte Carlo runs can be reduced; if a low-factor approximation to a correlation structure is acceptable, the number of operations per Monte Carlo run can be reduced.

Other approaches do not compromise the accuracy, but they increase the size and complexity of the software implementation, and thus its setup and maintenance costs. One example is to parallelize the implementation, which is very suitable for Monte Carlo type models [8], as Monte Carlo simulation is an "embarrassingly parallel" computational task. Alternatively, a variety of variance reduction techniques can reduce the number of Monte Carlo runs required to obtain a desired accuracy [6]. Drift correction, in turn, allows reduction of the discretization resolution and thus the number of operations per Monte Carlo run. Yet these approaches increase the setup costs: they require additional hardware and/or the implementation of additional source code. But more importantly, they also increase the dominant maintenance costs: larger and more complex source code is more expensive to document, learn, modify, test, port and maintain. (An overview of such optimization methods is given in [5].)

Finally, there are some product-specific optimizations. For example, some financial products only rely on certain rates and fixing dates, which makes jump procedures feasible [9]. Such product-specific code—in addition to increasing code size and maintenance costs—makes the model implementation unsuitable to flexibly adapt to new product types.

In short, the goal of a fast and accurate model that is cheap to set up and to maintain, and product-independent, is difficult to achieve in a conventional Monte Carlo implementation.

This paper proposes an alternative software design that overcomes some of these limitations. The design specifically tackles two issues: (a) provide a very fast response time for model users, and (b) allow for product-independent pricing, i.e., process arbitrary, unforeseen payoff function types at run-time.

The former is achieved by separating the computational effort of the Monte Carlo simulation in two parts: the costly forward rate evolution and the elementary payoff aggregation. This avoids redundant computations in the typical case where many products have to be priced.

The latter issue—adapting to arbitrary new types of payoff functions at run-time without changes to the software implementation—is addressed with an on-the-fly compilation of the payoff functions.

The approach does not compromise accuracy: in fact, the approach easily allows for arbitrarily accurate results (i.e., using millions of Monte Carlo paths, a fine-grained discretization, and potentially full factors (Schoenmakers [10], e.g., illustrates the disadvantages of low factor models)), without affecting the response time.

Also, the source code can be implemented in an unoptimized, straightforward way, minimizing its size and complexity, and thus reducing its maintenance costs.

To sum up, the approach achieves arbitrarily accurate pricing with a response time of seconds (even when using full factors), with a small-footprint software design that is cheap to maintain and flexible, i.e., not tailored to any specific product.

However, there are a few drawbacks. The approach does not help with the inverse problem of parameter calibration. If products' tenor dates don't correspond to the model ones, interpolation is required during simulation. The approach also increases the setup costs due to the recommended 64-bit computer architectures. Finally, intraday calibration might require some degree of parallelization, partially offsetting the achieved software size/maintenance cost reductions.

1.1 Implications for practitioners

The proposed software design for implementing the Libor model has implications regarding two main concerns of decision makers: the usability of a model, and the costs of creating and maintaining it. First, our approach provides fast response times for end users like traders—it is user-friendly. (Note: several major financial data providers still refrain from providing the Libor model due to the problem of slow response times with traditional Monte Carlo.) Second, our approach is cheap: it runs on lean hardware, and requires no expensive optimization of the source code. Especially the maintenance costs can be cut, as graduate-level software engineers are perfectly able to implement and maintain the system; whereas hand-optimized Libor models might well require PhD-level mathematicians.

Section 2 illustrates the background and the benefits of the proposed approach in greater detail. Section 3 describes the implementation details and recommends some programming idioms to keep the software design transparent. Section 4 outlines some drawbacks of the approach. Section 5 concludes and outlines possible improvements.

2 Approach and benefits

The Libor model describes the arbitrage-free evolution of forward rates. Sticking to the notation of Glasserman [6], a (forward measure type) Libor model evolves according to

$$\begin{aligned}\frac{dL_n(t)}{L_n(t)} &= -\sum_{j=n+1}^{M}\frac{\delta_j}{1+\delta_j L_j(t)}\vec{\sigma}_n(t)^\top\vec{\sigma}_j(t)L_j(t)dt\\ &+ \vec{\sigma}_n(t)^\top dW(t)\end{aligned}$$

The numeraire under the forward measure, required for the appropriate discounting, is given by

$$B_{M+1}(t) = B_{\eta(t)}(t)\prod_{j=\eta(t)}^{M}\frac{1}{1+\delta_j L_j(t)}$$

To actually price a product in this model setup using Monte Carlo, numeraire-adjusted payoff functions must be averaged:

$$p(0) = B_{M+1}(0)E^{Q_{M+1}}\left[\frac{p(T)}{B_{M+1}(T)}\right]$$

So there are two parts in Monte Carlo pricing: evolving the forward rates (and the numeraire), and aggregating the payoffs. In the typical case where many products have to be priced, the first part is redundant and should be computed only once. This leads to the following separation:

1. Evolve and store all forward rate paths. All forward rate paths, as well as the corresponding numeraire paths, are pre-computed and stored. This is computationally expensive, but it is done only once for every new Libor model calibration, or when a new forward rate snapshot has to be used as initial condition.

2. Aggregate payoffs for each product. Given the pre-computed forward rate paths, the pricing of products boils down to averaging numeraire-adjusted payoff functions of the stored forward rates. This is computationally cheap.

Separating the two parts of the computation makes the pricing of individual products fast—prices are obtained within seconds. But it has an additional advantage: if there is enough time to perform the evolution of the forward rates, it is possible to have a high-accuracy model while keeping the corresponding software design very simple. It is, e.g., possible to use a high discretization resolution, and a large number of Monte Carlo runs. This increases the accuracy but will not impact the speed of the pricing step, which accesses the same, just more accurately pre-computed, forward rate paths. Also, it is possible to simply use a full-factor model as opposed to a reduced-factor one [3] without affecting the pricing speed.

For all this, a simple software design can be used (object-oriented data structures, no obscure and error-prone optimizations, etc.), which greatly reduces maintenance costs. Also note, that no product-specific optimizations (like those required for jump procedures) take place: all forward rate paths are pre-computed and stored, making the implementation flexible, i.e., suitable for a vast array of product types.

Some Libor model implementations we are aware of benefit from this possible separation to some extent: they compute several payoffs in a given portfolio simultaneously, i.e., using the same paths. This paper extends the approach to handle new, unforeseen types of payoff functions or products that might emerge during a trading day and need to be priced with a fast response time. This is achieved by compiling new payoff functions on-the-fly into an executable, and accessing the pre-computed forward rate paths via a shared memory architecture.

The approach thus consists of the following steps:

0. Calibrate Libor model (minutes)
1. Evolve all forward rate paths and store them in shared memory (hours/single processor; minutes/multiple processors)

2. For each new product: aggregate payoffs (seconds)
   (a) compile the product's payoff function on-the-fly;
   (b) average numeraire-adjusted payoffs.

The following section describes the implementation of this distribution of computational workload. Several programming idioms are proposed to provide flexibility for new types of payoff functions and products, and to handle the large amount of memory required to store and access the pre-computed forward rate paths.

2.1 Evolve and store rates

The main idea of the proposed approach is to pre-compute and store all forward rate paths required for the Monte Carlo simulation. If this has to happen only once per day, a single-processor setup is sufficient. Yet in our approach, the forward rates must be re-computed not only when a new calibration takes place, but also when a new snapshot of initial forward rates—i.e., $L_i(t = 0)$—has to be used. This is likely to be necessary several times during a trading day, but a modest parallelization would achieve an acceptable performance for this step: a (parallelized) server application can (1) pre-compute the forward rate paths starting with the current forward rate snapshot; after completion, it will (2) designate this set of newly computed forward rates to be the one used for payoff aggregation; finally, it will (3) pre-compute the next set of forward rate paths based on the updated, most current forward rate snapshot.

Calculating the forward rates thus seems feasible. But storing them is less trivial. Our approach proposes to store them in main memory, to obtain optimal pricing speed. Thus, main memory size becomes the limiting factor. For n Monte Carlo runs, and M forward rates, $n*(M^2+M)/2$ values are required to store all forward rate paths (note: as rates expire, not every rate has to be tracked over the whole simulation period). An additional n*M values are required to hold all numeraire paths.

In a typical setup (n=107 Monte Carlo runs, and 20 years to simulate, with M =40 semi-annual forward rates), $8.6*10^9$ values need to be stored. Each value, if stored at double precision, requires 8 bytes, yielding a total of 68.8 gigabytes. In the scenario of intra-day updates, twice this memory is required to store both the forward rate paths being used for pricing at a given time, and those being precomputed from an updated forward rate snapshot (and bound to be used for pricing at completion). Unfeasible a few years ago, such an amount of memory can nowadays be handled well by 64-bit computer architectures.

2.2 Aggregate payoffs

When all forward rate paths are known, pricing a product becomes averaging its (numeraire-adjusted) payoffs. One preliminary aspect is how to handle payoff functions.

There are various ways to handle different product types or payoff function types in a Monte Carlo implementation. A common one is a "semi-hardcode" approach: each product type corresponds to some data structure, and users can parameterize those structures, e.g., by setting a strike parameter.

Another approach is to let users enter a payoff function as a text string, which in turn is evaluated by a software interpreter (much like a Basic-type mini-language). This is more flexible (new product types do not require changes to the software implementation of the model), but inefficient.

A similarly flexible, but faster alternative is, again, to allow the user to enter the payoff function as a text string, but then to compile this string on-the-fly into an executable (Examples for software tools that use a similar on-the-fly compilation technique are cash flow engines that need to adapt flexibly to a variety of cash flow structures.). C++ is the language of choice for this, especially because its operator overloading capability makes the payoff function look very intuitive.

For example, a user could enter the following line to define a specific caplet payoff:

```
B(0)/B(8)*max(0,delta(7)*(L(7,7)-0.03))
```

A parser then checks the input for syntax errors, adds code for the Monte Carlo loop, and slightly transforms the user input by adding Monte Carlo path indices:

```
double price = 0.0;
for (int n=0; n<10000000; n++) {
price +=
B(n,0)/B(n,8)*max(0,delta(7)*(L(n,7,7)-0.03));
}
price /= 10000000.0;
```

The various elements in the transformed code above are:

- delta(t): distances between the tenor dates.
- L(n,j,t): C++ object with overloaded parenthesis operator that accesses the precomputed forward rate paths and returns the j-th forward rate at time index t in the n-th Monte Carlo run.
- B(n,t): Another memory access object that accesses the numeraire paths for time index t in the n-th Monte Carlo run.

The payoff executable—which averages numeraire adjusted payoffs—is then executed to obtain the price. Note: the code transformation, the compilation, and the execution take place in the background and are invisible to the user, hence it is a “runtime” solution.

The executable’s speed will mainly depend on the memory access patterns within the “L”-object, and on the processor’s cache hierarchy. But even with sub-optimal memory access patterns, and with millions of Monte Carlo paths, the payoff executable will yield a price within seconds.

2.2.1 Operator overloading

To access data structures like arrays in C, the []-operator is used, e.g., DATA [2][7]. In C++, objects can be wrapped around the basic data structure to provide safe and maintainable code, e.g., DATA.getElement (2,7). More elegantly, C++

allows redefinition of the ()-operator, yielding highly readable code like DATA (2,7).

There are several reasons to use parenthesis operator overloading:

- It makes function calls and memory access operations look consistent.
- The data structure that is accessed is not a conventional n*M*M array, as rates only have to be stored up to their fixing date. Access with the standard []-operator would therefore look inelegant, and is best hidden behind the parenthesis notation.
- If the layout of the data structure is modified (e.g., to optimize the number of cache misses), it only requires changes within the parenthesis operator, but no changes to the operator's interface.
- Exception handling inside the operator can be used to signal invalid memory access during development.
- There is no performance penalty, as the parenthesis operator can be implemented as an inline function.

2.2.2 Memory access

Finally, some remarks about memory access. It is preferable to keep the pre-computed forward rate paths in main memory all the time to avoid delays when loading the data from a hard drive. It is also not advisable to let each payoff executable have their own copy of the forward rates, which would waste memory and cause disk activity due to virtual memory architectures if several payoff executables run at the same time. It is better to store and access only one copy of all forward rate paths in main memory, and let the payoff executables access them via shared memory constructs.

The pricing speed will now mainly depend on the memory access and the number of cache misses it causes. It is advisable to store the forward rate paths in such a way that the arguments to most products' payoff functions are stored close to each other in memory. If, for example, most payoffs are expected to require the 5- and 10-year forward rates, the Monte Carlo instances of these rates should be stored in one continuous area of the main memory.

3 Drawbacks

The benefits of the proposed approach may not always outweigh the drawbacks:

- Only the pricing step is fast (this ensures fast response times for users); precomputing the forward rate evolutions is—if the model implementation is deliberately kept simple to reduce costs—unoptimized and slow. Thus, the approach is not helpful for the inverse problem of finding suitable model parameters.
- Some products' fixing times will not coincide with the model's tenor structure. In that case, interpolation techniques like the geometric Brownian bridge (outlined by Brigo and Mercurio [2]) can be used. Other interpolation methods are described by Fries [4]. The approach's acceptance will depend on a comparison between interpolating spot rates in the simulation versus the conventional approach of interpolating the input data to achieve product-specific model tenors.

- If the model has to be calibrated several times during the trading day, more computational resources are needed to pre-compute the forward rate evolutions, potentially requiring some parallelization. This can offset some of the maintenance cost reductions achieved with the smaller and simpler code base. In addition, the proposed 64-bit computer architecture will adversely affect the setup costs (we consider those costs to be dominated by maintenance costs, though).
- Access to shared memory is implemented differently on different operating systems, which can affect the code's portability. As the size of these code segments is very small, this negative impact is small.

4 Conclusion

This paper proposes to separate the computational workload in a typical Libor model setting in two parts: an initial stage that pre-computes and stores forward rate paths, and a pricing stage, that merely aggregates payoffs to obtain a product's price. It also suggests compilation of payoff functions on-the-fly in order to adapt flexibly to unforeseen payoff function types at run-time.

This approach is fast for an end-user: product prices are obtained in seconds. It is arbitrarily accurate: it is possible to use a large number of Monte Carlo runs, a high discretization resolution, and even full factor models without affecting the pricing speed. The implementation is cheap to maintain: the software design is simple and the code base is small. The implementation is also flexible: a vast variety of products can be priced without tailoring the implementation to certain product types.

The approach, however, does not help with the inverse problem of parameter calibration; it is best implemented using 64-bit computer architectures with large memory capacity; and the individual products' tenor structure can require some interpolation during the simulation.

There is ample space for improvements: choosing a cache efficient storage order for the forward rate paths can minimize cache misses and further speed up the pricing; accounting for "single instruction, multiple data" (SIMD) commands common in modern processors could improve performance; various interpolation techniques need to be analyzed to assess the pricing accuracy on dates that do not correspond to model tenor dates.

References

[1] Brace, A., Gatarek, D., and Musiela, M. The market model of interest rate dynamics. Mathematical Finance 7, 2 (1997), 127–147.

[2] Brigo, D., and Mercurio, F. Interest Rate Models—Theory and Practice: With Smile, Inflation and Credit. Springer, 2006.

[3] Fan, R., Gupta, A., and Ritchken, P. On pricing and hedging in the swaptions market: how many factors really? Journal of Derivatives 15, 1 (2007), 9–33.

[4] Fries, C. Mathematical Finance: Theory, Modeling, Implementation. Wiley, 2007.
[5] Gatarek, D., Bachert, P., and Maksymiuk, R. The Libor Market Model in Practice. Wiley, 2006.
[6] Glasserman, P. Monte Carlo Methods in Financial Engineering. Springer, 2003.
[7] Jamshidian, F. Libor and swap market models and measures. Finance and Stochastics 1 (1997), 293–329.
[8] Moritsch, H., and Pflug, G. High-performance numerical pricing methods. Concurrency and Computation: Practice and Experience (2002), 665–678.
[9] Rebonato, R. Modern Pricing of Interest-Rate Derivatives. Princeton University Press, 2002.
[10] Schoenmakers, J. Robust Libor Modelling and Pricing of Derivative Products. Chapman and Hall, 2005.