Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

WORK REDUCTION IN FINANCIAL SIMULATIONS

Jeremy Staum

Industrial Engineering & Management Sciences
Northwestern University
Evanston, IL 60208-3119, U.S.A.

Samuel Ehrlichman
Vadim Lesnevski

Operations Research & Industrial Engineering
Cornell University
Ithaca, NY 14853, U.S.A.

ABSTRACT

We investigate the possibility of efficiency gains from schemes that reduce the expected cost of a simulated path, which allows more paths given a fixed computational budget. Many such schemes impart bias, so we look at the bias-variance tradeoff in terms of mean squared error. The work reduction schemes we consider are fast numerical evaluation of functions, such as the exponential, as well as changes to simulation structure and sampling schemes. The latter include descriptive sampling, reducing the number of time steps, and dispensing with some factors in a multi-factor simulation. In simulations where computational budgets are tightly constrained, such as risk management and calibration of financial models, using cheaper, less accurate algorithms can reduce mean squared error.

1 INTRODUCTION

This paper explores several ideas that might enable practitioners to improve the efficiency of their simulations. These ideas are work reduction techniques because they reduce the time spent per simulated path, rather than reducing the variance per simulated path, as variance reduction techniques typically do. Efficiency improvement in simulation is generally thought of as variance reduction, that is, a reduction in the variance of the simulation estimate given a fixed budget C of computer time. In the prototypical case, the simulation estimate is $\bar{X} = \sum_{i=1}^{n} X_i/n$ where the X_i are iid copies of an unbiased estimator X computed on one sample path. Variance reduction often means a scheme to reduce the variance V of X. Such schemes frequently require more work W, i.e. computer time, per path than a standard simulation does. Given a fixed budget, they have fewer paths than a standard simulation. To be effective, such a variance reduction method must produce a reduction in V which outweighs the increase in W: the figure of merit is VW because the variance of $\bar{X}$ is $V/n = VW/C$. In this paper, we consider some work reduction schemes, which reduce the computational cost W per path, and investigate when they improve the efficiency of financial simulations.

Some work reduction schemes we investigate create bias, so VW is not the appropriate figure of merit. For biased simulations, it is standard to focus on the mean squared error (MSE), seeking to reduce it given the fixed computational budget. We focus on samples of moderate size, not asymptotics as do Fox and Glynn (1989).

Instead of fixing the budget, one may fix a target MSE and seek to reduce the work required to achieve it. This is often appropriate in financial applications, where achieving a very low MSE merely wastes time, Spock-like, on unhelpful precision. For example, if the derivative security to be priced trades with a bid-ask spread of 1% of its value, it is not much use to attain a root mean squared error (RMSE) of less than, say, 0.1% of its value. More importantly, model risk also puts limits on how much precision is profitable and suggests that one should not be too concerned about bias. If our models are not all that close to being correct, why worry about computing an expected discounted payoff or a quantile of portfolio value too precisely under this particular probability measure?

Nonetheless, it is worth worrying about the efficiency of financial simulations. At present, the computational burden of simulation hinders accurate large-scale risk measurement, making it more difficult to manage risks and price them in the context of a firm's overall risk profile. For a different approach to simulation design for risk measurement, balancing the number of paths simulating the state at the risk measurement time horizon against the number of paths used to simulate security prices at that time, see Lee (1998) and Lee and Glynn (1999). The computational burden also slows the recalibration of models, degrading the quality of pricing and hedging. The application of variance or work reduction methods can make such simulations faster and more practical.

The computationally demanding applications for which these work reduction ideas might be worth implementing, such as calibration and risk management, are generally

performed frequently, perhaps daily. The parameters do not change much from one repetition to the next. Therefore if we optimize the level of work reduction for yesterday's simulation, having seen its result, this level is very likely best today too.

The first work reduction method to apply would be not one of the ones we described here, but simply writing a computer program that refrains from unnecessary memory operations and tailors the algorithm to the machine's architecture. Memory access is rather expensive. We found that it tends to take longer than generation of a uniform random variate, costing as much as 10–20 additions. According to Goedecker and Hoisie (2001): "Achieving high performance on modern architectures is intimately related to a coding style that, by minimizing memory traffic, maximizes processor utilization." "Memory access problems are usually the single most detrimental factor leading to large performance degradation. The basic principles are rather simple and rewards are large." "Applying optimization techniques when writing a code, leading to an optimal mapping of algorithms to the computer architecture, can significantly speed up a program, often by a factor of 10 to 100."

Aside from such optimization, there are also gains in speed to be had from other ways of utilizing hardware more efficiently. Using single instead of double precision for floating-point numbers makes division and memory access 1.5–2 times as fast, but single precision may be inadequate for some applications. Simulations also tend to be amenable to parallel processing, although this is not trivial to implement. A simple, limited version of parallelization is available for some processors, such as Intel Pentium III and IV: single instruction multiple data (SIMD) processing, which allows the same operation to be performed on four single precision floating-point numbers at once. We found that SIMD usually increases speed somewhat more than threefold.

We focus not on these, but on work reduction schemes that involve changing the simulation algorithm itself, not just its implementation. First, in Section 2, we consider reducing the cost of approximations to functions such as the exponential. As Goedecker and Hoisie (2001) point out, implementations in standard scientific computing libraries are often very accurate, and there exist much faster, slightly less accurate algorithms, including some optimized for particular processors by their vendors. Given our requirement for only modest precision in most financial applications, we might do better to use these cheaper approximations. Second, there are schemes for avoiding the generation of random variables, including the descriptive sampling of Saliby (1990, 1997). Third, we investigate changes to the stochastic process or cashflows being simulated. These latter two topics are discussed in Section 3.

2 REDUCED COST APPROXIMATIONS

In this section, we investigate the effect of using different algorithms to approximate the exponential on the speed and MSE of simple simulations. We assume that the requisite standard normal random variates are stored. (If it is faster, they could be generated anew or regenerated using a fixed seed for the random number generator.) In these examples, the expectations are actually known, allowing us to compute MSEs. Although simulation is not necessary for these examples at all, the results are suggestive in that the best exponential algorithms here may be good for other applications where simulation is necessary.

Vendor algorithms optimized to their hardware are potentially the most powerful tools to use here. We discuss them separately in Section 2.5 because it is difficult to separate the increase in speed they produce into elements attributable to lower accuracy, tailoring to processor architecture, and vectorization—many of these functions process an entire array of arguments in one function call instead of being called repeatedly to process each element of the array in turn. Throughout this section, except when mentioned, results are reported from simulations run at single precision.

2.1 Candidate Algorithms

We compare five algorithms for approximating the exponential. Two are highly accurate, while the others are actually approximations in the sense that their accuracy is substantially less than the computer's precision.

One is the standard algorithm to which the compiler automatically links. As the link is to the precompiled library `libm`, we do not report what these standard algorithms do. Another is `__ieee754_exp`, which is available at `http://www.netlib.org/fdlibm/e_exp.c`. It uses argument reduction to change the problem of computing $\exp(x)$ to computing $2^k \exp(r)$ where $x = k \ln 2 + r$ and $|r| \leq (\ln 2)/2 \approx 0.34657$. Its core is a rational approximation to $\exp(r)$ involving a polynomial of fifth degree in r^2. Argument reduction is somewhat costly because the time it takes to compute k and r from x is nonnegligible compared to that for computing the rational approximation itself. However, it transforms a good approximation on a small domain into an approximation that is good everywhere.

One approximation to $\exp(x)$ is the fourth-order Taylor expansion $1 + x + x^2/2 + x^3/6 + x^4/24 = (((0.041\bar{6}x + 0.1\bar{6})x + 0.5)x + 1)x + 1$. The latter representation is faster to compute. Argument reduction can enhance a Taylor expansion, which only approximates well near the expansion point, which is zero here. The algorithm `bargain` is the fourth-order Taylor expansion on $[-0.5, 0.5]$. Elsewhere on $[-1.5, 1.5]$, it returns either $1/e$ or e times the result of the Taylor expansion for the reduced argument in $[-0.5, 0.5]$. Outside $[-1.5, 1.5]$, it calls the `libm` exponential. This

means that the execution time of the function is random when the argument is random: it is faster when the arguments tend to be near zero.

Finally, we also consider the approximation of Schraudolph (1999). This macro uses one multiplication and one addition, taking advantage of the structure of floating-point representation, to achieve the effect of a lookup table with linear interpolation.

2.2 Examples

Our first financial example is pricing a European call option under the Black-Scholes model. We simulate in a single step under the risk-neutral measure $\mathbf{Q}$. The estimator is

$$(a\exp(bZ)-c)^+$$

where $a = S_0\exp(-(\sigma^2/2)T)$, $b = \sigma\sqrt{T}$, $c = \exp(-rT)K$, and Z is a standard normal random variate. The parameters in our base case are $S_0 = 100$, $K = 100$, $\sigma = 20\%$, and $r = 5\%$. Computing $a\exp(bZ)$ instead of $\exp(\ln a + bZ)$ helps because bZ tends to be near zero, which means that less argument reduction needs to be done in methods that use it, and the Taylor approximations are more accurate.

An alternative estimator is

$$\mathbf{1}\{Z > d\}(a\exp(bZ)-c)$$

where $d = (\ln(c/a))/b$, which saves computation of an exponential when the payoff is zero at the cost of adding an `if` statement. It is expected to be faster except when the option is extremely deep in the money. In the tables of Section 2.4, these two variants of the example are referred to as call and call-c (for "conditional exponentiation") respectively.

The second financial example is pricing a cap under a two-factor Gaussian HJM model. We simulate multiple steps under the spot LIBOR measure $\tilde{\mathbf{Q}}$, for which see for instance Musiela and Rutkowski (1997, §§13.3 and 14.3.3). The estimator is

$$\sum_{i=1}^{m-1} D_{i+1}\delta(L_i-\kappa)^+$$

where D_{i+1} is the discount factor for a cashflow arriving at T_{i+1} and L_i is the spot LIBOR rate at T_i for the period $[T_i, T_{i+1}]$. We simulate the negative log bond prices $X_{ik} = -\ln B(T_i, T_k)$ at time T_i for maturity T_k by

$$X_{ik} = X_{i-1,k} - X_{i-1,i} + a_i + b_i\cdot Z_i,$$

where $\{Z_i\}_{i=1,\ldots,m}$ are iid multivariate standard normal random variables, $a_i = \frac{1}{2}\delta\|\nu_{ik}\|^2$, and $b_i = \sqrt{\delta}\nu_{ik}$. Then

$$L_i = \frac{1}{\delta}(\exp(X_{i,i+1})-1)$$

and

$$D_i = \exp(Y_i) \quad \text{where} \quad Y_i = \sum_{j=1}^{i} X_{j-1,j}.$$

Conditional exponentiation is applicable here, as for the European call option. The estimator can be rewritten as

$$\sum_{i=1}^{m} \mathbf{1}\{X_{i-1,i} > c_1\}\exp(Y_i)(\exp(X_{i-1,i})+c_2)$$

where $c_1 = \ln(1+\delta\kappa)$ and $c_2 = -(1+\delta\kappa)$. When the indicator is zero at step i, we need not evaluate the other factors in that term. We just go to the next step, updating $X_{i,i+1},\ldots,X_{i,m}$ and Y_{i+1}, but we do not need to evaluate the two exponentials at this step. Thus the work done in exponention per path is significantly random, and its expectation varies directly with how deep in the money the cap is. Table 3 illustrates the effect this has.

We parametrized our example with $T = 5$, $\delta = 0.25$, and made the initial yield curve flat at 5%. In the base case, the strike was $\kappa = 5\%$. All forward rate instantaneous volatilities had magnitude $\|\sigma_{ik}\| = 1\%$. The greater $k-i$, i.e. the further into the future the time period of the forward rate, the greater σ_{ik}'s coefficient on the second factor, leading to correlations of 0.5–1 for forward rates. These forward rate volatilities produce the bond instantaneous volatilities via the equation $\nu_{ik} = \delta\sum_{j=i}^{k-1}\sigma_{jk}$.

2.3 Results: Bias

We will only use fast approximations to the exponential if they lead to acceptably low bias in simulation applications. Our guideline is that relative bias much greater than 0.1% is not acceptable. We estimated the relative bias (one minus the ratio of expected simulated price to true price) for each approximation in various examples. This was done by substituting a simulation estimate with a low standard error (0.01% for call examples and 0.03% for cap examples) for the unknown expected simulated price in the expression for relative bias. The true price is known in these examples.

The `__ieee754_exp` function states that it is accurate to within one unit in the last place at double precision, while the `libm` algorithms are accurate to double precision. The `bargain` algorithm is not as accurate, but the 95% confidence interval for its relative bias included zero for every example that we checked. The fourth-order Taylor approximation also had zero in the confidence interval for

the base call and cap examples, but generated significant bias in other examples, listed in Table 1.

Schraudolph's approximation algorithm is not accurate enough for use in any of the examples we considered. It had a relative bias of 4.3% for the call example and 300% for the cap example. This approximation was designed for use in neural networks, where the logistic function $(1+\exp(x))^{-1}$ is of more interest than $\exp(x)$ itself. By contrast, in finance, it is functions of the form $\exp(x)-1$ that are of most interest, and these demand a greater degree of precision in approximating $\exp(x)$ to attain an acceptable relative error.

We further examine the quality of the fourth-order Taylor and Schraudolph approximations by investigating how they vary with the size of the argument. To illustrate this dependence, we show in Table 1 the bias of an out-of-the-money European call pricing simulation with varying maturity T. The longer the maturity, the greater the variability of the argument to the exponential. The moneyness of the option is held constant: in each case $K = S_0 \exp((r-\sigma^2/2)T + \sigma\sqrt{T})$, so that the risk-neutral probability of a nonzero payoff is $\Phi^{-1}(-1) \approx 16\%$. Looking at the relevant truncated normal distribution, we can see that the simulation's bias will reflect primarily the accuracy of the approximation to the exponential for arguments whose size is $\ln(K/S_0)$ or a small multiple of it.

For the examples in Table 1, it was necessary to use a better random number generator than the `rand` function of `libm`, which introduced bias of up to 0.8%. We used instead `MRG32k3a`, which is described in L'Ecuyer (1999).

Table 1: Effect of Argument Size on Relative Biases in Call Pricing

	Algorithm	
T	4th order	Schraudolph
0.25	$\approx 0\%$	-3.7%
1	-0.04%	14%
2	-0.1%	7.5%
5	-0.8%	-1.6%
10	-2.6%	-1.2%

Table 1 shows that the relative bias induced in call prices is unacceptably large for the Schraudolph approximation, and is acceptable for the fourth order Taylor approximation only when the arguments are small enough. The superiority on small arguments of a Taylor approximation about zero motivated the combination with elementary argument reduction ideas to produce the `bargain` algorithm. The Schraudolph algorithm performs best, although not well enough, when arguments tend to be large because our concern with relative bias and a payoff function similar in form to $\exp(x)-1$ makes the demand for precision greater when x tends to be smaller. This also explains its disastrous performance in the cap example, where the arguments tend to be the smallest.

2.4 Results: Speed

The relative speeds of different algorithms depend greatly on whether exponentiation is done in isolation or as part of a financial simulation in which other operations occur. This is because a modern processor involves several subunits that perform operations simultaneously during execution of a nontrivial program, which is known as "on-chip parallelism." For this reason, it is not straightforward to deduce from the speed of an approximation to the exponential when run alone what speed-up it will produce when used in a particular financial simulation. The examples we give thus can only suggest what might happen in other applications. As we will see, it is typical for the speed gains in a complicated example to be less than those in isolation.

In Table 2, we report the number of seconds required to do ten million simulations, for each of the algorithms and examples under consideration. The compilers and processors used are:

- Microsoft Visual C++ 7.0, Intel Pentium III 1GHz
- Microsoft Visual C++ 7.0, Intel Pentium IV 1.8GHz
- `gcc` 2.95, Sun UltraSparc IIi 440MHz.

To establish the reliability of these results, we replicated them on different Pentium III and IV processors. The results were similar, up to adjustment for varying clock speeds.

We present a range of times for running the `bargain`, `__ieee754_exp`, and Sun `libm` algorithms in isolation because argument reduction causes the speed of an algorithm to depend on its input. In the other examples, the arguments are random, as specified by the financial simulation. In that case, despite argument reduction, it makes sense to present a total time, corresponding to an average time per path.

From Table 2 we see that, depending on the processor and example, the `bargain` algorithm compared to `libm` produces increases in speed of about 1.3–1.7 (Sun), 1.6–2.2 (Pentium III), or 2.9–3.8 (Pentium IV) in these basic financial simulations. In isolation, its relative speed is even greater, but such outperformance is not to be expected in the context of a simulation containing an overhead of other computations not affected by the choice of algorithm for the exponential.

The improvement over `libm` is greater on the faster processors. (Even the highly accurate `__ieee754_exp` outperforms `libm` significantly on the faster processors.) Apparently, `libm` is not much faster on a Pentium IV 1.8GHz than on a Pentium III 1.0GHz. The call example is even slower. The cap example is faster, but this relates primarily to faster simulation of log bond prices, which is a large overhead in this example. Likewise, `__ieee754_exp` in

Table 2: Effect of Approximations to the Exponential on Speed of Financial Simulations (Seconds / 10^7)

Sun				
Method/Example	alone	call	call-c	cap
4th-order Taylor	0.9	1.4	1.05	76
`bargain`	1.1–1.7	2.15	1.65	94
`__ieee754_exp`	2.2–3.5	3.55	2.5	122
`libm`	2.5–3.2	3.6	2.5	125
Pentium III				
Method/Example	alone	call	call-c	cap
4th-order Taylor	0.28	0.85	0.65	41
`bargain`	0.35–0.75	0.95	0.7	43
`__ieee754_exp`	1.2–2.3	1.65	1.15	57
`libm`	1.7	2.1	1.35	69
Pentium IV				
Method/Example	alone	call	call-c	cap
4th-order Taylor	0.13	0.56	0.44	15
`bargain`	0.14–0.26	0.58	0.45	15.5
`__ieee754_exp`	1.4–2.1	1.3	0.85	31
`libm`	1.6	2.2	1.3	54

isolation is no better with the Pentium IV. It has often been remarked that increased clock speed is no guarantee of superior performance for real applications. The cheaper exponential algorithms seem better able to take advantage of the opportunities for on-chip parallelism, breaking down into smaller sub-tasks which finish more quickly on the faster processor, while the more accurate standard algorithms do not fare so well. Improvements in processors, far from diminishing the importance of fast approximations, are increasing it.

We also see that argument reduction makes argument size affect the speed of the algorithm by a factor of as much as two. Moreover, the `if` statements required to implement it have a nonnegligible cost even when the argument is already small and they are not needed. This is why `bargain` always costs more than the fourth-order Taylor approximation, which is the same thing without argument reduction. However, the cost of argument reduction in financial simulations using `bargain` diminishes for faster processors.

Another interesting point is that conditional evaluation of exponentials only when in the money speeds up the simulation by a factor of 1.3–1.4 for `bargain` or 1.5–1.7 for `libm`. The example given here is an at-the-money option; the effect would be greater for out-of-the-money options.

One anomaly in Table 2 is that on the Pentium IV, the call simulation with `__ieee754_exp` is faster than running that function alone, even on small arguments. It is hard to know what to make of this, but it may be simply a surprising effect of on-chip parallelism.

Table 3 shows the effect of the cap's strike on speed gains from faster exponentials. As discussed in Section 2.2, a simulation of an out-of-the-money cap will have fewer nonzero cashflows, and thus less need for evaluations of the exponential. A strike of 10% is very seldom triggered, so the choice of approximate exponential algorithm is almost irrelevant. With a strike of 2%, the cap is deep in the money, and the exponentials are usually evaluated.

Table 3: Effect of Moneyness on Speed of Cap Simulation (Seconds / 10^7)

Pentium III, 5 years				
Strike	10%	7%	5%	2%
`bargain`	33	35	43	48
`libm`	33	40	69	96
Pentium IV, 5 years				
Strike	10%	7%	5%	2%
`bargain`	11	12	15.5	18
`libm`	11	18	54	87
Pentium IV, 1 year				
Strike	10%	5.2%	5%	2%
`bargain`	1	1.6	2.5	2.8
`libm`	1	6.2	16.1	28.1

It turns out that the increase in speed on the exponentials alone, excluding the overhead for doing the deep-out-of-the-money simulation with no exponentials, does not match the ratios for small arguments in Table 2: for instance, $(96-33)/(48-33) \neq 1.7/0.35$. One possible explanation for this is on-chip parallelism. We also see that for another realistic example, an at-the-money 1-year cap, `bargain` performs even better against `libm`, increasing speed more than 6.4-fold. In this example, there is relatively much less overhead, because fewer log bond prices need to be simulated.

2.5 Vendor-Specific Products

There are a variety of highly efficient functions optimized to specific processors and made available by the hardware manufacturer. We maintain our focus on the exponential function and report briefly on the Intel Performance Primitives (IPP) package. We also report on SIMD, which was mentioned in the introduction. It is not a separate package, but rather a set of instructions intrinsic to the processor, which some compilers know how to access. There is also an Intel Math Kernel Library, which like IPP has vectorized functions for single or double precision, and in versions whose accuracy is high or low (but not as low as in IPP). For the Sun platform, the Sun C compiler includes optimized exponential functions in scalar and vector form.

Vectorization is itself a form of optimization to the hardware. Evaluating a function on many arguments in one call, taking advantage of the processor's memory cache, can

be more efficient than calling the function many times in a row. However, there is a drawback to using vectorized functions, or even SIMD, which processes four arguments at once. Vectorization is not straightforward to implement unless all paths have same structure. For instance, conditional evaluation of the exponential in the European call example seems difficult to implement with vectorization. There would be an array of one thousand standard normal random variates Z on which the exponential is to be evaluated when $Z > d$. Those numbers satisfying $Z > d$ are only a subset, and not stored contiguously in memory. For similar reasons, even argument reduction is nontrivial to implement in vectorized form.

When using vectorized functions, the size of the memory cache limits the number of arguments that can be processed in one call. On these Pentiums, one thousand seemed to be optimal, so a loop of ten thousand calls to an IPP vectorized function performed ten million exponentiations.

IPP includes two approximations to the exponential, guaranteeing 11 and 24 bits of accuracy respectively. On the Pentium III, these appeared to be the same function—there was no cheaper 11-bit version. On the Pentium IV, they took respectively 0.045 and 0.075 seconds per ten million. The exponential with 24 bits of accuracy generated no statistically significant bias in the call pricing examples listed in Table 2.3. The 11-bit version produced biases of 0.02% for $T = 1$ and 0.04% for $T = 0.25$.

Table 4 presents the speed in isolation and on the call example of the IPP 11-bit exponential and the fourth-order Taylor approximation implemented with SIMD, compared to `libm` without the use of any special features.

Table 4: IPP and SIMD Speed (Seconds / 10^7)

	precision feature algorithm	double none `libm`	single IPP IPP(11)	single SIMD 4th-order
Pent. III	alone	1.7	0.24	0.12
	call	2.35	0.34	0.35
Pent. IV	alone	1.7	0.045	0.045
	call	2.2	0.23	0.088

Table 4 is comparable to Table 2, which reported the performance, without hardware-specific enhancement, of approximate exponential algorithms alone and on the call example. We see that the IPP vectorized exponential with 11-bit accuracy is faster than any algorithm in Table 2. For the call example, it yields speeds about 2.5 times greater than those reported in Table 2. The performance of the fourth-order Taylor approximation implemented in SIMD was similar, except that it was highly favorable in the call example on the Pentium IV. The use of SIMD led to a 25-fold improvement over the standard use of `libm` at double precision. This particular entry in the table is somewhat surprising, because the speed is over six times greater than that of the same example without SIMD, which works by processing four numbers at once, ordinarily speeding up a simulation by a factor of less than four.

Vendor-supplied packages of optimized functions contain many potentially useful things, but one of the most important is approximate division. We found regular division to be about as costly as memory access, about ten additions or multiplications. SIMD contains a faster approximate division. When this is available, rational approximations such as found in `__ieee754_exp`, or most routines for approximating the standard normal inverse cdf Φ^{-1}, become more attractive.

3 AVOIDING WORK

The previous section dealt with choices of algorithm for evaluating functions approximately. We now turn to changes in the structure of the simulation itself, not in the functions it uses. We have already explored one such change: in the difference between the call and call-c examples, Table 2 shows that evaluating exponentials only when the option finishes in the money can produce a noticeable savings of time. Another such work reduction idea involves stopping some simulated paths early (Glasserman and Staum 2002). Neither of these produces bias.

Here we focus on the mortgage-backed security (MBS), described in Section 3.1, as an example of a complicated simulation in which several corners can be cut in the hope of saving time while incurring an acceptably small bias. The possibilities including dropping factors in a multi-factor simulation, reducing the number of time steps below the number of dates specified in the financial contract, and replacing some random variates with things that are cheaper to compute.

As described below, our MBS simulation requires three factors to simulate an interest rate, a stochastic discount factor, and a mortgage rate. However, either or both of the latter two processes may be approximated as functions of the first, in which case the corresponding factor does not need to be simulated.

In reality, a mortgage has 360 monthly payments, so it is natural to price a MBS with a simulation of 360 time steps. We also simulated a mortgage of 240, 180, 120, or 30 equally spaced payments instead. We considered altering the mortgage rate as well, so as to keep the annualized percentage rate the same no matter the number of payments. It might be thought that this would ameliorate the bias that changing the number of payments introduces, but we found that it made it worse. There is another aspect to this bias, which is that longer payment periods lead to a slower amortization of the mortgage. These two effects are partially offsetting in this example. Indeed, it turns out that offsetting bias effects were crucial to any improvements identified in this section.

Setting the number of time steps to be less than the number of cashflows in order to reduce work is different from the problem addressed by Duffie and Glynn (1995). They considered setting the number of time steps to be greater than the number of cashflows in order to reduce bias from discretization of the stochastic differential equation (SDE) describing the underlying process. In this example, the SDE is integrated exactly.

The MBS example is high-dimensional, so we can apply Latin hypercube sampling (LHS) with 100 strata as a variance reduction technique. If random variates are generated during the simulation, not stored, a lower-cost alternative to LHS is the descriptive sampling (DS) of Saliby (1990). DS differs from LHS in always using the midpoint of a cell rather than sampling uniformly within a cell. It was intended as a variance reduction technique (Saliby 1997, Saliby and Pacheco 2002), but the variance reduction is only substantial when the number of strata is low, in which case the bias of DS tends to be very high. Here we consider it as a work reduction technique—we save the cost of generating a uniform random variate by simply using the cell midpoint. The results of Section 3.2 show that this savings tends to be quite small. However, the decision about whether to implement a simulation efficiency improvement depends on the relationship between the effort of implementation and the magnitude of the gains. Here the gains are small, but it is easier to implement DS than LHS.

A similar modification would be to replace, for instance, normal random variates with random variates that are cheaper to generate. Kloeden and Platen (1992, p. 458) discuss this possibility in terms of asymptotic convergence rates as the number of time steps goes to infinity, giving a binomial example. Such methods may improve MSE, for instance, when the overall budget is small, the number of time steps is large, and there is not too much path-sensitivity, but we do not investigate them here. Of course, in simulation applications where random variates are stored, none of these substitutions are helpful.

3.1 Mortgage-Backed Securities

For background on mortgage-backed securities (MBS), see for instance Richard and Roll (1989). We consider a security whose payments are the 360 monthly cashflows from a pool of 30-year mortgages. Simulations involving MBS are often demanding in part because of the large number of time steps, and possibly complexity in interest rate modeling. The characteristic feature of MBS is the homeowner's option to prepay the mortgage. In this example, we model the number of mortgages prepaying as a function (involving an arctangent and fitting the data of Richard and Roll 1989) of the newly available mortgage refinancing rate, although other factors may be modeled in practice.

We assume a very simple interest rate model: the spot interest rate r and the mortgage refinancing rate R both follow Ornstein-Uhlenbeck processes, i.e. the Vasiček model. To avoid bias, it is necessary to simulate a third factor, the log increment $Y_{i+1} = \ln D_{i+1} - \ln D_i$ of the stochastic discount factor $D_i = \int_0^{t_i} r(s)\,ds$, which is jointly normal with $r_{i+1} = r(t_{i+1})$. We assume conditional independence between R_i and D_i given r_i and all information from previous time steps, but the instantaneous correlation between r and R is 0.9. The initial rates are $r_0 = 6.5\%$ and $R_0 = 8.5\%$, but r and R have the same parameters otherwise: mean reversion level 10%, mean reversion strength 0.15, and instantaneous volatility 1%.

When we want to avoid simulating D with a separate factor, we use the approximation

$$D_j = \exp\left(-\delta \sum_{i=0}^{j-1} r_j\right).$$

When we want to avoid simulating R as a separate factor, we replace R_{i+1} with $\mathbf{E}[R_{i+1}|r_{i+1}, R_i]$.

3.2 Results

There is no closed-form expression for the value of a MBS, so we use a very precise Monte Carlo estimate as the true mean for purposes of calculating MSE. For each method, we report the RMSE given a fixed computational budget. The results of this section are from the `gcc` 3.2 compiler on an Intel Pentium III 450MHz, using double precision.

First suppose that the MBS pricing simulation uses simple random sampling (SRS). One work reduction technique is decreasing the number of time steps in the simulation from 360 to 240, 180, 120, or 30. The other is reducing the number of factors from three by approximating either D or both D and R in terms of r. We considered all combinations of these techniques. Table 5 shows, for different values of the computational budget, which combination produced the lowest RMSE, and how much less that was than the base case of SRS with 360 time steps and three factors. The MBS value was about 1.07.

Table 5: RMSE Improvement over SRS for MBS Pricing

budget	base	Best Method		
(seconds)	RMSE	steps	factors	gain
10	0.0023	240	1	33%
50	0.0010	240	2	7%
100	0.0007	360	3	0%

Next suppose that the MBS pricing simulation uses Latin hypercube sampling (LHS). This produces a 35% reduction in RMSE (i.e. standard error, because there is

no bias) in the base case of three factors and 360 time steps. We now consider all of the above possibilities for approximation, as well as the use of descriptive sampling (DS) instead of LHS. Whether the budget was 10, 50, or 100 seconds, we found that the best method was to use one factor, 360 time steps, and DS, leading to a reduction in RMSE of 42–45%. This is not so much because DS reduces the time per path (the reduction is only 4%), but because of a surprising cancellation of the bias introduced by using one factor. On its own, DS introduces a bias of -0.0002 into a simulation of three factors and 360 time steps. When used in a simulation with one factor and 360 time steps, which has a bias of 0.0019, it reduces the bias to 0.00005.

The utility of results such as these depends on the stability of biases from day to day as simulation parameters change. If a bias cancellation such as this one between DS and the one-factor approximation is relatively robust to changes in parameters, it can be used for a while with some confidence. Perhaps tests to find the currently optimal work reduction scheme could be run on the weekend, then used for one week.

We checked whether such robustness is present in this example by running the simulations with initial values $r_0 = 7.5\%$ and $R_0 = 9.5\%$, a fairly large increase of 1% in the interest rates over the base case. For this example too, some of the possible approximations provide a reduction in MSE of almost half. However, the approximation of one factor, 360 time steps, and DS increased MSE by 175%. The bias of this simulation is 0.00264: no favorable cancellation of bias takes place for these changed parameters. This shows that the approximations considered here are not suitable for this problem, at least unless the the parameters change very little inside an optimization or over the relevant timespan.

4 CONCLUSIONS

The work reduction techniques studied here have the potential to increase significantly the speed of computation-intensive financial simulations in reach an MSE target. Their success (or lack thereof) will vary greatly from problem to problem. These work reduction techniques may easily be combined with variance reduction techniques, but this too changes their effectiveness. In particular, they are compatible with quasi-Monte Carlo.

Cheap function evaluation with fast but moderately accurate approximations is a technique that can be applied fairly safely to most financial problems. A change to the exponential algorithm alone can increase the simulation speed by factors ranging from three to over six, for realistic examples on a Pentium IV processor. Using techniques specific to the processor can lead to a 25-fold increase in speed. However, efficient coding practices, such as avoiding unnecessary computation and memory access, are just as important, if not more so.

We also saw that changes to the structure of the simulation could also almost double its speed. Here the approximations that can be made depend on the particular problem, as does the combination that works best. Each problem has to be studied separately, but this may well be worthwhile in the case of expensive problems that are repeated frequently. A serious issue surrounding the use of these techniques is the question of the stability of bias with respect to changes in parameters; unless such stability exists, these techniques are not safe to use.

Academics may be interested in applying these techniques when they study variance reduction methods. Frequently a variance reduction method decreases variance per path while increasing cost per path. Cost per path depends on the accuracy of numerical function evaluation and coding style, which makes it hard to assess. The ideas presented here can help to design an efficient baseline simulation, and thus give a realistic picture of the efficiency gains from variance reduction. Most of all, practitioners may find it beneficial to check whether any of the techniques described here help to reduce the RMSE in their toughest simulation applications, given a fixed budget, or allow them to reach an RMSE target in less time.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. DMS-0202958. We thank Paul Glasserman, Peter Glynn, and Keith Lewis for discussions and references, but the views expressed are those of the authors, who are solely responsible for any errors.

REFERENCES

Duffie, D., and P. Glynn. 1995. Efficient Monte Carlo Simulation of Security Prices. *Annals of Applied Probability* 5(4): 897–905.

Fox, B. L., and P. W. Glynn. 1989. Replication Schemes for Limiting Expectations. *Probability in the Engineering and Informational Sciences* 3: 299–318.

Glasserman, P., and J. Staum. 2002. Resource Allocation among Simulation Time Steps. Forthcoming, *Operations Research*.

Goedecker, S., and A. Hoisie. 2001. *Performance Optimization of Numerically Intensive Codes*. Philadelphia: Society of Industrial and Applied Mathematics.

Kloeden, P. E., and E. Platen. 1992. *Numerical Solution of Stochastic Differential Equations*. New York: Springer-Verlag.

L'Ecuyer, P. 1999. Good Parameter Sets for Combined Multiple Recursive Random Number Generators. *Operations Research* 47(1): 159–164.

Lee, S.-H., 1998. *Monte Carlo Computation of Conditional Expectation Quantiles*. Doctoral dissertation, Stanford University.

Lee, S.-H., and P. W. Glynn. 1999. Computing the Distribution Function of a Conditional Expectation via Monte Carlo: Discrete Conditioning Spaces. In *Proceedings of the 1999 Winter Simulation Conference*, ed. P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, 1654–1663. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers. Available online via <http://www.informs-cs.org/wsc99papers/239.PDF>.

Musiela, M., and M. Rutkowski. 1997. *Martingale Methods in Financial Modelling*. New York: Springer-Verlag.

Richard, S. F., and R. Roll. 1989. Prepayments on Fixed-Rate Mortgage-Backed Securities. *Journal of Portfolio Management* 15: 73–82.

Saliby, E. 1990. Descriptive Sampling: A Better Approach to Monte Carlo Simulation. *The Journal of the Operational Research Society* 41(12): 1133–1142.

Saliby, E. 1997. Descriptive Sampling: An Improvement over Latin Hypercube Sampling. In *Proceedings of the 1997 Winter Simulation Conference*, ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson, 230–233. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers. Available online via <http://www.informs-cs.org/wsc97papers/0230.PDF>.

Saliby, E., and F. Pacheco. 2002. An Empirical Evaluation of Sampling Methods in Risk Analysis Simulation: Quasi-Monte Carlo, Descriptive Sampling, and Latin Hypercube Sampling. In *Proceedings of the 2002 Winter Simulation Conference*, ed. E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, 1606–1610. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers. Available online via <http://www.informs-cs.org/wsc02papers/220.PDF>.

Schraudolph, Nicol N. 1999. A Fast, Compact Approximation of the Exponential Function. *Neural Computation* 11: 853–862.

AUTHOR BIOGRAPHIES

JEREMY STAUM is Assistant Professor in the Department of Industrial Engineering and Management Sciences at Northwestern University. He received his Ph. D. in Management Science from Columbia University in 2001. His research interests include variance reduction techniques and financial engineering. His e-mail address is <staum@iems.northwestern.edu>, and his web page is <www.iems.northwestern.edu/~staum>.

SAMUEL EHRLICHMAN is a doctoral student at Cornell University, School of Operations Research and Industrial Engineering. He received his B.A. in Mathematics and Computer Science from Swarthmore College in 1995. Prior to studying at Cornell, he worked as a software developer in the financial services industry. His email address is <se52@cornell.edu>.

VADIM LESNEVSKI is a doctoral student at Cornell University, School of Operations Research and Industrial Engineering. His email address is <vadim@orie.cornell.edu>.