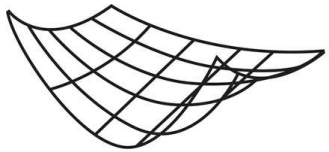
Generic and object-oriented programming techniques for Monte Carlo simulation in C++

Erik Schlögl



QUANTITATIVE FINANCE
RESEARCH CENTRE

University of Technology, Sydney

3 December, 2013

Outline

Some observations

- Financial instruments and the quant models used to price them are increasingly complex.
- There is a multitude of alternative models, and being able to switch easily between models is a useful advantage.
- There is a great deal of commonality between initially disparate areas of application, e.g.
  - FX modelling analogy for inflation or commodities
  - Interest rates & zero coupon bond prices versus credit spreads & risk–neutral probabilities of default
  - The two “perspectives” of Black/Scholes/Merton
- Increasing need for models integrating several sources of risk.

Background

As the complexity of software systems increased, “clean design” programming techniques were developed and refined, culminating in the now well–established discipline of *software engineering*.

- procedural programming
- generic and object–oriented programming
  - classes, objects and instances thereof
  - methods and message passing
  - abstraction: composition and inheritance
  - encapsulation
  - polymorphism
  - decoupling
  - templatisation

Design patterns

The observation that similar design problems recur in varying contexts led to the idea of common, reusable design patterns.

- Originally proposed as an architectural concept (Alexander et al. (1977) *A Pattern Language*)
- Established as a paradigm in software engineering (Gamma/Helm/Johnson/Vlissides (1995) *Design Patterns: Elements of Reusable Object–Oriented Software*)
- Playing an increasingly important role in quant finance applications (Joshi (2004) *C++ Design Patterns and Derivatives Pricing*)

Aim

Software that is

- reusable
- maintainable
- extensible
- satisfies the “implement once” principle

This is not always straightforward in quant finance applications, e.g.

- “model polymorphism” versus “payoff polymorphism”
- “generic programming” (templates) versus “object–oriented programming” (class hierarchies)

In this talk

- Pragmatic approach: use what works, rather than strictly adhering to one school or another
- Seek to create a collection of building blocks which can be varied independently
- Implement once (and in the right place) — doesn't mean code once, and doesn't preclude step-wise generalisation
- Important: figuring out the right way to decompose the problem
- Show that the design works when the building blocks are "mixed and matched"

Generic algorithm for Monte Carlo simulation

1. Initialise the variables RunningSum = 0
   RunningSumSquared = 0
   $i = 0$
2. Generate the value of the stochastic process X at each date T_k relevant to evaluate the payoff.
3. Based on the values generated in step 2, calculate the payoff.
4. Add the payoff to RunningSum and the square of the payoff to RunningSumSquared.
5. Increment the counter i.
6. If i is less than the maximum number of iterations, go to step 2.
7. Calculate the simulated mean by dividing RunningSum by the total number of iterations.
8. Calculate the variance of the simulations by dividing RunningSumSquared by the total number of iterations and subtracting the square of the mean.

A first pattern I

Monte Carlo gatherer or accumulator — collects MC results for each draw and can be queried for the resulting MC estimate & relevant statistics.

```
// instantiate object
MCGatherer my_gatherer;
...
// inside MC loop, say x contains the MC value for
// that particular draw
my_gatherer += x;
...
// after MC loop, query mean and standard deviation
result = my_gatherer.mean();
stddev = my_gatherer.stddev();
```

A first pattern II

Joshi (2004) presents an example of the MC gatherer in C++.

A more sophisticated approach in C++ would be to use the `Boost.Accumulators` template library (see www.boost.org).

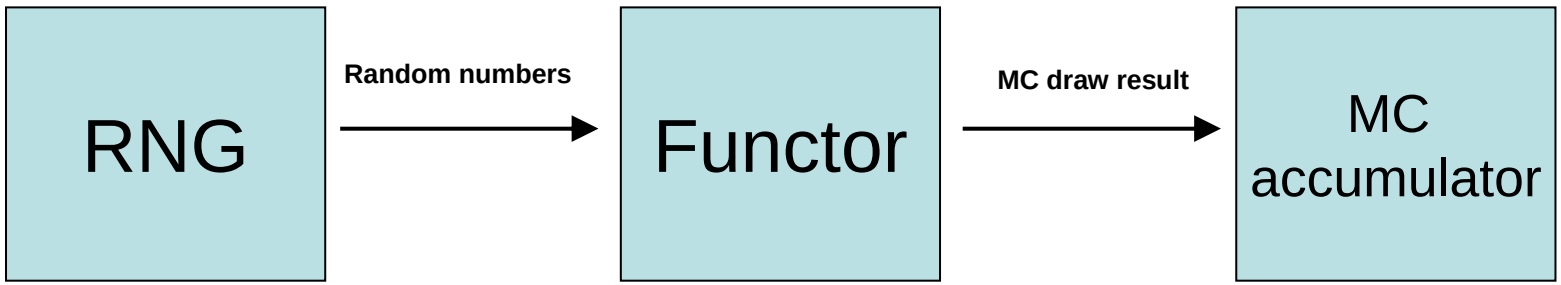
From the documentation:

> *The library deals primarily with the concept of an accumulator, which is a primitive computational entity that accepts data one sample at a time and maintains some internal state.*

Template–based implementation of the generic MC algorithm

Template parameters: argument type
return type
random number generator type

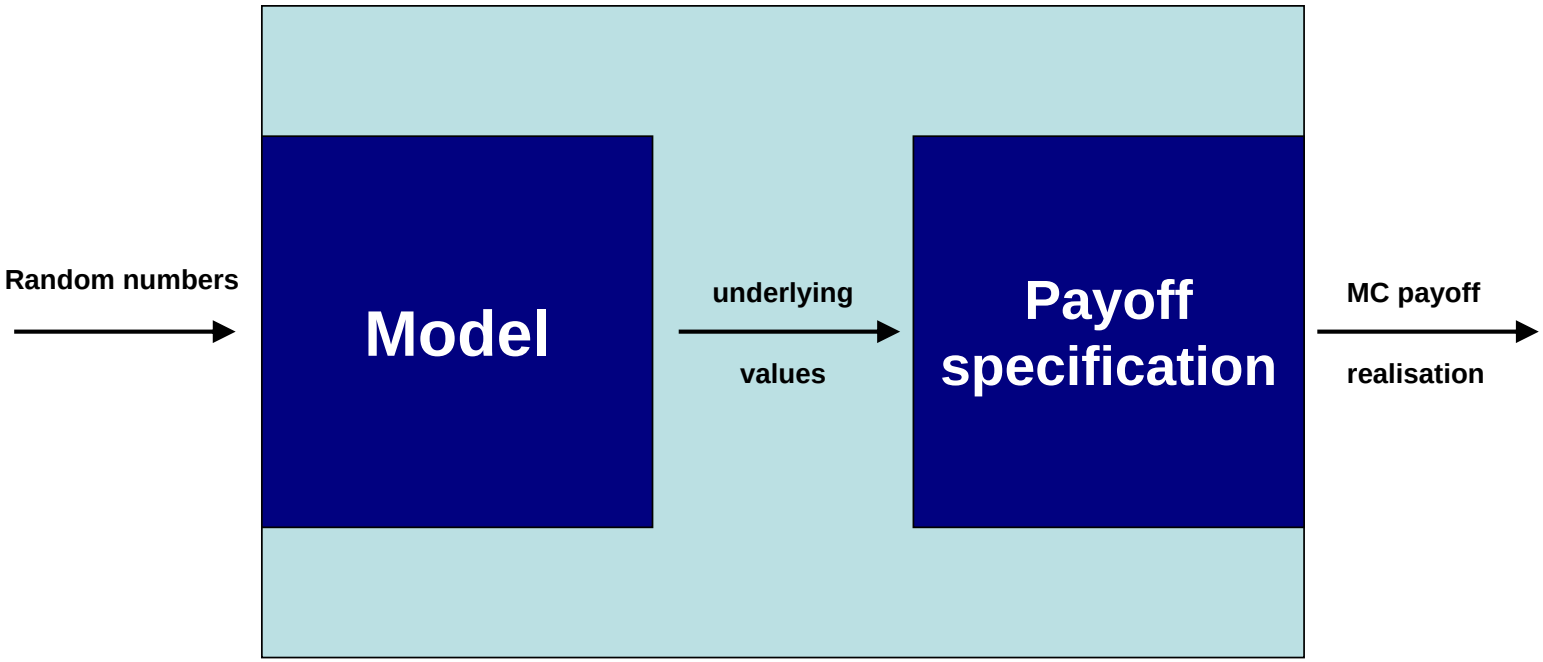
Data members: random number generator instance
functor (object representing a function)
mapping the *output of the random number generator* to a realisation of the MC draw



Role of the functor

Thus the functor encapsulates the mapping of the random numbers to the MC draw result.

In the typical quant finance context:



Generic MC algorithm in C++ — class template

```
/** Template for generic Monte Carlo simulation.

    The template parameter class random_number_generator_type
    must implement a member function random(), which returns a
    realisation of the random variable of the type given by
    the template parameter class argtype.
    */
template <class argtype,class rettype,
          class random_number_generator_type>
class MCGeneric {
private:
  NormalDistribution                                        N;
  /// Functor mapping random variable to Monte Carlo payoff.
  boost::function<rettype (argtype)>                        f;
  random_number_generator_type      random_number_generator;
public:
  inline MCGeneric(boost::function<rettype (argtype)> func,
                   random_number_generator_type& rng)
    : f(func),random_number_generator(rng) { };
  void simulate(MCGatherer<rettype>& mcgatherer,
                unsigned long number_of_simulations);
  double simulate(MCGatherer<rettype>& mcgatherer,
                  unsigned long initial_number_of_simulations,
                  double required_accuracy,
                  double confidence_level = 0.95);

};
```

Generic MC algorithm in C++ — simulation loop

```
template <class argtype,class rettype,
          class random_number_generator_type>
void MCGeneric<argtype,rettype,random_number_generator_type>::
simulate(MCGatherer<rettype>& mcgatherer,
         unsigned long number_of_simulations)
{
  unsigned long i;
  for (i=0;i<number_of_simulations;i++)
    mcgatherer += f(random_number_generator.random());
}
```

Recap: implementation of the generic algorithm I

- The initialisations in Step 1 were done by the constructor of the `MCGatherer` object.
- The function call `random_number_generator.random()` generates the random variable in Step 2,
- to which the functor `f` is applied to calculate the payoff in Step 3. Note that `f` encapsulates the entire mapping from random variables to discounted payoff, thus the generic implementation of the Monte Carlo algorithm abstracts from the *stochastic model* (which maps basic random variables to the dynamics of the underlying assets) and any particular *financial instrument* (which maps the prices of underlying assets to a discounted payoff).

Recap: implementation of the generic algorithm II

- Step 4 is embodied in the overloaded `operator+=()` of the `MCGatherer` object.
- Steps 5 and 6 are managed by the `for` loop.
- Steps 7 and 8 are completed on demand by the member functions `mean()` and `stddev()` of `MCGatherer`.

Generic implementation of the functor

- The payoff specification mapping of a set of asset values at a set of time points to a discounted payoff can be encapsulated in a class.

  This is advisable since the payoff mapping should not depend on how the asset values were generated.
- Typically, asset values are generated by a class representing a stochastic process.

  Note that except for the canonical case of simulating under the risk neutral measure under deterministic interest rates, the stochastic process must also supply the realisation of the numeraire.

Generic implementation of the functor — the mapping

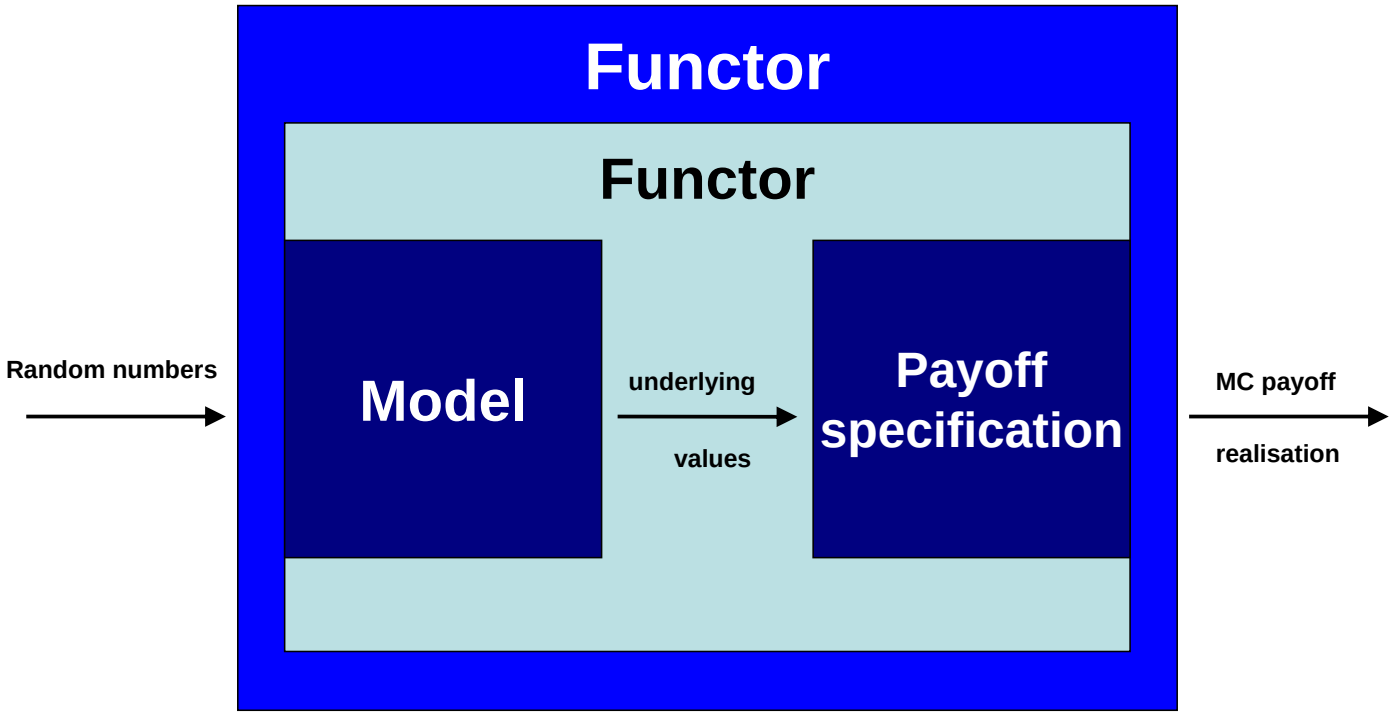
Given a stochastic process and a payoff specification, it remains to link the two together.

This procedure is again independent of the specifics of the stochastic process and the payoff, thus can be accomplished generically by a template.

MC mapping

```
class MCMapping {
private:  // ...
public:
  /// Constructor.
  MCMapping(MCPayoff& xpayoff,price_process& xprocess,const
TermStructure& xts,int xnumeraire_index = -1);
  /// Choose the numeraire asset for the equivalent martingale
  /// measure under which the simulation is carried out.
  bool set_numeraire_index(int xnumeraire_index);
  /// The function mapping a realisation of the (often
  /// multidimensional) random variable x to the discounted
  /// payoff.
  double mapping(random_variable x);
  /// The function mapping a realisation of the (often
  /// multidimensional) random variable x to multiple
  /// discounted payoffs.
  Array<double,1> mappingArray(random_variable x);

};
```

First extension: antithetic variates



Outer functor calls inner functor twice: once with the original variates, once with the antithetics, and averages the result.

A second extension: control variates

Control variates can be implemented at different levels:

- Naive product control variates (e.g. geometric average option vs. arithmetic average option) can be encapsulated in a class derived from `MCPayoff`.
- Product control variates with (typically unknown) optimal weights require estimation from the simulation via (multivariate) regression.

  $\rightarrow$ implement at the `MCGatherer` level
- Control variates at the model level (i.e. using a simpler model permitting closed form solutions as a control variate) would involve two stochastic processes

  $\rightarrow$ implement at the `MCMapping` level (not yet implemented).

Single control variate

f is the target functor, and g is closely correlated, but with known expectation.

Monte Carlo estimate

$$\hat{I}_N = \left(\frac{1}{N} \sum_{i=1}^{N} (f(u_i) - \beta g(u_i)) \right) + \beta E[g(U)]$$

An optimal choice of β minimises $\mathrm{Var}[f(U) - \beta g(U)]$, thus given by

$$\beta = \frac{\mathrm{Cov}[f(U), g(U)]}{\mathrm{Var}[g(U)]}$$

Single control variate — C++

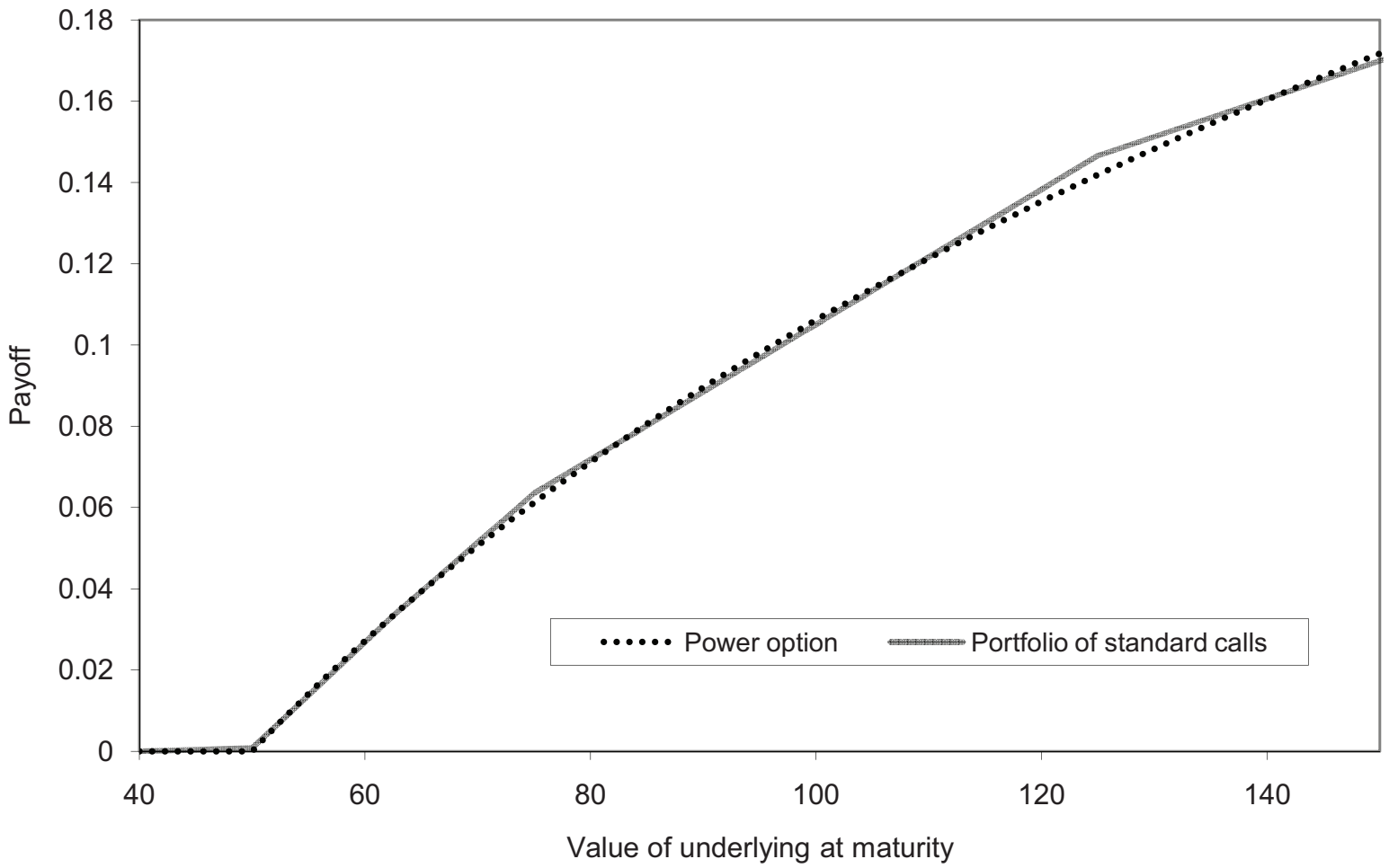
```
double MCGatherer<Array<double,1> >::CVestimate(int target,int
CV,double CV_expectation) const
{
  double b = CVweight(target,CV);
  return mean(target) - b*(mean(CV) - CV_expectation);
}

double MCGatherer<Array<double,1> >::CVweight(int target,int CV)
const
{
  double b = 0.0;
  if (CVon) {
  b = (covar(target,CV)/number_of_simulations() -
mean(target)*mean(CV)) / (variance(CV)*number_of_simulations()); }
  return b;
}
```

Multiple control variates — power option

Paths	MC value	CI width	CV MC value	CV CI width	Acc. gain
100	0.098103	1.63E-02	0.101249	5.75E-04	28.29
400	0.100593	8.19E-03	0.101322	2.86E-04	28.62
1600	0.101214	4.30E-03	0.101166	1.81E-04	23.79
6400	0.101297	2.17E-03	0.101146	8.48E-05	25.63
25600	0.100603	1.09E-03	0.101104	4.90E-05	22.20
102400	0.101176	5.44E-04	0.101110	2.39E-05	22.77
409600	0.101147	2.71E-04	0.101109	1.18E-05	23.08
1638400	0.101118	1.36E-04	0.101103	5.91E-06	23.01

Table 7.4 *Pricing an option with payoff (7.66), strike 1.47876, power* $\alpha = 0.1$ *maturing in 1.5 years. Closed form value of the option is 0.101104. The model is Black/Scholes with initial stock price 100, 42.43% volatility and an interest rate of 5%. Pricing is done by "plain" Monte Carlo and with five standard call options with strikes 37.5, 50, 62.5, 75 and 125 as control variates. "CI width" refers to the width of a 95% confidence interval around the Monte Carlo estimate.*

Multiple control variates — power option



Motivation

- Using binomial lattice models or finite difference schemes, one can price by backward induction, and thus the continuation value is known whenever the early decision needs to be evaluated.
- Simulating a Monte Carlo path, the continuation value is *not* known.
- The most common approach to approximate the continuation value is to estimate it by regression on a set of basis functions of state variables on a Monte Carlo path.
- This regression maps the values of the state variables at the time of potential early exercise to a (predicted) continuation value.

Lower bound on the true value

- As the early exercise strategy only uses information that would be available to the option holder at the time of exercise, it can be no better than the best possible (i.e. optimal) strategy based on available information.
- The quality of the lower bound depends on the choice of basis functions.

The lower bound algorithm: Notation

- Consider a set of financial variables $X_i, i = 1, ..., N$, observed at times $T_k, k = 0, ..., K$.
- Denote by $T_{\bar{k}(j)}$, $0 \leq \bar{k}(j) \leq K$ the j-th time of potential exercise of an option
- For each early exercise time $T_{\bar{k}(j)}$, $0 \leq \bar{k}(j) < K$, approximate the continuation value by a linear combination of n basis functions,

$$V(T_{\bar{k}(j)}) = \sum_{\ell=1}^{n} c_{\ell j} \psi_{lj} \tag{1}$$

$$\psi_{\ell j} := \psi_\ell \left(T_{\bar{k}(j)}, \{X_i(T_k)\}_{\substack{i=1,\ldots,N, \\ k=0,\ldots,\bar{k}(j)}} \right), \quad \ell = 1, \ldots, n$$

The lower bound algorithm I

Following Longstaff and Schwartz (2001), estimate $c_{\ell j}$ for each $T_{\bar{k}(j)}$ by

Step 1: Generate `npaths` simulated Monte Carlo paths of the financial variables X_i observed at times T_k. (These paths will be used to estimate the $c_{\ell j}$ and then discarded.)

Step 2: Initialise a vector y of length `npaths` with the option payoff at time T_K on each path, discounted to the *last* time point $T_{\bar{k}(J)} < T_K$ of potential early exercise. Set the "current" time $t = T_{\bar{k}(J)}$. The initial value of the index j is J (i.e. we will be proceeding by backward induction).

Step 3: Calculate a vector e of length `npaths`, containing the payoff on each path if the option is exercised at time t.

The lower bound algorithm II

Step 4: Determine the $c_{\ell j}$ for all $\ell = 1, \ldots, n$ and the current j by least squares fit, i.e. choose the $c_{\ell,j}$ to minimise

$$\sum_{h=1}^{\texttt{npaths}} (y_h - V^{(h)}(T_{\bar{k}(j)}))^2 \tag{2}$$

where $V^{(h)}(T_{\bar{k}(j)})$ is given by (1) calculated using the realisations of $X_i(T_k)$ on the h-th path.

Step 5: Using the thus determined $c_{\ell j}$, set each y_h to $\max(e_h, V^{(h)}(T_{\bar{k}(j)}))$ discounted to time $T_{\bar{k}(j-1)}$.

Step 6: If $T_{\bar{k}(j)}$ is the earliest time of potential early exercise, stop. Otherwise, set $j = j - 1$ and $t = T_{\bar{k}(j)}$, and go to Step 3.

Longstaff & Schwartz's example

```
Array<double,2> paths(8,4);
paths = 1.00, 1.09, 1.08, 1.34,
        1.00, 1.16, 1.26, 1.54,
        1.00, 1.22, 1.07, 1.03,
        1.00, 0.93, 0.97, 0.92,
        1.00, 1.11, 1.56, 1.52,
        1.00, 0.76, 0.77, 0.90,
        1.00, 0.92, 0.84, 1.01,
        1.00, 0.88, 1.22, 1.34;
Array<double,1> T(4);
T = 0.0, 1.0, 2.0, 3.0;
double K = 1.10;
Payoff put(K,-1);
boost::function<double (double)> f;
f = boost::bind(std::mem_fun(&Payoff::operator()),&put,_1);
FlatTermStructure ts(0.06,0.0,10.0);
Array<double,3> genpaths(8,4,1);
genpaths(Range::all(),Range::all(),0) = paths;
Array<double,2> numeraire_values(8,4);
for (i=0;i<4;i++) numeraire_values(Range::all(),i) = 1.0/ts(T(i));
boost::function<double (double,const Array<double,1>&)>
  payoff = boost::bind(LSArrayAdapter,_1,_2,f,0);
```

Longstaff & Schwartz's example

```
vector<boost::function<double (double,const Array<double,1>&)> >
  basisfunctions;
int degree = 2;
Array<int,1> p(1);
for (i=0;i<=degree;i++) {
  p(0) = i;
  add_polynomial_basis_function(basisfunctions,p); }
LongstaffSchwartzExerciseBoundary
genls(T,genpaths,numeraire_values,payoff,basisfunctions);
Array<double,2> genpath(4,1);
Array<double,1> num(4);
for (i=0;i<8;i++) {
  genpath = genpaths(i,Range::all(),Range::all());
  num = numeraire_values(i,Range::all());
  MCestimate += genls.apply(genpath,num); }
std::cout << MCestimate.mean() << std::endl;
```

Early exercise and the generic MC templates

```
LSExerciseStrategy<LongstaffSchwartzExerciseBoundary>
  exercise_strategy(boundary);
MCMapping<GeometricBrownianMotion,Array<double,2> >
  mc_mapping(exercise_strategy,gbm,ts,numeraire_index);
boost::function<double (Array<double,2>)> func =
  boost::bind(&MCMapping<GeometricBrownianMotion,
                         Array<double,2> >::mapping,
              &mc_mapping,_1);
MCGeneric<Array<double,2>,double,
          RandomArray<ranlib::NormalUnit<double>,double> >
  mc(func,random_container);
MCGatherer<double> mcgatherer;
size_t n = minpaths;
boost::math::normal normal;
double d = boost::math::quantile(normal,0.95);
cout << "Paths,MC value,95% CI lower bound,95% CI upper bound"
     << endl;
while (mcgatherer.number_of_simulations()<maxpaths) {
  mc.simulate(mcgatherer,n);
  cout << mcgatherer.number_of_simulations() << ','
       << mcgatherer.mean() << ','
       << mcgatherer.mean()-d*mcgatherer.stddev() << ','
       << mcgatherer.mean()+d*mcgatherer.stddev() << endl;
  n = mcgatherer.number_of_simulations(); }
```

Lower bounds with different basis functions

	Polynomial only			Incl. European put		
Paths	MC price	95% conf. bounds		MC price	95% conf. bounds	
10240	12.357	12.1589	12.5551	12.4288	12.2224	12.6352
20480	12.3821	12.2416	12.5225	12.4645	12.3181	12.6109
40960	12.472	12.3725	12.5715	12.538	12.4343	12.6417
81920	12.4689	12.3986	12.5392	12.5196	12.4464	12.5929
163840	12.4582	12.4086	12.5079	12.5054	12.4537	12.5571
327680	12.4808	12.4457	12.516	12.5312	12.4945	12.5678
655360	12.5039	12.4791	12.5288	12.5525	12.5266	12.5784
1310720	12.4976	12.4801	12.5152	12.5442	12.5259	12.5625

Pseudo-random vs. quasi-random I

- The pseudo–random Monte Carlo estimate converges with $\mathcal{O}(N^{-\frac{1}{2}})$.
- Quasi–random Monte Carlo seeks to improve on this convergence by deterministically filling the unit hypercube "more evenly."
- Acceleration of convergence to nearly $\mathcal{O}(N^{-1})$ — herein lies the difference to the variance reduction techniques, which only affected the implicit constant in $\mathcal{O}(N^{-\frac{1}{2}})$ convergence.

Filling “more evenly”

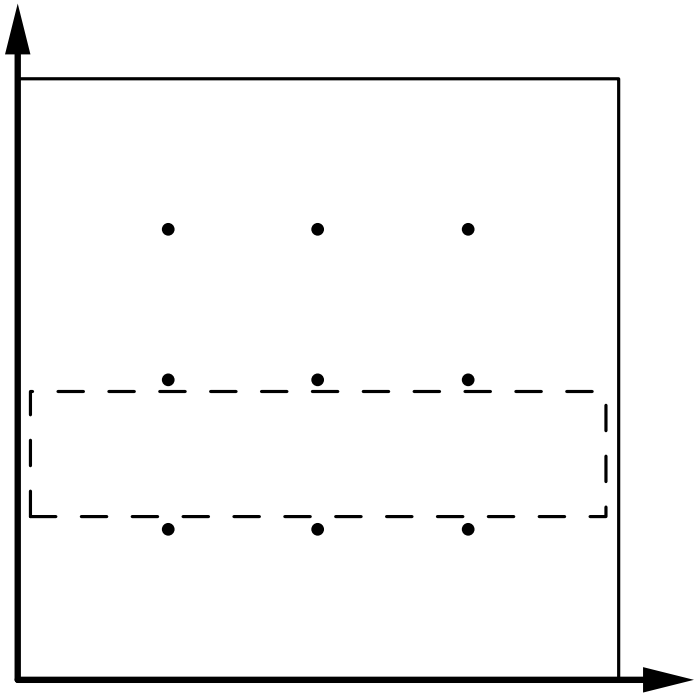


Figure 7.4 *A grid of points in the unit square: the dashed box is completely empty of points*

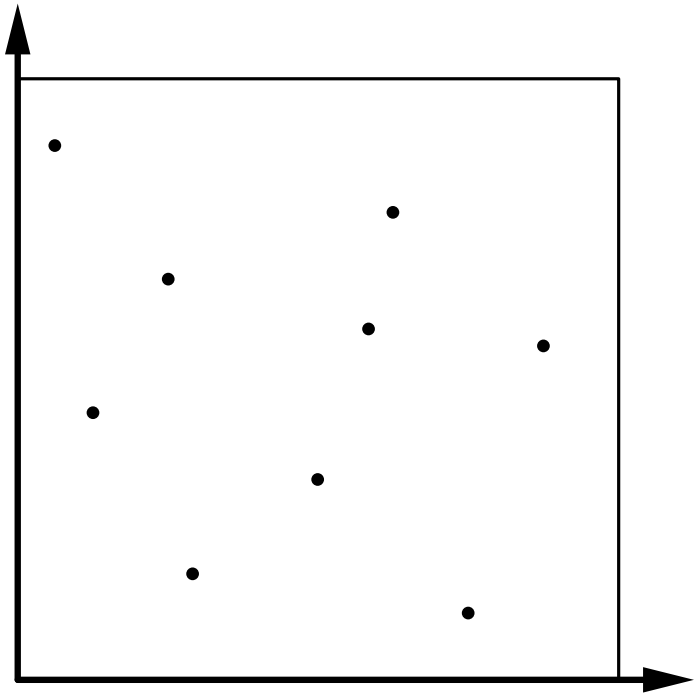


Figure 7.5 *Better coverage of the unit square*

Pseudo-random vs. quasi-random II

- Quasi–random Monte Carlo explicitly depends on the problem dimension.
- In pseudo–random Monte Carlo a single univariate random number generator was sufficient to generate all the required independent random variates for a sample path.
- In quasi–random Monte Carlo, two consecutive elements of the quasi–random sequence are not independent.

Pseudo-random vs. quasi-random III

- One must distinguish between the set of random variates making up a single random vector u_i required to produce a realisation of the simulation $f(u_i)$, and consecutive realisations of the random vector u_i drawn N times to calculate the Monte Carlo estimate.
- The dimension d of the u_i is the required dimension of the quasi–random sequence, while N is the length of the sequence.

Randomised quasi–random Monte Carlo I

- It is difficult to obtain a meaningful error estimate through quasi–random methods alone, but a particularly simple way of getting around this problem is to randomise (in a pseudo–random sense) over a set of quasi–random simulation runs.
- A simple way a quasi–random sequence can be randomised is by shifting each point in the sequence by the same random amount v, uniform on $[0,1]^d$.

Randomised quasi–random Monte Carlo II

- Coordinate values greater that one are "wrapped" using modulo–1 division, i.e. if $u_i^{(j)}$ is the j–th coordinate of the i–th element in the original sequence, its randomised counterpart $\hat{u}_i^{(j)}$ is given by

$$\hat{u}_i^{(j)} = (u_i^{(j)} + v^{(j)}) \mod 1 \tag{3}$$

where $v^{(j)}$ is the j–th coordinate of v.

We now have

- quasi–random Monte Carlo (simulation using the sequence $\hat{u}$) nested inside pseudo–random Monte Carlo (repeated draws of v)
- a meaningful standard deviation of the Monte Carlo estimate (which is now an average over the different quasi–random Monte Carlo estimates for each $\hat{u}$) can be obtained at the level of the outer loop.

Convergence example

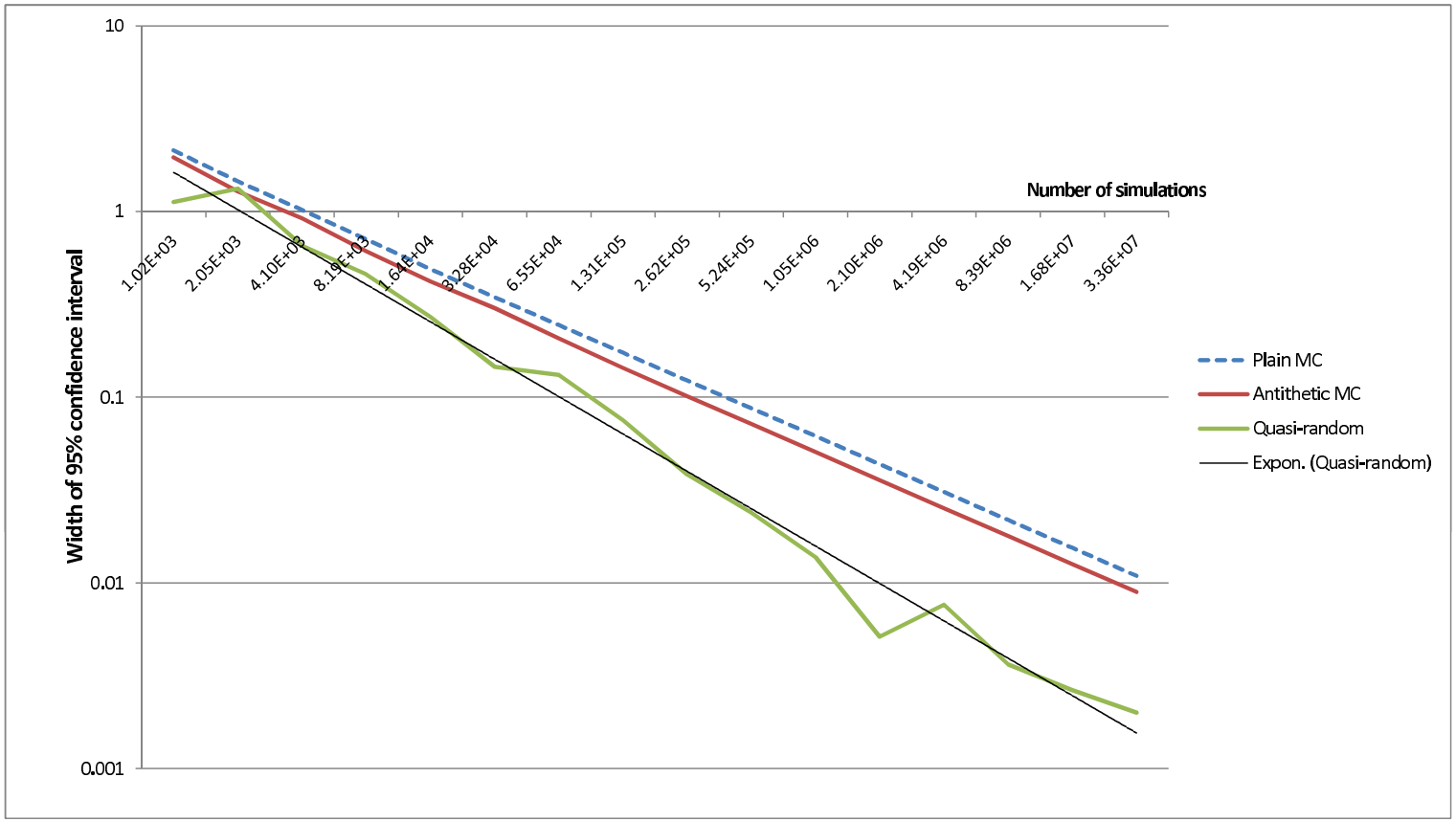


Figure 7.6 *Convergence of confidence intervals for Monte Carlo estimate of Margrabe option price, log/log scale, trendline added for quasi-random case.*

Example with control variates

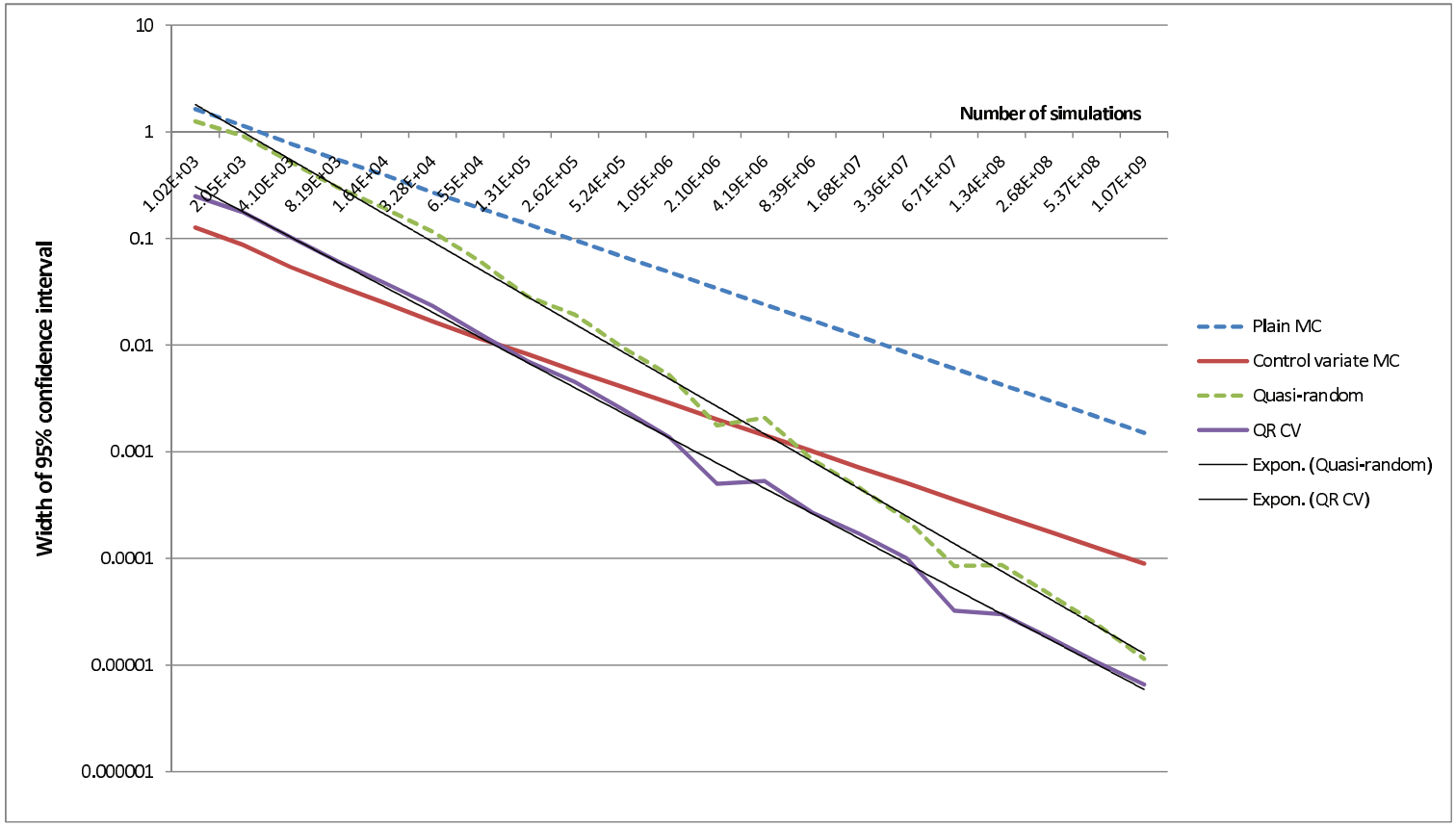


Figure 7.7 *Convergence of confidence intervals for Monte Carlo estimate of Margrabe option price, with and without control variates (CV), log/log scale, trendline added for quasi-random (QR) cases.*

High–level summary I

- Up to now, the simulations were implemented using the stochastic process class `GeometricBrownianMotion` (representing a multi–asset Black/Scholes model under deterministic interest rates).
- This can be replaced by a class `GaussMarkovWorld` (representing a multi–asset, multi–currency Gauss/Markov Heath/Jarrow/Morton model).
- It is convenient to simulate zero coupon bond prices and *forward* asset prices and exchange rates, meaning that payoffs have to be appropriately defined.

High–level summary II

- All other building blocks remain unchanged (e.g. `MCGatherer`, `MCGeneric`, `MCMapping`, the Longstaff/Schwartz implementation, etc.).
- Extension to the LIBOR Market Model (not yet implemented) would only require the implementation of the appropriate class to replace `GaussMarkovWorld`.

Max. polynomial deg.			Training	MC	Max. polynomial deg.			Training	MC
Fwd	ZCB	Spot	paths	value	Fwd	ZCB	Spot	paths	value
0	0	2	1000	49.60	0	2	2	1000	52.02
0	0	2	10000	49.65	0	2	2	10000	52.04
0	0	2	200000	49.66	0	2	2	200000	52.08
0	2	0	1000	48.95	2	0	2	1000	50.56
0	2	0	10000	48.70	2	0	2	10000	50.33
0	2	0	200000	48.61	2	0	2	200000	50.62
2	0	0	1000	42.01	2	2	0	1000	47.76
2	0	0	10000	47.04	2	2	0	10000	50.91
2	0	0	200000	50.06	2	2	0	200000	51.35
2	2	2	1000	51.99	1	2	2	1000	52.04
2	2	2	10000	52.07	1	2	2	10000	52.07
2	2	2	200000	52.11	1	2	2	200000	52.11
3	3	3	1000	50.46	1	2	3	1000	51.34
3	3	3	10000	38.65	1	2	3	10000	49.70
3	3	3	200000	45.07	1	2	3	200000	50.78

Table 8.3 *Suboptimal values for an American put option based on Longstaff/Schwartz exercise strategies with different polynomial basis functions and different numbers of training paths. The MC values are based on 3276800 paths with 60 time steps on each path; the 95% confidence intervals are* ± 0.07 *around the given value. The price of the corresponding European option is 46.66.*

Martingale measures and real–world probabilities

Well–known results of Financial Economics:

- Any asset with a strictly positive price process is a valid numeraire.
- For every numeraire S, in an arbitrage–free market there exists at least one probability measure Q_S equivalent to the real–world measure P, such that all (traded) asset prices discounted by S are Q_S–martingales.

The Numeraire Portfolio

Perhaps less well–known, the converse is also true:

For every probability measure Q_H equivalent to the real–world measure P, there exists a self–financing portfolio strategy with a value process H that is strictly positive almost surely, such that all (traded) asset prices discounted by H are Q_H–martingales.

Note that

Since P is trivially equivalent to P, there exists a portfolio strategy $\hat{H}$ such that all (traded) asset prices discounted by $\hat{H}$ are martingales under the real–world measure. $\hat{H}$ is commonly called the **numeraire portfolio** or **deflator**.

Properties of the Numeraire Portfolio

- If there are no redundant assets, the numeraire portfolio is unique.
- The numeraire portfolio exists even in an incomplete market — however, only *traded* asset prices discounted by $\hat{H}$ are necessarily martingales under P.
- Adding further, non-redundant assets typically changes the composition of the numeraire portfolio.
- When the numeraire portfolio is defined as (its density of) weights in the Arrow/Debreu securities, these weights can be identified with the real–world probability measure.
- $\hat{H}$ maximises the expected growth rate of the portfolio value and the expected logarithm of terminal wealth. It minimises the expected time needed to reach a given wealth level for the first time.

Characterisation of the numeraire portfolio in continuous time

Assume that asset dynamics are given by

$$dS_i(t) = S_i(t)(\mu_i dt + \sigma^{(i)} dW(t))$$

where $W(t)$ is a d–dimensional Brownian motion under the real–world measure.

Switching numeraires from the numeraire portfolio $\hat{H}(T)$ to the continuously compounded savings account $\beta(T)$, we have

$$\left.\frac{dP}{dQ_\beta}\right|_{\mathcal{F}_T} = \frac{\hat{H}(T)\beta(0)}{\hat{H}(0)\beta(T)}$$

Market price of risk

This is the Q_β–martingale

$$\exp\left\{\int_0^{\cdot} \sigma_{\hat{H}} dW_\beta(t) - \frac{1}{2}\int_0^{\cdot} \sigma_{\hat{H}}^2 dt\right\}$$

Thus by Girsanov's theorem P and Q_β are related via

$$dW(t) = dW_\beta(t) - \sigma_{\hat{H}} dt$$

and we can write

$$dS_i(t) = S_i(t)(\mu_i dt + \sigma^{(i)}(dW_\beta(t) - \sigma_{\hat{H}} dt))$$

We must therefore have

$$\mu_i - \sigma_{\hat{H}}\sigma^{(i)} = r$$

and we may interpret $\sigma_{\hat{H}}$ as the market price of risk.

Implementation

This is implemented in the C++ code, though not explicitly discussed in the book:

`GaussMarkovWorld` implements a member function

```
set_numeraire(int num);
```

which allows the selection of a simulated asset as the numeraire (and thus the stochastic dynamics are generated under the appropriate measure, with the appropriate drifts).

Selecting as the numeraire an asset with a volatility (vector) equal to the market price of risk results in simulation under the physical measure.

Further reading

This talk was mainly based on this book:

For more information, and to download the C++ code (released under a Modified BSD License), visit:

`www.schlogl.com/QF`