Implementing QuantLib

Luigi Ballabio

6 The Monte Carlo framework

Great expectations

THE CONCEPT behind Monte Carlo methods is deceptively simple—generate random paths, price the desired instrument over such paths, and average the results. However (and even assuming that the tasks I just enumerated were as simple as they seem: they're not) several choices can be made regarding the implementation of the model. Pseudo-random or quasi-random numbers? Which ones? When should the simulation stop? What results should be returned? How about more advanced techniques such as likelihood ratio or pathwise Greeks?

The mission of a library writer, should he accept it, is to make most (possibly all) such choices independently available to the user; to make it possible to model the underlying stochastic process in a polymorphic way; and on top of it all, to provide at least part of the scaffolding needed to put everything together and run the simulation.

The first part of this chapter describes the design of our Monte Carlo framework and in what measure it meets (or falls short of) the above requirements; the second part outlines how it can be integrated with the pricing-engine framework described in chapter 2.

6.1. Path generation

The task of generating random paths puts together a number of building blocks and concepts; in our implementation, it also puts together a number of programming techniques. In this section, I'll describe such blocks and how they ultimately yield a path-generator class.

6.1.1. Random-number generation

Among the basic building blocks for our task are random-number generators (RNG), the most basic being uniform RNGs. The library provides quite a few of them, the most notable being the Mersenne Twister among pseudo-random generators and Sobol among low-discrepancy ones. Here, I'll gloss over their implementation, as well as over when and how you should use either kind of generator; such information (as well as everything else related to Monte Carlo

Listing 6.1: Common interface of random-number generator classes.

```
template <class T>
struct Sample {
    typedef T value_type;
    T value;
    Real weight;
};

class MersenneTwisterUniformRng {
  public:
    typedef Sample<Real> sample_type;
    explicit MersenneTwisterUniformRng(unsigned long seed = 0);
    sample_type next() const;
};

class SobolRsg {
  public:
    typedef Sample<std::vector<Real> > sample_type;
    SobolRsg(Size dimensionality,
             unsigned long seed = 0);
    const sample_type& nextSequence() const;
    Size dimension() const { return dimensionality_; }
};
```


methods) is covered in literature much better than I ever could, for instance in Glasserman [14] or Jäckel [18].

The interface of uniform RNGs, shown in listing 6.1, doesn't make for a very exciting reading. Its methods are the ones you would expect: pseudo-random generators sport a constructor that takes an initialization seed and a method, `next`, that returns the next random value; low-discrepancy generators take an additional constructor argument that specify their dimensionality and return a vector of values from their `nextSequence` method.[1] The only feature of note is the `Sample` struct, used to return the generated values, which also contains a weight. This was done to make it possible for generators to sample values with some kind of bias (e.g., when using importance sampling). However, at this time the library doesn't contain any such generator; all available RNGs return samples with unit weight.

So much for uniform RNGs. However, they're only the rawest material; we need to refine them further. This is done by means of classes (wrappers, or combinators,

[1] On second thought, we might have called the `nextSequence` method simply `next`. Including the "sequence" token in the method name reminds me of Hungarian notation, which is up there with `goto` in the list of things considered harmful.

Listing 6.2: Sketch of the `RandomSequenceGenerator` class template.

```
template<class RNG>
class RandomSequenceGenerator {
  public:
    typedef Sample<std::vector<Real> > sample_type;
    RandomSequenceGenerator(Size dimensionality,
                            const RNG& rng)
    : dimensionality_(dimensionality), rng_(rng),
      sequence_(std::vector<Real>(dimensionality), 1.0) {}
    const sample_type& nextSequence() const {
        sequence_.weight = 1.0;
        for (Size i=0; i<dimensionality_; i++) {
            typename RNG::sample_type x(rng_.next());
            sequence_.value[i] = x.value;
            sequence_.weight  *= x.weight;
        }
        return sequence_;
    }
  private:
    Size dimensionality_;
    RNG rng_;
    mutable sample_type sequence_;
};
```


if you will; or again, you might see them as implementations of the Decorator pattern) that take an instance of a uniform RNG and yield a different generator. We need two kinds of decorators. On the one hand, we need tuples of random numbers rather than single ones: namely, $M \times N$ numbers in order to evolve M variables for N steps. Low-discrepancy generators return tuples already (of course they need to do so, for their low-discrepancy property to hold at the required dimensionality). Since we want a generic interface, we'll make the pseudo-random generators return tuples as well, by simply drawing from them repeatedly; the class template that does so, called `RandomSequenceGenerator`, is shown in listing 6.2.

On the other hand, we usually want Gaussian random numbers, not uniform ones. There's a number of ways to obtain them, but not all of them might be equally suited for any given situation; for instance, those consuming M uniform random numbers to yield N Gaussian ones don't work with low-discrepancy generators (especially if M varies from draw to draw, like in rejection techniques). If you want to stay generic, the simplest method is probably to feed your uniform numbers to the inverse-cumulative Gaussian function, which yield a Gaussian per uniform; this is also the default choice in the library. The corresponding class template,

Listing 6.3: Sketch of the `InverseCumulativeRsg` class template.

```
template <class USG, class IC>
class InverseCumulativeRsg {
  public:
    typedef Sample<std::vector<Real> > sample_type;
    InverseCumulativeRsg(const USG& uniformSequenceGenerator,
                         const IC& inverseCumulative);
    const sample_type& nextSequence() const  {
        typename USG::sample_type sample =
            uniformSequenceGenerator_.nextSequence();
        x_.weight = sample.weight;
        for (Size i = 0; i < dimension_; i++) {
            x_.value[i] = ICD_(sample.value[i]);
        }
        return x_;
    }
  private:
    USG uniformSequenceGenerator_;
    Size dimension_;
    mutable sample_type x_;
    IC ICD_;
};
```


`InverseCumulativeRsg`, is shown in listing 6.3. It takes the implementation of the inverse-cumulative function as a template argument, since there's a few approximations and no closed formula for it; most of the times you'll be fine with the one provided by the library, but at times one might choose to change it in order to trade accuracy for speed (or the other way around).

At this point, you might have started to have bad feelings about this framework. I have only covered random numbers, and yet we already have two of three choices to make in order to instantiate a generator. By the time we're done, we might have a dozen of template arguments on our hands. Well, bear with me for a few more pages. In section 6.3, I'll describe how the library tries to mitigate the problem.

A final note: as you might have noticed, the sequence generators shown in the listings keep a mutable sequence as a data member and fill it with the new draws in order to avoid copies. Of course, this prevents instances of such classes to be shared by different threads—although I'd be a very happy camper if all the problems QuantLib had with multithreading were like this one.

6.1.2. Stochastic processes

The whole point of using RNGs is, of course, to use the generated random numbers to drive a stochastic process. The interface of the `StochasticProcess` class (displayed in listing 6.4) was designed with this application in mind—and it shows. The concept modeled by the class was a somewhat generic n-dimensional stochastic process described by the equation

$$d\mathbf{x} = \mu(t, \mathbf{x})dt + \sigma(t, \mathbf{x}) \cdot d\mathbf{w}$$

(yes, it took six chapters, but there's an equation in this book. I was a physicist once, you know). However, the interface of the class deals more with the discretization of the process over the time steps of a sampled path.

Straight away, the class defines a polymorphic inner class `discretization`. It is an instance of the Strategy pattern, whose purpose is o make it possible to change the way a process is discretely sampled. Its methods take a `StochasticProcess` instance, a starting point $(t, \mathbf{x})$, and a time step Δt and returns a discretization of the process drift and diffusion.[2]

Back to the `StochasticProcess` interface. The class defines a few inspectors. The `size` method returns the number of variables modeled by the process. For instance, it would return 1 for a regular Black-Scholes process with a single underlying, or 2 for a stochastic-volatility model, or N for the joint process of N correlating underlying assets each following a Black-Scholes process. The `factors` method returns the number of random variates used to drive the variables; it has a default implementation returning the same number as `size`, although on afterthought I'm not sure that it was a great idea; we might have required to specify that explicitly. To round up the inspectors, the `initialValues` method returns the values of the underlying variables at $t = 0$; in other words, the present values, in

[2] by this time, you'll have surely noticed that there's no provision for jumps. Yes, I know. Hold that thought; I'll have more to say on this later.

Aside: the road more traveled.

The upcoming C++ standard will include RNGs as part of its standard library, based on the existing Boost implementation. I don't know if, at some point, we'll switch to the those RNGs, nor how we'll do it. We might ditch our implementation and switch to the standard classes; or we could use the new classes internally, to replace the implementation of our classes; or, again, we could provide adaptors between the two interfaces. In any case, it's likely that the changes will break backward compatibility, which means that they'll have to wait for an entirely new revision of QuantLib.

Listing 6.4: Partial implementation of the `StochasticProcess` class.

```
class StochasticProcess : public Observer,
                          public Observable {
  public:
    class discretization {
      public:
        virtual ~discretization() {}
        virtual Disposable<Array> drift(
                                    const StochasticProcess&,
                                    Time t0, const Array& x0,
                                    Time dt) const = 0;
        // same for diffusion etc.
    };

    virtual ~StochasticProcess() {}

    virtual Size size() const = 0;
    virtual Size factors() const;
    virtual Disposable<Array> initialValues() const = 0;

    virtual Disposable<Array> drift(Time t,
                                    const Array& x) const = 0;
    virtual Disposable<Matrix> diffusion(Time t,
                                         const Array& x)
                                                   const = 0;
    virtual Disposable<Array> expectation(Time t0,
                                          const Array& x0,
                                          Time dt) const;
    virtual Disposable<Matrix> stdDeviation(Time t0,
                                            const Array& x0,
                                            Time dt) const;
    virtual Disposable<Matrix> covariance(Time t0,
                                          const Array& x0,
                                          Time dt) const;
    virtual Disposable<Array> evolve(Time t0,
                                     const Array& x0,
                                     Time dt,
                                     const Array& dw) const;
    virtual Disposable<Array> apply(const Array& x0,
                                    const Array& dx) const;
};
```

Listing 6.4 (continued).

```
Size StochasticProcess::factors() const {
    return size();
}

Disposable<Array>
StochasticProcess::expectation(Time t0,
                               const Array& x0,
                               Time dt) const {
    return apply(x0, discretization_->drift(*this,t0,x0,dt));
}

Disposable<Matrix>
StochasticProcess::stdDeviation(Time t0,
                                const Array& x0,
                                Time dt) const {
    return discretization_->diffusion(*this,t0,x0,dt);
}

Disposable<Matrix>
StochasticProcess::covariance(Time t0,
                              const Array& x0,
                              Time dt) const {
    return discretization_->covariance(*this,t0,x0,dt);
}

Disposable<Array>
StochasticProcess::evolve(Time t0, const Array& x0,
                          Time dt, const Array& dw) const {
    return apply(expectation(t0,x0,dt),
                 stdDeviation(t0,x0,dt)*dw);
}

Disposable<Array>
StochasticProcess::apply(const Array& x0,
                         const Array& dx) const {
    return x0 + dx;
}
```

which no randomness is involved. As in most `StochasticProcess` methods, the result is wrapped in a `Disposable` class template. This is a way to avoid a few copies when returning arrays or matrices; if you're interested in the details, you can have a look at appendix A. When we can rely on compilers supporting the C++0X standard, we'll be able to replace those with r-value references [15, 1]; but as usual, it will be a few years before we can do this (where "this" also includes being able to drop support for older compilers with a clear conscience, and without leaving users outside in the rain).

The next pair of methods (also inspectors, in a way) return more detailed information on the process. The `drift` method return the $\mu(t, \mathbf{x})$ term in the previous formula, while the `diffusion` method returns the $\sigma(t, \mathbf{x})$ term. Obviously, for dimensions to match, the drift term must be a N-dimensional array and the diffusion term a $M \times N$ matrix, with M and N being the dimensions returned by he `factors` and `size` methods, respectively. Also, the diffusion matrix must include correlation information.

The last set of methods deals with the discretization of the process over a path. The `expectation` method returns the expectation values of the process variables at time $t + \Delta t$, assuming they had values $\mathbf{x}$ at time t. Its default implementation asks the stored `discretization` instance for the integrated drift of the variables over Δt and applies it to the starting values (`apply` is another virtual method that takes a value $\mathbf{x}$ and a variation $\Delta\mathbf{x}$ and unsurprisingly returns $\mathbf{x} + \Delta\mathbf{x}$. Why we needed a polymorphic method to do this, you ask? It's a sad story that will be told in a short while). The `stdDeviation` method returns the diffusion term integrated over $t + \Delta t$ and starting at $(t, \mathbf{x})$, while the `covariance` method returns its square; in their default implementation, they both delegate the calculation to the `discretization` instance. Finally, the `evolve` method provides some kind of higher-level interface; it takes a starting point $(t, \mathbf{x})$, a time step Δt, and an array $\mathbf{w}$ of random Gaussian variates, and returns the end point $\mathbf{x}'$. Its default implementation asks for the standard-deviation matrix, multiplies it by the random variates to yield the random part of the simulated path, and applies the resulting variations to the expectation values of the variables at $t + \Delta t$ (which includes the deterministic part of the step).

As you've seen, the methods in this last set are all given a default implementation, sometimes calling methods like `drift` or `diffusion` by way of the discretization strategy and sometimes calling directly other methods in the same set; it's a kind of multiple-level Template Method pattern, as it were. This makes it possible for classes inheriting from `StochasticProcess` to specify their behavior at different levels.

At the very least, such a class must implement the `drift` and `diffusion` methods, which are purely virtual in the base class. This specifies the process at the innermost level, and is enough to have a working stochastic process; the other methods would use their default implementation and delegate the needed

calculations to the contained `discretization` instance. The derived class can prescribe a particular discretization, suggest one by setting it as a default, or leave the choice entirely to the user.

At a somewhat outer level, the process class can also override the `expectation` and `stdDeviation` methods; this might be done, for instance, when it's possible to write an closed formula for their results. In this case, the developer of the class can decide either to prevent the constructor from taking a `discretization` instance, thus forcing the use of the formula; or to allow the constructor to take one, in which case the overriding method should check for its presence and possibly forward to the base-class implementation instead of using its own. The tasks of factoring in the random variates and of composing the integrated drift and diffusion terms are still left to the `evolve` method.

At the outermost level, the derived class can override the `evolve` method and do whatever it needs—even disregard the other methods in the `StochasticProcess` interface and rely instead on ones declared in the derived class itself. This can serve as a Hail Mary pass for those process which don't fit the interface as currently defined. For instance, a developer wanting to add stochastic jumps to a process might add them into `evolve`, provided that a random-sequence generator can generate the right mixture of Gaussian and Poisson variates to pass to the method.

Wouldn't it be better to support jumps in the interface? Well, yes, in the sense that it would be better if a developer could work *with* the interface rather than *around* it. However, there are two kinds of problems that I see about adding jumps to the `StochasticProcess` class at this time.

On the one hand, I'm not sure what the right interface should be. For instance, can we settle on a single generalized formula including jumps? I wouldn't want to release an interface just to find out later that it's not the correct one; which is likely, given that at this time we don't have enough code to generalize from.

On the other hand, what happens when we try to integrate the next feature? Do we add it to the interface, too? Or should we try instead to generalize by not specializing; that is, by declaring fewer methods in the base-class interface, instead of more? For instance, we could keep `evolve` or something like it, while moving `drift`, `diffusion` and their likes in subclasses instead (by the way, this might be a way to work on a jump interface without committing too early to it: writing some experimental classes that define the new methods and override `evolve` to call them. Trying to make them work should provide some useful feedback).

But I went on enough already. In short: yes, Virginia, the lack of jumps in the interface is a bad thing. However, I think we'd be likely to make a poor job if we were to add them now. I, for one, am going to stay at the window and see if anybody comes along with some working code. If you wanted to contribute, those experimental classes would be most appreciated.

One last detour before we tackle a few examples. When implementing a one-dimensional process, the `Array`s and their likes tend to get in the way; they make

Listing 6.5: Partial implementation of the `StochasticProcess1D` class.

```
class StochasticProcess1D : public StochasticProcess {
  public:
    class discretization {
      public:
        virtual ~discretization() {}
        virtual Real drift(const StochasticProcess1D&,
                           Time t0, Real x0, Time dt) const = 0;
        // same for diffusion etc.
    };
    virtual Real x0() const = 0;
    virtual Real drift(Time t, Real x) const = 0;
    virtual Real diffusion(Time t, Real x) const = 0;
    virtual Real expectation(Time t0, Real x0, Time dt) const {
        return apply(x0, discretization_->drift(*this, t0,
                                                x0, dt));
    }
    virtual Real stdDeviation(Time t0, Real x0, Time dt) const {
        return discretization_->diffusion(*this, t0, x0, dt);
    }
    virtual Real variance(Time t0, Real x0, Time dt) const;
    virtual Real evolve(Time t0, Real x0,
                        Time dt, Real dw) const {
        return apply(expectation(t0,x0,dt),
                     stdDeviation(t0,x0,dt)*dw);
    }
  private:
    Size size() const { return 1; }
    Disposable<Array> initialValues() const  {
        return Array a(1, x0());
    }
    Disposable<Array> drift(Time t, const Array& x) const {
        return Array a(1, drift(t, x[0]));
    }
    // ... the rest of the StochasticProcess interface...
    Disposable<Array> StochasticProcess1D::evolve(
                               Time t0, const Array& x0,
                               Time dt, const Array& dw) const {
        return Array(1, evolve(t0,x0[0],dt,dw[0]));
    }
};
```

the code more tiresome to write and less clear to read. Ideally, we'd want an interface like the one declared by `StochasticProcess` but with methods that take and return simple real numbers instead of arrays and matrices. However, we can't declare such methods as overloads in `StochasticProcess`, since they don't apply to a generic process; and neither we wanted to have a separate hierarchy for 1-D processes, since they belong with all the others.

The library solves his problem by providing the `StochasticProcess1D` class, shown in listing 6.5. On the one hand, this class declares an interface that mirrors exactly the `StochasticProcess` interface, even down to the default implementations, but uses the desired `Real` type. On the other hand, it inherits from `StochasticProcess` (establishing that a 1-D process "is-a" stochastic process, as we wanted) and implements its interface in terms of the new one by boxing and unboxing `Real`s into and from `Array`s, so that the developer of a 1-D process needs not care about it: in short, a straightforward application of the Adapter pattern. Because of the 1:1 correspondence between scalar and vectorial methods (as can be seen in the listing) the 1-dimensional process will work the same way when accessed from either interface; also, it will have the same possibilities of customization provided by the `StochasticProcess` class and described earlier in this section.

And now, an example or three. Not surprisingly, the first is the one-dimensional Black-Scholes process. You've already met briefly the corresponding class (by the unwieldy name of `GeneralizedBlackScholesProcess`) in chapter 2, where it was passed as an argument to a pricing engine for a European option; I'll go back to that code later on, when I discuss a few shortcomings of the current design and possible future solutions. For the time being, let's have a look at the current implementation, sketched in listing 6.6.

Well, first of all, I should probably explain the length of the class name. The "generalized" bit doesn't refer to some extension of the model, but to the fact that this class was meant to cover different specific processes (such as the Black-Scholes process proper, the Black-Scholes-Merton process, the Black process and so on); the shorter names were reserved for the specializations.

The constructor of this class takes handles to the full set of arguments needed for the generic formulation: a term structure for the risk-free rate, a quote for the varying market value of the underlying, another term structure for its dividend yield, and a term structure for its implied Black volatility. In order to implement the `StochasticProcess` interface, the constructor also takes a `discretization` instance. All finance-related inputs are stored in the constructed class instance and can be retrieved by means of inspectors.

You might have noticed that the constructor takes a Black volatility $\sigma_B(t,k)$, which is not what is needed to return the $\sigma(t,x)$ diffusion term. The process

performs the required conversion internally,[3] using different algorithms depending on the type of the passed Black volatility (constant, depending on time only, or with a smile). The type is checked at run-time with a dynamic cast, which is not pretty but in this case is simpler than a Visitor pattern. At this time, the conversion algorithms are built into the process and can't be changed by the user; nor it is possible to provide a precomputed local volatility.

The next methods, `drift` and `diffusion`, show the crux of this class: it's actually a process for the logarithm of the underlying that masquerades as a process for the underlying itself. Accordingly, the term returned by `drift` is the one for the logarithm, namely, $r(t) - q(t) - \frac{1}{2}\sigma^2(t,x)$, where the rates and the local volatility are retrieved from the corresponding term structures (this is another problem in itself, to which I'll return). The `diffusion` method returns the local volatility.

The choice of $\log(x)$ as the actual stochastic variable brings quite a few consequences. First of all, the variable we're using is not the one we're exposing. The quote we pass to the constructor holds the market value of the underlying; the same value is returned by the `x0` method; and the values passed to and returned from other methods such as `evolve` are for the same quantity. In retrospect, this was a bad idea (yes, I can hear you say "D'oh.") We were led to it by the design we had developed, but it should have been a hint that some piece was missing.

Other consequences can be seen all over the listing. The choice of $\log(x)$ as the working variable is the reason why the `apply` method is virtual, and indeed why it exists at all: applying to x a drift Δ for the drift is not done by adding them, but by returning $x\exp(\Delta)$. The `expectation` method is affected, too: its default implementation would apply the drift as above, thus returning $\exp(E[\log(x)])$ instead of $E[x]$. To prevent it from returning the wrong value, the method is currently disabled (it raises an exception). In turn, the `evolve` method, whose implementation relied on `expectation`, had to be overridden to work around it.[4]

On top of the above smells, we have a performance problem—which is known to all those that tried a Monte Carlo engine from the library, as well as to all the people on the Wilmott forums to whom the results were colorfully described. Using `apply`, the `evolve` method performs an exponentiation at each step; but above all, the `drift` ad `diffusion` methods repeatedly ask term structures for values. If you remember chapter 3, this means going through at least a couple of levels of virtual method calls, sometimes (if we're particularly unlucky) retrieving a rate from discount factors that were obtained from the same rate to begin with.

[3] The class uses the Observer pattern to determine when to convert the Black volatility into the local one. Recalculation is lazy; meaning that it happens only when the local volatility is actually used and not immediately upon a notification from an observable.

[4] The `evolve` method might have been overridden for efficiency, even in `expectation` were available. The default implementation in `StochasticProcess1D` would return $x\exp(\mu dt)\exp(\sigma dw)$, while the overridden one calculates $x\exp(\mu dt + \sigma dw)$.

Listing 6.6: Partial implementation of the `GeneralizedBlackScholesProcess` class.

```
class GeneralizedBlackScholesProcess
    : public StochasticProcess1D {
  public:
    GeneralizedBlackScholesProcess(
        const Handle<Quote>& x0,
        const Handle<YieldTermStructure>& dividendTS,
        const Handle<YieldTermStructure>& riskFreeTS,
        const Handle<BlackVolTermStructure>& blackVolTS,
        const boost::shared_ptr<discretization>& d =
                            boost::shared_ptr<discretization>());
    Real x0() const {
        return x0_->value();
    }
    Real drift(Time t, Real x) const {
        Real sigma = diffusion(t,x);
        Time t1 = t + 0.0001;
        return riskFreeRate_->forwardRate(t,t1,Continuous,...)
             - dividendYield_->forwardRate(t,t1,Continuous,...)
             - 0.5 * sigma * sigma;
    }
    Real diffusion(Time t, Real x) const;
    Real apply(Real x0, Real dx) const {
        return x0 * std::exp(dx);
    }
    Real expectation(Time t0, Real x0, Time dt) const {
        QL_FAIL("not implemented");
    }
    Real evolve(Time t0, Real x0, Time dt, Real dw) const {
        return apply(x0, discretization_->drift(*this,t0,x0,dt) +
                         stdDeviation(t0,x0,dt)*dw);
    }
    const Handle<Quote>& stateVariable() const;
    const Handle<YieldTermStructure>& dividendYield() const;
    const Handle<YieldTermStructure>& riskFreeRate() const;
    const Handle<BlackVolTermStructure>& blackVolatility() const;
    const Handle<LocalVolTermStructure>& localVolatility() const;
};
```

Finally, there's another design problem. As I mentioned previously, we already used the Black-Scholes process in the `AnalyticEuropeanEngine` shown as an example in chapter 2. However, if you look at the engine code in the library, you'll see that it doesn't use the process by means of the `StochasticProcess` interface; it just uses it as a bag of quotes and term structures. This suggests that the process is a jack of too many trades.

How can we fix it, then? We'll probably need a redesign. It's all hypothetical, of course; there's no actual code yet, just a few thoughts I had while I was reviewing the current code for this chapter. But it could go as follows.

First of all (and at the risk of being told once again that I'm overengineering) I'd separate the stochastic-process part of the class from the bag-of-term-structures part. For the second one, I'd create a new class, probably called something like `BlackScholesModel`. Instances of this class would be the ones passed to pricing engines, and would contain the full set of quotes and term structures; while we're at it, the interface should also allow the user to provide a local volatility, or to specify new ways to convert the Black volatility with some kind of Strategy pattern.

From the model, we could build processes. Depending on the level of coupling and the number of classes we prefer, we might write factory classes that take a model and return processes; we might add factory methods to the model class that return processes; or we might have the process constructors take a model. In any case, this could allow us to write process implementations that could be optimized for the given simulation. For instance, the factory (whatever it might be) might take as input the time nodes of the simulation and return a process that precomputed the rates $r(t_i)$ and $q(t_i)$ at those times, since they don't depend on the underlying value; if a type check told the factory that the volatility doesn't have smile, the process could precompute the $\sigma(t_i)$, too.

As for the x vs $\log(x)$ problem, I'd probably rewrite the process so that it's explicitly a process for the logarithm of the underlying: the `x0` method would return a logarithm, and so would the other methods such as `expectation` or `evolve`. The process would gain in consistency and clarity; however, since it would be no longer guaranteed that a process generates paths for the underlying, client code taking a generic process would also need a function or a function object that converts the value of the process variable into a value of the underlying. Such function might be added to the `StochasticProcess` interface, or passed to client code along with the process.

Back to some existing code. For an example of a well-behaved process, you can look at listing 6.7, which shows the `OrnsteinUhlenbeckProcess` class. The Ornstein-Uhlenbeck process is a simple one, whose feature of interest here is that its mean-reverting drift term $\theta(\mu - x)$ and its constant diffusion term σ can be integrated exactly. Therefore, besides the mandatory `drift` and `diffusion` methods, the class also overrides the `expectation` and `stdDeviation` methods

Listing 6.7: Implementation of the `OrnsteinUhlenbeckProcess` class.

```
class OrnsteinUhlenbeckProcess : public StochasticProcess1D {
  public:
    OrnsteinUhlenbeckProcess(Real speed,
                             Volatility vol,
                             Real x0 = 0.0,
                             Real level = 0.0);
    Real x0() const;
    ... // other inspectors
    Real drift(Time, Real x) const  {
        return speed_ * (level_ - x);
    }
    Real diffusion(Time, Real) const  {
        return volatility_;
    }
    Real expectation(Time t0, Real x0, Time dt) const  {
        return level_ + (x0 - level_) * std::exp(-speed_*dt);
    }
    Real stdDeviation(Time t0, Real x0, Time dt) const  {
        return std::sqrt(variance(t,x0,dt));
    }
    Real variance(Time t0, Real x0, Time dt) const {
        if (speed_ < std::sqrt(QL_EPSILON)) {
            return volatility_*volatility_*dt;
        } else {
            return 0.5*volatility_*volatility_/speed_*
                (1.0 - std::exp(-2.0*speed_*dt));
        }
    }
  private:
    Real x0_, speed_, level_;
    Volatility volatility_;
};
```

so that they implement the formulas for their exact results. The `variance` method (in terms of which `stdDeviation` is implemented) has two branches in order to prevent numerical instabilities; for small θ, the formula for the variance is replaced by its limit for $\theta \to 0$.

Finally, for an example of a multi-dimensional process, we'll have a look at the `StochasticProcessArray` class, sketched in listing 6.8. It doesn't model a specific process, but rather the composition in a single entity of N correlated

Listing 6.8: Partial implementation of the `StochasticProcessArray` class.

```
class StochasticProcessArray : public StochasticProcess {
  public:
    StochasticProcessArray(
        const std::vector<shared_ptr<StochasticProcess1D> >& ps,
        const Matrix& correlation)
    : processes_(ps), sqrtCorrelation_(pseudoSqrt(correlation)) {
        for (Size i=0; i<processes_.size(); i++)
            registerWith(processes_[i]);
    }
    // ...
    Disposable<Array> drift(Time t, const Array& x) const {
        Array tmp(size());
        for (Size i=0; i<size(); ++i)
            tmp[i] = processes_[i]->drift(t, x[i]);
        return tmp;
    }
    Disposable<Matrix> diffusion(Time t, const Array& x) const {
        Matrix tmp = sqrtCorrelation_;
        for (Size i=0; i<size(); ++i) {
            Real sigma = processes_[i]->diffusion(t, x[i]);
            std::transform(tmp.row_begin(i), tmp.row_end(i),
                           tmp.row_begin(i),
                           bind2nd(multiplies<Real>(),sigma));
        }
        return tmp;
    }
    Disposable<Array> expectation(Time t0, const Array& x0,
                                  Time dt) const;
    Disposable<Matrix> stdDeviation(Time t0, const Array& x0,
                                    Time dt) const;
    Disposable<Array> evolve(Time t0, const Array& x0,
                             Time dt, const Array& dw) const {
        const Array dz = sqrtCorrelation_ * dw;
        Array tmp(size());
        for (Size i=0; i<size(); ++i)
            tmp[i] = processes_[i]->evolve(t0, x0[i], dt, dz[i]);
        return tmp;
    }
  private:
    std::vector<shared_ptr<StochasticProcess1D> > processes_;
    Matrix sqrtCorrelation_;
};
```

one-dimensional processes. I'll use it here to show how correlation information can be included in a process.

Its constructor takes the vector of 1-D processes for the underlyings and their correlation matrix. The processes are stored in the corresponding data member, whereas the correlation is not: instead, the process precomputes and stores its square root.[5] The constructor also registers with each process, in order to forward any notifications they might send.

Most other methods, such as `initialValues`, `drift`, or `expectation` simply loop over the stored processes, calling their corresponding methods and collecting the results in an array. The `diffusion` method also loops over the processes, but combines the results with the correlation: it multiplies each row of its square root by the diffusion term of the corresponding process, and returns the results (if you multiply it by its transposed, you'll find the familiar terms $\sigma_i \rho_{ij} \sigma_j$ of the covariance matrix). The `stdDeviation` method does the same, but using the standard deviation of the underlying processes which also include the passed Δt.

This leaves us with the `evolve` method. If we knew that all the processes behaved reasonably (i.e., by adding the calculated variations to the values of their variables) we might just inherit the default implementation which takes the results of the `expectation` and `stdDeviation` methods, multiplies the latter by the array of random variates, and adds the two terms. However, we don't have such guarantee, and the method needs to be implemented in a different way. First, it deals with the correlation by multiplying the Gaussian variates by its square root, thus obtaining an array of correlated random variates. Then, it calls the `evolve` method of each one-dimensional process with the respective arguments and collects the results.

6.1.3. Random path generators

Before tackling path generation proper, we need a last basic component: a structure to hold the path itself. Listing 6.9 shows the `Path` class, which models a random path for a single variable. It contains the `TimeGrid` instance and the `Array` holding the node times and values, respectively, and provides methods to access and iterate over them. The method names are usually chosen to follow those of the standard containers, allowing for a more familiar interface; methods with domain-based names such as `time` are also available.

The `MultiPath` class, also shown in listing 6.9, holds paths for a number of underlying variables. It is basically a glorified container, holding a vector of `Path` instances and providing a few additional inspectors to uphold the law of Demeter.[6]

[5] That would be its matricial square root; that is, $\sqrt{A} = B$ if $BB^T = A$.

[6] For instance, if `p` is a `Multipath` instance, the additional methods allow us to write `p.pathSize()` instead of `p[0].length()`. Besides being uglier, the latter might also suggest that `p[1].length()` could be different.

The simplicity of the implementation is at the expense of some space: by storing a vector of `Path` instances, we're storing N identical copies of the time grid.

What remains to be done for path generation is now simply to connect the dots between the classes that I described so far. The logic for doing so is implemented in the `MultiPathGenerator` class template, sketched in listing 6.10; I'll leave it to you to decide whether this makes it a Factory.

Its constructor takes and stores the stochastic process followed by the variable to be modeled; the time grid to be used for generating the path nodes; a generator of random Gaussian sequences, which must be of the right dimension for the job (that is, at each draw it must return $N \times M$ numbers for N factors and M steps); and a boolean flag to specify whether or not it should use Brownian bridging, which at this time cannot be set to `true` and with which we're unfortunately stuck for backward compatibility.

The interface of the class provides a `next` method, returning a new random path (well, multipath, but cut me some slack here); and an `antithetic` method, returning the antithetic path of the one last returned by `next`.[7] Both methods forward to a private method `next(bool)`, which implements the actual logic. If the `brownianBridge` flag was set to `true`, the method bails out by raising an exception since the functionality is not yet implemented. Following the general principles of C++, which suggest to fail as early as possible, we should probably move the check to the constructor (or better yet, implement the thing; send me

[7] Unfortunately, the correctness of the `antithetic` method depends on its being called at the correct time, that is, after a call to `next`; but I can't think of any acceptable way out of this.

Aside: access patterns.

The basic inspectors in the `Path` class, such as `operator[]` and the iterator interface, return the values S_i of the underlying variable rather than the pairs (t_i, S_i) including the node times. While it could well be expected that `p[i]` return the whole node, we thought that in the most common cases a user would want just the value, so we optimized such access (its some kind of Huffman-coding principle, in which the most common operations should require the least typing; the Perl language shines at this, or so I'm told).

In a sad note about reuse (or lack thereof) if we were to add to the `Path` class a few methods to return whole nodes, we probably wouldn't use `std::pair<Time,Real>` to hold the data, but an inner `Node` struct. This is not because we instinctively leap at the most complex implementation—even though it may seem so at times—but because code such as `p.node(i).second` is not self-describing (wait, is `second` the time or the value?) The advantage of writing the above as `p.node(i).value` would offset the cost of defining a `Node` struct.

Listing 6.9: Interface of the `Path` and `MultiPath` classes.

```
class Path {
  public:
    Path(const TimeGrid& timeGrid, const Array& values);
    bool empty() const;
    Size length() const;
    Real operator[](Size i) const;
    Real at(Size i) const;
    Real& operator[](Size i);
    Real& at(Size i);
    Real front() const;
    Real& front();
    Real back() const;
    Real& back();
    Real value(Size i) const;
    Real& value(Size i);
    Time time(Size i) const;
    const TimeGrid& timeGrid() const;
    typedef Array::const_iterator iterator;
    typedef Array::const_reverse_iterator reverse_iterator;
    iterator begin() const;
    iterator end() const;
    reverse_iterator rbegin() const;
    reverse_iterator rend() const;
  private:
    TimeGrid timeGrid_;
    Array values_;
};

class MultiPath {
  public:
    MultiPath(const std::vector<Path>& multiPath);
    // ... more constructors...
    Size assetNumber() const;
    Size pathSize() const;
    const Path& operator[](Size j) const;
    const Path& at(Size j) const;
    Path& operator[](Size j);
    Path& at(Size j);
  private:
    std::vector<Path> multiPath_;
};
```

Listing 6.10: Sketch of the `MultiPathGenerator` class template.

```
template <class GSG>
class MultiPathGenerator {
  public:
    typedef Sample<MultiPath> sample_type;
    MultiPathGenerator(const shared_ptr<StochasticProcess>&,
                       const TimeGrid&,
                       GSG generator,
                       bool brownianBridge = false);
    const sample_type& next() const; { return next(false); }
    const sample_type& antithetic() const { return next(true); }
  private:
    const sample_type& next(bool antithetic) const;
    bool brownianBridge_;
    shared_ptr<StochasticProcess> process_;
    GSG generator_;
    mutable sample_type next_;
};
```


a patch if you do). As it is now, one can build a path generator and have the constructor succeed, only to find out later that the instance is unusable.

If Brownian bridging is not required, the method gets to work. First, it retrieves the random sequence it needs: a new one if we asked for a new path, or the latest sequence we used if we asked for an antithetic path. Second, it performs some setup. It gets a reference to the path to be built (which is a data member, not a temporary; more on this later); retrieves the array of the initial values of the variables and copies them at the beginning of the path; creates an array to hold the subsets of the random variables that will be used at each step; and retrieves the time grid to be used. Finally, it puts the pieces together. At each step, it takes as starting point the values at the previous one (it uses the same array where it stored the initial values, so the first step is covered, too); it retrieves from the time grid the current time `t` and the interval `dt` over which to evolve the variables; it copies the subset of the random variables that it needs (the first N for the first step, the second N for the second step, and so on, N being the number of factors; also, it negates them if it must generate an antithetic path); and to close the circle, it calls the `evolve` method of the process and copies the returned values at the correct places in the paths.

As I mentioned, the returned path is stored as a data member of the generator instance. This saves quite a few allocations and copies; but of course it also makes it impossible to share a path generator between threads, since they would race for

Listing 6.10 (continued).

```
template <class GSG>
const typename MultiPathGenerator<GSG>::sample_type&
MultiPathGenerator<GSG>::next(bool antithetic) const {
    if (brownianBridge_) {
        QL_FAIL("Brownian bridge not supported");
    } else {
        typedef typename GSG::sample_type sequence_type;
        const sequence_type& sequence_ =
            antithetic ? generator_.lastSequence()
                       : generator_.nextSequence();
        Size m = process_->size(), n = process_->factors();
        MultiPath& path = next_.value;
        Array asset = process_->initialValues();
        for (Size j=0; j<m; j++)
            path[j].front() = asset[j];
        Array temp(n);
        next_.weight = sequence_.weight;

        TimeGrid timeGrid = path[0].timeGrid();
        Time t, dt;
        for (Size i = 1; i < path.pathSize(); i++) {
            Size offset = (i-1)*n;
            t = timeGrid[i-1];
            dt = timeGrid.dt(i-1);
            if (antithetic)
                std::transform(sequence_.value.begin()+offset,
                               sequence_.value.begin()+offset+n,
                               temp.begin(),
                               std::negate<Real>());
            else
                std::copy(sequence_.value.begin()+offset,
                          sequence_.value.begin()+offset+n,
                          temp.begin());

            asset = process_->evolve(t, asset, dt, temp);
            for (Size j=0; j<m; j++)
                path[j][i] = asset[j];
        }
        return next_;
    }
}
```

the same storage. However, that's just the last nail in the coffin. Multithreading was pretty much out of the picture as soon as we stored the Gaussian number generator as a data member; the path generator can only be as thread-safe as its RNG—that is, not much, since most (if not all) of them store state. A much easier strategy to distribute work among threads would be to use different `MultiPathGenerator` instances, each with its own RNG. The problem remains of creating RNGs returning non-overlapping sequences; for algorithms that allow it (such as Sobol's) one can tell each instance to skip ahead a given number of draws.

To close the section, I'll mention that the library also provides a `PathGenerator` class for 1-D stochastic processes. It has some obvious differences (it calls the `StochasticProcess1D` interface and it creates `Path`s instead of `MultiPath`s; also, it implements Brownian bridging, which is a lot easier in the 1-D case) but other than that, it has the same logic as `MultiPathGenerator` and doesn't warrant being described in detail here.

6.2. Pricing on a path

There's not much to say on the `PathPricer` class template, shown in listing 6.11. It defines the interface that must be implemented by a path pricer in order to be used in a Monte Carlo model: an `operator()` taking a path and returning some kind of result. It is basically a function interface: in fact, if we were to write the framework now instead of ten years ago, we'd just use `boost::function` and dispense with `PathPricer` altogether.

The template arguments generalize on both the type of the argument, which can be a `Path`, a `MultiPath`, or a user-defined type if needed;[8] and the type of the result, which defaults of a simple `Real` (usually the value of the instrument) but might be an array of results, or some kind of structure. On the one hand, one might like to have the choice: an `Array` would be more appropriate for returning a set of homogeneous results, such as the amounts of a series of coupons, while

[8] For instance, when pricing instruments with an early-termination feature, one might want to save time by using a path class that generates nodes lazily and only if they are actually used.

Listing 6.11: Interface of the `PathPricer` class template.

```
template<class PathType, class ValueType=Real>
class PathPricer : public unary_function<PathType,ValueType> {
  public:
    virtual ~PathPricer() {}
    virtual ValueType operator()(const PathType& path) const=0;
};
```

Aside: stepping on one's own toes.

Unfortunately, there's quite a bit of memory allocation going on during multi-path generation. The obvious instances (the two `Array`s being created to store the current variable values and the current subset of the random numbers) could be avoided by storing them as data members; but on the one hand, we would have yet more mutable members, which is something we should use sparingly; and on the other hand, the allocation of the two arrays only happens once in a whole path generation, so it might not make a lot of difference.

Instead, the worst offender might be the underlying process, which at each step creates an `Array` instance to be returned from its `evolve` method. How can we fix this? In this case, I wouldn't return a reference to a data member if I can help it: the `StochasticProcess` class might just work in a multithreaded environment (well, if market data don't change during calculation, or if at least the process is kept frozen) and I'd rather not ruin it. Another possibility might be that the `evolve` method take a reference to the output array as an argument; but then, we (and for *we*, I mean whoever will implement a stochastic process) would have to be very carefully about aliasing.

The sad fact is that the natural way to write the `evolve` method would usually be something like (in pseudocode, and in the case of, say, two variables)

```
void evolve(const Array& x, Time t, Time dt,
            const Array& w, Array& y) {
    y[0] = f(x[0], x[1], t, dt, w[0], w[1]);
    y[1] = g(x[0], x[1], t, dt, w[0], w[1]);
}
```

where `x` is the input array and `y` is the output. However, it would be just as natural for a user to write

```
p->evolve(x,t,dt,w,x);
```

(that is, to pass the same `x` as both input and output) meaning to replace the old values of the variables with the new ones.

Well, you see the catch. When the first instruction in `evolve` writes into `y[0]`, it actually writes into `x[0]`. The second instruction then calls `g` with a value of `x[0]` which is not the intended one. Hilarity ensues.

Fixing this requires either the developer to make each process safe from aliasing, by writing

```
a = f(...); b = g(...); y[0] = a; y[1] = b;
```

or something to this effect; or the user to be mindful of the risk of aliasing, and never to pass the same array as both input and output. Either solution requires more constant care than I credit myself with.

So, where does this leave us? For the time being, the current interface might be an acceptable compromise. One future possibility is that we take away the allocation cost at the origin, by using some kind of allocation pool inside the `Array` class. As usual, we're open to suggestions.

a structure with named fields would be better suited for inhomogeneous results such as the value and the several Greeks of an instrument. On the other hand, the results will need to be combined, and therefore they'll need some basic algebra; the `Array` class already provides it, which makes it ready to use, whereas a results structure would need to define some operators in order to play well with the facilities described in the next section.

6.3. Putting it all together

Having read all about the different pieces of a model, you'd expect me to start assembling them and finally get a working Monte Carlo model. However, before doing so, we still have the task of picking a specific set of pieces among those available in our toolbox. This will be the subject of the next subsection—after which we'll build the model, developer's honor.

6.3.1. Monte Carlo traits

As I mentioned before, there's a number of choices we have to make in order to implement a Monte Carlo model. Depending on the dimension of the problem and the stochastic process we choose, we can use one- or multi-dimensional paths and generators; or we can use pseudo-random numbers or low-discrepancy sequences (and even though I haven't listed them, there's quite a few algorithms available for either type).

For at least one of those choices, dynamic polymorphism is not an option: one-dimensional and multi-dimensional generators can't have a common interface, since their methods have different input and return types. Therefore, we'll go with static polymorphism and supply the required classes to the model as template arguments.

The problem is, this can result very quickly in unwieldy declarations; the sight of something like

```
MonteCarloModel<
   MultiPathGenerator<
      InverseCumulativeRsg<SobolRsg, InverseCumulativeNormal> > >
```

can bring shudders to the most seasoned developer. Sure, `typedefs` can help; by using them, a user might assign mnemonics to default choices and shorten the declaration. Default template arguments might alleviate the pain, too. However, we went for a mechanism that also allowed us on the one hand, to define mnemonics for commonly used groups of related choices; and on the other hand, to add helper methods to the mix. To do this, we decided to use traits.

Listing 6.12 shows the default traits classes for pseudo-random and low-discrepancy number generation. First comes the `GenericPseudoRandom` class template. It takes as template arguments the type of a uniform pseudo-random number generator and that of an inverse-cumulative function object, and builds a

Listing 6.12: Example of random-number generation traits.

```
template <class URNG, class IC>
struct GenericPseudoRandom {
    typedef URNG urng_type;
    typedef InverseCumulativeRng<urng_type,IC> rng_type;
    typedef RandomSequenceGenerator<urng_type> ursg_type;
    typedef InverseCumulativeRsg<ursg_type,IC> rsg_type;
    enum { allowsErrorEstimate = 1 };
    static rsg_type make_sequence_generator(Size dimension,
                                            BigNatural seed) {
        ursg_type g(dimension, seed);
        return rsg_type(g);
    }
};

typedef GenericPseudoRandom<
            MersenneTwisterUniformRng,
            InverseCumulativeNormal>
    PseudoRandom;

template <class URSG, class IC>
struct GenericLowDiscrepancy {
    typedef URSG ursg_type;
    typedef InverseCumulativeRsg<ursg_type,IC> rsg_type;
    enum { allowsErrorEstimate = 0 };
    static rsg_type make_sequence_generator(Size dimension,
                                            BigNatural seed) {
        ursg_type g(dimension, seed);
        return rsg_type(g);
    }
};

typedef GenericLowDiscrepancy<
            SobolRsg,
            InverseCumulativeNormal>
    LowDiscrepancy;
```

number of other types upon it. The type of the passed generator itself is defined as `urng_type`—emphasis on "u" for uniform and "n" for number. Based on this type, it defines `rng_type`, no longer uniform since it uses the inverse-cumulative function to return numbers according to its distribution; `ursg_type`, where the "n" is replaced by "s" for sequence; and finally `rsg_type`, which generates sequences of numbers according to the passed distribution. The compile-time constant `allowErrorEstimate`, written as an enumeration to satisfy older compilers (it should really be a `static const bool`) tells us that this generator allows us to estimate the Monte Carlo error as function of the number of samples; and the helper function `make_sequence_generator` makes it easier to create a generator based on the passed inputs.

Then, we instantiate the class template with our weapons of choice. For the basic generator, that would be the `MersenneTwisterUniformRng` class; for the function object, the `InverseCumulativeNormal` class, since we'll most often want normally distributed numbers. The resulting traits class will be our default for pseudo-random generation; fantasy being not our strong suit, we defined it as the `PseudoRandom` class.

The `GenericLowDiscrepancy` class template is defined is a similar way, but with two differences. Since low-discrepancy numbers are generated in sequences,

Listing 6.13: Example of Monte Carlo traits.

```
template <class RNG = PseudoRandom>
struct SingleVariate {
    typedef RNG rng_traits;
    typedef Path path_type;
    typedef PathPricer<path_type> path_pricer_type;
    typedef typename RNG::rsg_type rsg_type;
    typedef PathGenerator<rsg_type> path_generator_type;
    enum { allowsErrorEstimate = RNG::allowsErrorEstimate };
};

template <class RNG = PseudoRandom>
struct MultiVariate {
    typedef RNG rng_traits;
    typedef MultiPath path_type;
    typedef PathPricer<path_type> path_pricer_type;
    typedef typename RNG::rsg_type rsg_type;
    typedef MultiPathGenerator<rsg_type> path_generator_type;
    enum { allowsErrorEstimate = RNG::allowsErrorEstimate };
};
```

the types for single-number generation are missing; and the enumeration tells us that we can't forecast the statistical error we'll get. We define the `LowDiscrepancy` traits class as the one obtained by selecting the `SobolRsg` class as our generator and, again, the `InverseCumulativeNormal` class as our function object.

Finally, we defined a couple of traits classes to hold types related to specific Monte Carlo functionality, such as the types of used paths, path generators, and path pricers. They are shown in listing 6.13: the `SingleVariate` class holds the types we need for 1-D models, while the `MultiVariate` class holds the types for multi-dimensional ones. They are both template classes, and take as their template argument a traits class for random-number generation.

By combining the provided RNG and Monte Carlo traits (or any traits classes that one might want to define, if one wants to use any particular type) not only we can provide a model with all the necessary information, but we can do it with a simpler and more mnemonic syntax, such as

```
MonteCarloModel<SingleVariate, LowDiscrepancy>;
```

the idea being to move some complexity from users to developers. We have to use some template tricks to get the above to work, but when it does, it's a bit more readable (and writable) for users. But that's for next section, in which we finally assemble a Monte Carlo model.

6.3.2. The Monte Carlo model

Listing 6.14 shows the `MonteCarloModel` class, which is the low-level workhorse of Monte Carlo simulations. It brings together path generation, pricing and statistics, and as such takes template arguments defining the types involved: a `MC` traits class defining types related to the simulation, a `RNG` traits class describing random-number generation, and a statistics class `S` defaulting to the `Statistics` class.[9] The `MC` class is a template template argument, so that it can be fed the `RNG` traits (as shown in the previous section; see for instance the `MultiVariate` class).

The class defines aliases for a few frequently used types; most of them are extracted from the traits, by instantiating the `MC` class template with the `RNG` class. The resulting class provides the types of the path generator and the path pricer to be used; from those, in turn, the type of the sample paths and that of the returned prices can be obtained.

The constructor takes the pieces that will be made to work together; at least a path generator (well, a pointer to one, but you'll forgive me for not spelling out all of them), a path pricer, and an instance of the statistics class, as well as a boolean flag specifying whether to use antithetic variates. Then, there are a few optional arguments related to control variates: another path pricer, the analytic value of the control variate, and possibly another path generator. Optional

[9] Don't worry. I'm not going off another tangent, even though an early outline of this chapter had a section devoted to statistics. If you're interested, the `Statistics` class is in appendix A.

Listing 6.14: Implementation of the `MonteCarloModel` class template.

```
template <template <class> class MC, class RNG,
                          class S = Statistics>
class MonteCarloModel {
  public:
    typedef MC<RNG> mc_traits;
    typedef RNG rng_traits;
    typedef typename MC<RNG>::path_generator_type
                                          path_generator_type;
    typedef typename MC<RNG>::path_pricer_type path_pricer_type;
    typedef typename path_generator_type::sample_type
                                           sample_type;
    typedef typename path_pricer_type::result_type result_type;
    typedef S stats_type;

    MonteCarloModel(
        const boost::shared_ptr<path_generator_type>&
                                                pathGenerator,
        const boost::shared_ptr<path_pricer_type>& pathPricer,
        const stats_type& sampleAccumulator,
        bool antitheticVariate,
        const boost::shared_ptr<path_pricer_type>& cvPathPricer
                    = boost::shared_ptr<path_pricer_type>(),
        result_type cvOptionValue = result_type(),
        const boost::shared_ptr<path_generator_type>& cvGenerator
                    = boost::shared_ptr<path_generator_type>());
    void addSamples(Size samples);
    const stats_type& sampleAccumulator(void) const;
  private:
    boost::shared_ptr<path_generator_type> pathGenerator_;
    boost::shared_ptr<path_pricer_type> pathPricer_;
    stats_type sampleAccumulator_;
    bool isAntitheticVariate_;
    boost::shared_ptr<path_pricer_type> cvPathPricer_;
    result_type cvOptionValue_;
    bool isControlVariate_;
    boost::shared_ptr<path_generator_type> cvPathGenerator_;
};
```

Listing 6.14 (continued).

```
template <template <class> class MC, class RNG, class S>
MonteCarloModel<MC,RNG,S>::MonteCarloModel(
    const boost::shared_ptr<path_generator_type>& pathGenerator,
    ... other arguments... )
: pathGenerator_(pathGenerator), ... other data... {
   if (!cvPathPricer_)
      isControlVariate_ = false;
   else
      isControlVariate_ = true;
}

template <template <class> class MC, class RNG, class S>
void MonteCarloModel<MC,RNG,S>::addSamples(Size samples) {
   for (Size j = 1; j <= samples; j++) {
      sample_type path = pathGenerator_->next();
      result_type price = (*pathPricer_)(path.value);

      if (isControlVariate_) {
         if (!cvPathGenerator_) {
            price += cvOptionValue_-(*cvPathPricer_)(path.value);
         } else {
            sample_type cvPath = cvPathGenerator_->next();
            price +=
                cvOptionValue_-(*cvPathPricer_)(cvPath.value);
         }
      }

      if (isAntitheticVariate_) {
         path = pathGenerator_->antithetic();
         result_type price2 = (*pathPricer_)(path.value);
         if (isControlVariate_) {
            ... adjust the second price as above
         }
         sampleAccumulator_.add((price+price2)/2.0, path.weight);
      } else {
         sampleAccumulator_.add(price, path.weight);
      }
   }
}
```

arguments might not be the best choice, since they make it possible to pass a control variate path pricer and not the corresponding analytic value; it would have been safer to have a constructor with no control variate arguments, and another constructor with both path pricer and analytic value being mandatory and with an optional path generator. However, the current version saves a few lines of code. The constructor copies the passed arguments into the corresponding data members, and sets another boolean flag based on the presence or the lack of the control-variate arguments.

The main logic is implemented in the `addSamples` method. It's just a loop: draw a path, price, add the result to the statistics; but it includes a bit of complication in order to take care of variance reduction. It takes the number of samples to add; for each of them, it asks the path generator for a path, passes the path to the pricer, and gets back a price. In the simplest case, that's all there is to it; the price can just be added to the statistics (together with the corresponding weight, also returned from the generator) and the loop can start the next iteration. If the user passed control-variate data, instead, things get more interesting. If no path generator were specified, we pass to the alternate pricer the same path we used for the main one; otherwise, we ask the second generator for a path and we use that one. In both cases, we adjust the baseline price by subtracting the simulated price of the control and adding its analytic value.

It's not over yet. If the user also asked for antithetic variates, we repeat the same dance (this time asking the generator, or the generators, for the paths antithetic to the ones we just used) and we add to the statistics the average of the regular and antithetic prices; if not, we just add the price we obtained on the original paths. Lather, rinse, and repeat until the required number of samples is reached.

Finally, the full results (mean price and whatnot) can be obtained by calling the `sampleAccumulator` method, which returns a reference to the stored statistics. "Accumulator" is STL lingo; we should probably have used a method name taken from the financial domain instead. Such as, I don't know, "statistics." Oh well.

6.3.3. Monte Carlo simulations

Up one step in complexity and we get to the `McSimulation` class template, sketched in listing 6.15. Its job is to drive the simple-minded `MonteCarloModel` towards a goal, be it a required accuracy or number of samples. It can be used (and it was designed) as a starting point to build a pricing engine.

`McSimulation` follows the Template Method pattern. It asks the user to implement a few pieces of behavior, and in return it provides generic logic to instantiate a Monte Carlo model and run a simulation. Derived classes must define at least the `pathPricer` method, that returns the path pricer to be used; the `pathGenerator` method, that returns the path generator; and the `timeGrid` method, that returns the grid describing the nodes of the simulation. Other methods, returning the objects to be used for control variates, might or might not be defined; `McSimulation`

Listing 6.15: Sketch of the `McSimulation` class template.

```
template <template <class> class MC, class RNG,
          class S = Statistics>
class McSimulation {
  public:
    typedef
      typename MonteCarloModel<MC,RNG,S>::path_generator_type
        path_generator_type;
    typedef typename MonteCarloModel<MC,RNG,S>::path_pricer_type
        path_pricer_type;
    typedef typename MonteCarloModel<MC,RNG,S>::stats_type
        stats_type;
    typedef typename MonteCarloModel<MC,RNG,S>::result_type
        result_type;
    virtual ~McSimulation() {}
    result_type value(Real tolerance,
                      Size maxSamples = QL_MAX_INTEGER,
                      Size minSamples = 1023) const;
    result_type valueWithSamples(Size samples) const;
    void calculate(Real requiredTolerance,
                   Size requiredSamples,
                   Size maxSamples) const;
    const stats_type& sampleAccumulator(void) const;
  protected:
    McSimulation(bool antitheticVariate,
                 bool controlVariate);
    virtual shared_ptr<path_pricer_type> pathPricer() const = 0;
    virtual shared_ptr<path_generator_type> pathGenerator()
                                                        const = 0;
    virtual TimeGrid timeGrid() const = 0;
    virtual shared_ptr<path_pricer_type>
    controlPathPricer() const;
    virtual shared_ptr<path_generator_type>
    controlPathGenerator() const;
    virtual result_type controlVariateValue() const;
    virtual shared_ptr<PricingEngine> controlPricingEngine()
                                                        const;
    mutable shared_ptr<MonteCarloModel<MC,RNG,S> > mcModel_;
    bool antitheticVariate_, controlVariate_;
};
```

Listing 6.15 (continued).

```
template <template <class> class MC, class RNG, class S>
typename McSimulation<MC,RNG,S>::result_type
McSimulation<MC,RNG,S>::value(Real tolerance,
                              Size maxSamples,
                              Size minSamples) const {
    ...
    Real error = mcModel_->sampleAccumulator().errorEstimate();
    while (error > tolerance) {
        QL_REQUIRE(sampleNumber < maxSamples, ...);
        Real order = (error*error)/(tolerance*tolerance);
        Size nextBatch =
            std::max<Size>(sampleNumber*order*0.8-sampleNumber,
                           minSamples));
        nextBatch = std::min(nextBatch, maxSamples-sampleNumber);
        sampleNumber += nextBatch;
        mcModel_->addSamples(nextBatch);
        error = mcModel_->sampleAccumulator().errorEstimate();
    }
    return mcModel_->sampleAccumulator().mean();
}

template <template <class> class MC, class RNG, class S>
void McSimulation<MC,RNG,S>::calculate(Real requiredTolerance,
                                       Size requiredSamples,
                                       Size maxSamples) const {
    if (!controlVariate_) {
        mcModel_ = shared_ptr<MonteCarloModel<MC,RNG,S> >(
            new MonteCarloModel<MC,RNG,S>(
                      pathGenerator(), pathPricer(),
                      S(), antitheticVariate_));
    } else {
        ... // same as above, but passing the control-variate args, too.
    }

    if (requiredTolerance != Null<Real>())
        this->value(requiredTolerance, maxSamples);
    else
        this->valueWithSamples(requiredSamples);
}
```

provides default implementations that return null values, so derived classes that don't want to use the control variate technique can just forget about it.

Based on such methods, `McSimulation` provides most of the behavior needed by an engine. Its constructor takes two boolean flags specifying whether it should use either antithetic or control variates; the second will only matter if the derived class implements the required methods.

The `value` method adds samples to the underlying model until the estimated error matches a required tolerance.[10] It looks at the current number of samples n and the current error ϵ, estimates the number of samples N that will be needed to reach the given tolerance τ as $N = n \times \epsilon^2/\tau^2$ (since of course $\epsilon \propto 1/\sqrt{n}$), and adds a new batch of samples $N - n$ that gets the total closer to the estimated number; then it assesses the error again, and repeats the process as needed.

The `valueWithSamples` method just adds a batch of samples so that their total number matches the required one; its implementation is not shown here because of space constraints, but is simple enough.

Finally, the `calculate` method runs a complete simulation. It takes as arguments either a required tolerance or a required number of samples,[11] as well as a maximum number of samples; it instantiates a `MonteCarloModel` instance, with the `controlVariate_` flag determining whether to pass the control-variate arguments to the constructor; and it calls either the `value` or the `valueWithSamples` method, depending on what goal was required.

In next section, I'll show an example of how to use the `McSimulation` class to build a pricing engine; but before that, let me point out a few ways in which it could be improved.

First of all, it currently implements logic to run a simulation based on two criteria (accuracy or total number of samples) but of course, more criteria could be added. For instance, one could run a simulation until a given clock time is elapsed; or again, one could add several batches of samples, look at the result after each one, and stop when convergence seems to be achieved. This suggests that the hard-coded `value` and `valueWithSamples` methods could be replaced by an instance of the Strategy pattern, and the switch between them in the `calculate` method by just a call to the stored strategy.

In turn, this would also remove the current ugliness in `calculate`: instead of passing the whole set of arguments for both calculations and giving most of them to a null value (like in the good old days, where languages could not overload methods) one would pass only the required arguments to the strategy object, and then the strategy object to `calculate`.

Finally, the presence of `controlPricingEngine` method is a bit of a smell. The implementation of `McSimulation` doesn't use it, so it's not strictly a part of

[10] Of course, this needs an error estimate, so it won't work with low-discrepancy methods.

[11] One of the arguments must be null, but not both.

Aside: synchronized walking.

Both the `MonteCarloModel` and `McSimulation` class templates allow one to define an alternate path generator for control variates. Note, however, that this feature should be used with some caution. For this variance-reduction technique to work, the control-variate paths returned from the second generator must be correlated with the regular ones, which basically means that the two path generators must use two identical random-number generators: same kind, dimensionality, and seed (if any).

Unfortunately, this constraint rules out quite a few possibilities. For instance, if you're using a stochastic volatility model, such as the Heston model, you might be tempted to use the Black-Scholes model as control variate. No such luck: for n time steps, the Heston process requires $2n$ random numbers (n for the stock price and n for its volatility) while the Black-Scholes process just needs n. This makes it impossible to keep the two corresponding path generators in sync.

In practice, you'll have a use case for the alternate path generator if you have a process with a number of parameters which is not analytically tractable in the generic case, but has a closed-form solution for your option value if some of the parameters are null. If you're so lucky, you can use a fully calibrated process to instantiate the main path generator, and another instance of the process with the null parameters to generate control-variate paths.

the Template Method pattern and probably shouldn't be here. However, a couple of other classes (both inheriting from `McSimulation`, but otherwise unrelated) declare it and use it to implement the `controlVariateValue` method; therefore, leaving it here might not be the purest of designs but prevents some duplication.

6.3.4. Example: basket option

To close this chapter, I'll show and discuss an example of how to build a pricing engine with the Monte Carlo machinery I described so far. The instrument I'll use is a simple European option on a basket of stocks, giving its owner the right to buy or sell the whole basket at an given price; the quantities of each of the stocks in the basket are also specified by the contract.

For brevity, I won't show the implementation of the instrument class itself, `BasketOption`.[12] It would be quite similar to the `VanillaOption` class I've shown in chapter 2, with the addition of a data member for the quantities (added to both the instrument and its `arguments` class). Also, I won't deal with the Greeks; but if the `BasketOption` class were to define them, methods such as `delta` and `gamma` would return a vector.

[12] This class is not the same as the `BasketOption` class implemented in QuantLib. The one used here is simplified for illustration purposes.

Listing 6.16: Implementation of the `McEuropeanBasketEngine` class template.

```
template <class RNG = PseudoRandom, class S = Statistics>
class MCEuropeanBasketEngine
    : public BasketOption::engine,
      private McSimulation<MultiVariate,RNG,S> {
  public:
    typedef McSimulation<MultiVariate,RNG,S> simulation_type;
    typedef typename simulation_type::path_generator_type
        path_generator_type;
    ... // same for the other defined types
    MCEuropeanBasketEngine(
            const shared_ptr<StochasticProcess>&,
            const Handle<YieldTermStructure>& discountCurve,
            Size timeSteps,
            Size timeStepsPerYear,
            bool antitheticVariate,
            Size requiredSamples,
            Real requiredTolerance,
            Size maxSamples,
            BigNatural seed);
    void calculate() const {
        simulation_type::calculate(requiredTolerance_,
                                   requiredSamples_,
                                   maxSamples_);
        const S& stats = this->mcModel_->sampleAccumulator();
        results_.value = stats.mean();
        if (RNG::allowsErrorEstimate)
            results_.errorEstimate = stats.errorEstimate();
    }
  private:
    TimeGrid timeGrid() const;
    shared_ptr<path_generator_type> pathGenerator() const;
    shared_ptr<path_pricer_type> pathPricer() const;
    shared_ptr<StochasticProcess> process_;
    Handle<YieldTermStructure> discountCurve_;
    Size timeSteps_, timeStepsPerYear_;
    Size requiredSamples_;
    Size maxSamples_;
    Real requiredTolerance_;
    BigNatural seed_;
};
```

Listing 6.16 (continued).

```
template <class RNG, class S>
TimeGrid MCEuropeanBasketEngine<RNG,S>::timeGrid() const {
    Time T = process_->time(arguments_.exercise->lastDate());
    if (timeSteps_ != Null<Size>()) {
        return TimeGrid(T, timeSteps_);
    } else if (timeStepsPerYear_ != Null<Size>()) {
        Size steps = timeStepsPerYear_*T;
        return TimeGrid(T, std::max<Size>(steps, 1));
    } else {
        QL_FAIL("time steps not specified");
    }
}


template <class RNG, class S>
shared_ptr<typename MCEuropeanBasketEngine<RNG,S>::
           path_generator_type>
MCEuropeanBasketEngine<RNG,S>::pathGenerator() const {
    Size factors = processes_->factors();
    TimeGrid grid = timeGrid();
    Size steps = grid.size() - 1;
    typename RNG::rsg_type gen =
        RNG::make_sequence_generator(factors*steps,
                                     seed_);
    return shared_ptr<path_generator_type>(
              new path_generator_type(process_, grid, gen));
}


template <class RNG, class S>
shared_ptr<typename MCEuropeanBasketEngine<RNG,S>::
           path_pricer_type>
MCEuropeanBasketEngine<RNG,S>::pathPricer() const {
    Date maturity = arguments_.exercise->lastDate();
    return shared_ptr<path_pricer_type>(
        new EuropeanBasketPathPricer(
            arguments_.payoff, arguments_.quantities,
            discountCurve_->discount(maturity)));
}
```

The main class in this example is the one implementing the pricing engine. It is the `MCEuropeanBasketEngine` class template, shown in listing 6.16. As expected, it inherits publicly from the `BasketOption::engine` class; however, it also inherits privately from the `McSimulation` class template.

In idiomatic C++, the use of private inheritance denotes an "is implemented in terms of" relationship. We don't want public inheritance here, since that would imply an "is a" relationship; in our conceptual model, `MCEuropeanBasketEngine` is a pricing engine for the basket option and not a simulation that could be used on its own. The use of multiple inheritance is shunned by some, and in fact it could be avoided in this case; our engine might use composition instead, and contain an instance of `McSimulation`. However, that would require inheriting a new simulation class from `McSimulation` in order to implement its purely virtual methods (such as `pathGenerator` or `pathPricer`) and would have the effect to make the design of the engine more complex; whereas, by inheriting from `McSimulation`, we can define such methods in the engine itself.[13] Finally, note the template parameters: we leave to the user the choice of the RNG traits, but since we're modeling a basket option we specify the `MultiVariate` class as the Monte Carlo traits.

The constructor (whose implementation is not shown for brevity) takes the stochastic process for the underlyings, a discount curve, and a number of parameters related to the simulation. It copies the arguments into the corresponding data members (except for the `antitheticVariate` flag, which is passed to the McSimulation constructor) and registers the newly-built instance as an observer of both the stochastic process and the discount curve.

The `calculate` method, required by the `PricingEngine` interface, calls the method by the same name from `McSimulation` and passes it the needed parameters; then, it retrieves the statistics and stores the mean value and, if possible, the error estimate. This behavior is not specific of this particular option, and in fact it could be abstracted out in some generic `McEngine` class; but this would muddle the conceptual model. Such a class would inherit from `McSimulation`, but it could be reasonably expected to inherit from `PricingEngine`, too. However, it we did that, our engine would inherit from both `McEngine` and `BasketOption::engine`, leading to the dreaded inheritance diamond. On the other hand, if we didn't inherit it from `PricingEngine` (calling it `McEngineAdapter` or something) it would add more complexity to the inheritance hierarchy, since it would be an additional layer between `McSimulation` and our engine, and wouldn't remove all the scaffolding anyway: our engine would still need to define a `calculate` method forwarding to the one in the adapter. Thus, I guess we'll leave it at that.

[13] In order to avoid this dilemma, we would have to rewrite the `McSimulation` class so that it doesn't use the Template Method pattern; that is, we should make it a concrete class which would be passed the used path generator and pricer as constructor arguments. Apart from breaking backward compatibility, I'm not sure this would be worth the hassle.

Listing 6.17: Sketch of the `EuropeanMultiPathPricer` class.

```
class EuropeanBasketPathPricer : public PathPricer<MultiPath> {
  public:
    EuropeanBasketPathPricer(const shared_ptr<Payoff>& payoff,
                             const vector<Real>& quantities,
                             DiscountFactor discount);
    Real operator()(const MultiPath& path) const {
        Real basketValue = 0.0;
        for (Size i=0; i<quantities_.size(); ++i)
            basketValue += quantities_[i] * path[i].back();
        return (*payoff)(basketValue) * discount_;
    }
  private:
    shared_ptr<Payoff> payoff_;
    vector<Real> quantities_;
    DiscountFactor discount_;
};
```


The rest of listing 6.16 shows the three methods required to implement the `McSimulation` interface and turn our class into a working engine. The `timeGrid` method build the grid used by the simulation by specifying the end time (given by the exercise date) and the number of steps. Depending on the parameters passed to the engine constructor, they might have been specified as a given total number (the first `if` clause) or a number per year (the second). In either case, the specification is turned into a total number of steps and passed to the `TimeGrid` constructor. If neither was specified (the `else` clause) an error is raised.

The `pathGenerator` method asks the process for the number of factors, determines the number of time steps from the result of the `timeGrid` method I just described, builds a random-sequence generator with the correct dimensionality (which of course is the product of the two numbers above) and uses it together with the underlying process and the time grid to instantiate a multi-path generator.

Finally, the `pathPricer` method collects the data required to determine the payoff on each path—that is, the payoff object itself, the quantities and the discount at the exercise date, which the method precalculates—and builds an instance of the `EuropeanBasketPathPricer` class, sketched in listing 6.17.

The path pricer stores the passed data and uses them to calculate the realized value of the option in its `operator()` overloading, whose implementation calculates the value of the basket at maturity and returns the corresponding payoff discounted to the present time. The calculation of the basket value is a straightforward loop that combines the stored quantities with the asset values at maturity, retrieved from the end points of the respective paths.

Aside: need-to-know basis.

As usual, I didn't show in which files one should put the several classes I described. Well, as a general principle, the more encapsulation the better; thus, to begin with, helper classes should be hidden from the public interface. For instance, the `EuropeanBasketPathPricer` class is only used by the engine and could be hidden from client code. The best way would be to define such classes inside an anonymous namespace in a `.cpp` file, but that's not always possible: in our case, the engine is a class template, so that's not an option. The convention we're using in QuantLib is that helper classes that must be declared in a header file are placed in a nested namespace `detail`, and the user agrees to be a gentleman (or a lady, of course) and to leave it alone.

If, later on, we find out that we need the class elsewhere (for instance, because we want to use `EuropeanBasketPathPricer` as a control-variate path pricer for another engine) we can move it to a header file—or, if it's already in one, to the main `QuantLib` namespace—and make it accessible to everybody.

At this point, the engine is completed; but it's still a bit unwieldy to instantiate. We'd want to add a factory class with a fluent interface, so that one can write

```
engine = MakeMcEuropeanBasketEngine<PseudoRandom>(process,
                                                  discountCurve)
         .withTimeStepsPerYear(12)
         .withAbsoluteTolerance(0.001)
         .withSeed(42)
         .withAntitheticVariate();
```

but I'm not showing its implementation here. There's plenty of such examples in the library for your perusal.

Also, you'll have noted that I kept the example as simple as possible, and for that reason I avoided a few possible generalizations. For instance, another pricer might need some other specific dates besides the maturity, and the time grid should be built accordingly; or the payoff might have been path-dependent, so that the path pricer should look at several path values besides the last. But I don't think you'll have any difficulties in writing the corresponding code.

After all this, you might still have a question: how generic is this engine, really? Well, somewhat less that I'd like. It is generic in the sense that you can plug in any process for N asset prices, and it will work. It can even do more exotic stuff, such as quanto effects: if you go the traditional way and model the effect as an correction for the drift of the assets, you can write (or better yet, decorate) your process so that its `drift` or `evolve` method takes into account the correction, and you'll be able to use it without changing the engine code.

However, if you want to use a process that models other stochastic variables besides the asset prices (say, a multi-asset Heston process) you're likely to get into some trouble. The problem is that you'll end up in the `operator()` of the path pricer with N quantities and $2N$ paths, without any indication of which are for the prices and which for the volatilities. How should they be combined to get the correct basket price? Of course, one can write a specific pricer for any particular process; but I'd like to provide something more reusable.

One simple solution is for the process and the pricer to share some coding convention. For instance, if you arrange the process so that the first N paths are those of the prices and the last M are those of any other variables, the pricer in listing 6.17 will work; note that it loops over the number of quantities, not the site of the process. However, this leaves one wide open to errors that will go undetected if a process doesn't conform to the convention.

Another possibility that comes to mind is to decorate the process with a layer that would show the asset prices and hide the extra variables. The decorator would appear as a process requiring the same number of random variates as the original, but exposing less stochastic variables. It would have to store some state in order to keep track of the hidden variables (which wouldn't be passed to its methods): its `evolve` method, to name one, would have to take the current prices, add the extra variables stored in the instance, call `evolve` in the original process, store the new values of the extra variables so that they're available for next call, and return only the new asset prices.

However, I'm not a fan of this solution. It would only work when the decorated methods are called in the expected order, and would break otherwise; for instance, if you called a method twice with the exact same arguments (starting prices and random variates) you would get different return, due to the changed internal state. The long way to express this is that the methods of the decorated process would lose referential transparency. The short way is that its interface would be a lie.

A more promising possibility (albeit one that requires some changes to the framework) would be to let the path generator do the filtering, with some help from the process. If the process could provide a method returning some kind of mask—say, a list of the indices of the assets into the whole set of variables, or a vector of booleans where the i-th element would be set to `true` if the i-th path corresponds an asset price—then the generator could use it to drive the evolution of the variables correctly, and at the same time write in the multi-path only the asset prices. To make the whole thing backward-compatible, the mask would be passed to the generator only optionally, and the base `StochasticProcess` class would have a default implementation of the method returning an empty mask. If either way no mask were provided, the generator would revert to the current behavior. If needed, the method could be generalized to return other masks besides that for the prices; for instance, a variance swap would be interested in the volatility but not the price.

Finally, note that some assumptions are built right into the engine and cannot be relaxed without rewriting it. For instance, we're assuming that the discount factor for the exercise date is deterministic. If you wrote a process that models stochastic interest rates as well as asset prices, and therefore wanted to discount on each path according to the realization of the rates on that path, you'd have to write a custom engine and path pricer to be used with your specific process. Fortunately, that would be the only things you'd have to write; you'd still be able to reuse the other pieces of the framework.